Zpravodaj Československého sdružení uživatelů TeXu

Arnošt Štědrý

PostScript pro programátory, vědce i inženýry

Zpravodaj Československého sdružení uživatelů TeXu, Vol. 11 (2001), No. 1-3, 1–39

Persistent URL: http://dml.cz/dmlcz/150206

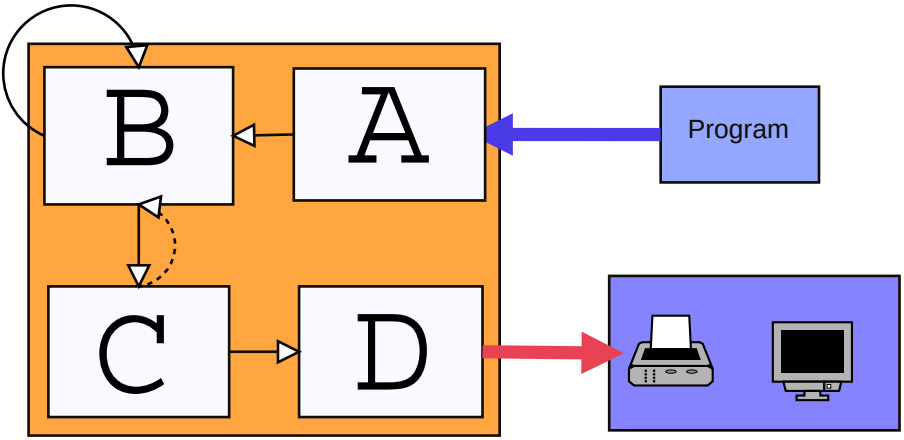
Základní mechanismy PostScriptu

Dříve, než se pustíme do odhalování všech PostScriptových tajemství, načrtneme si několik konceptů, podle kterých je jazyk PostScript navržen.

Jednak je dobré si uvědomit, že PostScript je interpretovaný jazyk. To znamená, že součástí každého výstupního zařízení podporující PostScript je interpret, program, který se stará o vykonávání příslušných PostScriptových příkazů.

Interpret

Strukturu interpretu PostScript můžeme popsat jako soustavu několika modulů. Schematicky je lze zakreslit obrázkem:



Obrázek 1: A: vstupní modul, B: nahrazovací modul, C: prováděcí modul, D: rastrovací modul

Spolupráce mezi moduly funguje zhruba řečeno takto: Modul obdrží na svůj vstup objekt, ten zpracuje a výsledek, což je většinou posloupnost jiných objektů, předává postupně jak vzniká dalšímu modulu. Nejmenší jednotkou komunikace

je tedy objekt. Objektem může být např. PostScriptový operátor, číslo nebo řetězec. Rozeberme si nyní práci jednotlivých modulů.

vstupní modul Čte posloupnost znaků ze vstupu a na základě syntaktických pravidel z ní vytváří objekty typu *jméno* (name) s přiřazeným atributem proveditelný/neproveditelný. V knize ho budeme někdy přirovnávat k ústům interpretu.

nahrazovací modul Pokud tento modul dostane jako vstup neproveditelný objekt, postupuje ho okamžitě dalšímu modulu. Pokud naopak přijde objekt proveditelný nahradí jej dle nahrazovacích tabulek (slovníků), většinou posloupností jiných objektů. Jsou-li výsledné objekty již neproveditelné, či tzv. operátory, tj. vestavěné PostScriptové funkce, odešle je k dalšímu zpracování. V opačném případě pokračuje v jejich nahrazování, než jsou všechny objekty buď neproveditelné, nebo operátory. Tento modul tedy rozkládá vyšší objekty na nižší. Lze ho přirovnat k trávicímu traktu.

prováděcí modul K tomuto modulu přicházejí již pouze neproveditelné objekty, či operátory. Neproveditelné objekty jsou ukládány na zásobník operandů, proveditelné jsou prováděny, tj. provede se funkce kterou reprezentují. Výjimkou jsou pouze grafické funkce, jejichž interpretace se přenechává ratsrovacímu modulu.

rastrovací modul (Raster Image Procesor) Mnoho vestavěných funkcí má za úkol vykreslování různých obrazců, či sazbu textu. Tyto příkazy jsou většinou vektorové, tj. nezávislé na rozlišení. Oproti tomu výstupní zařízení na rozlišení závislé je. Převod z vektorového zápisu na rastrovaný obrázek má na starosti právě RIP.

Přerušovaná šipka na obrázku znázorňuje, že prováděcí modul řídí nahrazovací modul přepisováním nahrazovacích tabulek — slovníků. Proto je velmi důležité pochopit, v jakém pořadí jsou objekty zpracovávány, neboť prováděcí modul může nahrazovacímu modulu změnit nahrazovací tabulky doslova uprostřed rozdělané práce. Hlouběji se tímto problémem budeme zabývat v kapitole .

Grafické funkce

PostScript je jazyk určený ke grafickému popisu stránky. Takového úkolu se lze zhostit například výčtem barev bod po bodu. To ovšem není zcela univerzální postup, neboť je určen pouze konkrétnímu zařízení (s daným rozlišením a určitými grafickými schopnostmi), navíc u velkých stránek přibudou i problémy s velikostí takového popisu. Proto autoři PostScriptu zvolili tzv. vektorový formát. V praxi to znamená, že většina objektů je reprezentována pomocí matematických objektů, jako jsou čáry, Bèziérovy křivky, jejichž napojováním vznikají cesty. Cesty pak můžeme vykreslovat, oblasti jimi ohraničené následně vyplňovat barvou, vzory atd. Protože písmena jsou reprezentována také pomocí cest, lze s nimi nakládat podobně, mimoto PostScript nabízí řadu nástrojů i k sazbě textu.

Rastrovací modul
PostScript vychází z následující ideje: Autora dokumentu (stránky) nezajímá, jaké jsou možnosti konkrétního výstupního zařízení. Tj. neví, jaké má rozlišení, dovede-li tisknout barevně, dokáže-li provádět separace barev atd. O realizaci rastrované podoby s přihlédnutím k všem specialitám se stará poslední modul PostScriptového interpretu — RIP (Raster Image Procesor). Ten definitivně rozhodne o barvách jednotlivých bodů.

Zásobníky a objekty PostScriptu

Program v PostScriptu, jako ostatně každý program, je posloupnost písmen, číslic a speciálních znaků. Písmena a číslice se seskupují do slov, kterými v PostScriptu označujeme *objekty*. Vše, s čím se nadále v PostScriptu setkáme, bude nějaký objekt. Objekty mohou mít různé měnitelné vlastnosti (attributes). Pro začátek rozčleňme objekty na vykonavatelné (executable) a nevykonavatelné (literal)[1]. Vykonavatelné objekty budou hrát roli procedur, nevykonavatelné objekty budou zastupovat data. Vykonavatelné objekty lze ještě rozdělit na vestavěné PostScriptové *operátory* a uživatelem definované *procedury*. Vykonavatelné objekty budou tedy hrát roli funkcí či operací, nevykonavatelné objekty reprezentují data, operandy. Pro správné pochopení principů PostScriptu je třeba si uvědomit toto: Názvy vykonavatelných objektů jsou pouhá jména (name object)[2], kterým až PostScriptový interpret dává jejich skutečné významy závislé na kontextu. V kapitole si ukážeme případ, kdy se jméno `add` bude interpretovat na různých místech různě. Jména musí být sestavena tak, aby interpret okamžitě poznal, jedná-li se o vykonavatelný, či nevykonavatelný objekt.

Jak se ale z posloupnosti znaků dostaneme k posloupnosti objektů? Vstupní modul PostScriptového interpretu — mohli bychom ho přirovnat k ústům — načítá ze vstupu posloupnost znaků a převádí ji v posloupnost objektů (jmen) s atributy vykonavatelný/nevykonavatelný. Takže je-li na vstupu posloupnost písmen `add` oddělená od ostatních písmen mezerami, vstupní modul ji načte a konvertuje na jméno `add`, kterému podle svých vnitřních pravidel přidá atribut vykonavatelnosti. Další moduly zacházejí s jménem už s jako nedělitelným objektem[3]. Jakmile vstupní modul vytvoří z posloupnosti znaků objekt jméno, předává ho okamžitě k dalšímu zpracování. Jako jméno bude interpretována jakákoliv posloupnost písmen a čísel neobsahující mezery, speciální symboly a oddělovače vyjma tečky. Pokud nelze tuto posloupnost interpretovat jako číslo

[1] Lepší překlad adjektiva literal by zněl spíše „doslovný“ (objekt), čímž je míněno, že takový objekt reprezentuje sám sebe.

[2] Je dobré si všimnout, že i jméno je objekt

[3] Jména tedy od této chvíle nejsou objekty složené z jednotlivých znaků jako třeba řetězce.

(viz níže) je interpretována jako jméno vykonavatelného objektu. Zvláštní funkci budou mít jména začínající speciálním symbolem `/`. Vstupní modul jim nepřidělí atribut vykonavatelnosti, pokud nanastane nějaký neobvyklý případ, dostanou se tudíž do zásobníku operandů. Lomítku zde chápeme pouze jako příznak nevykonavatelnosti a pro další zpracování je jako znak eleminováno. Příklady sekvencí, které se převádí na jména s atributem proveditelnosti, jsou tedy: `abc`, `Odd`, `2A`, `1-3`, `@ano`.

PostScriptový interpret dostává od vstupního modulu posloupnost jmen a speciálních symbolů. Pokud má jméno atribut vykonavatelnosti, vyhledá si jeho definici a provede ji. Pokud rozpozná objekt jako nevykonavatelný, uloží jej k dalšímu zpracování.

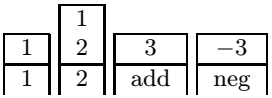
Speciální znaky mají, poplatny svému označení, v PostScriptu zvláštní funkce. Jsou to zejména:

- % Veškerý text bude od prvního výskytu tohoto znaku až do konce řádky ignorován. Samotný znak se % je interpretován jako mezera. Slouží k psaní komentářů. Zvláštní funkci mají řádky začínající dvěma procenty %%, které označují speciální typ komentářů — DSC (Document Structure Coments) sloužící zejména výstupním zařízením k orientaci v dokumentu. Někdy má znak % speciální funkci, zejména při označování standardního vstupu a výstupu (%stdin) a (%stdout), či při přístupu ke speciálním souborů.
- / Pokud znak / předchází jménu vykonavatelného objektu, zneplatní jeho vykonavatelnost a objekt se nadále chová jako nevykonavatelný. Někdy takovému objektu složenému se znaku / a jména budeme říkat objekt typu `klíč`. Samotný znak `/` není součástí jména, vypovídá pouze o nevykonavatelnosti jména. Pokud je jméno uvozeno dvojitým lomítkem `//`, je interpretováno jako okamžitě vyhodnocované jméno (immedialy evalued name) a je bezprostředně nahrazeno příslušnou hodnotou. Nejedná se o vykonání, ale o pouhou substituci. Podrobnosti viz kapitola .
- { } Pomocí složených závorek vytváříme složený proveditelný objekt. Posloupnosti mezi závorkami říkáme také procedura. Blíže viz. kapitola .
- [ ] Hranaté závorky slouží k tvorbě složených neproveditelných objektů typu pole. K složkám takto vytvořeného objektu můžeme následně přistupovat pomocí speciálních operátorů. Dále se jimi budeme zabývat v kapitole
- ( ) Kulaté závorky vymezují složené nevykonavatelné objekty typu řetězec (string). Objekty typu řetězec většinou budou reprezentovat text určený k tisku.
- <> Jednoduché špičaté závorky jsou vyhrazeny pro tvorbu speciálních řetězců.
- <<>> Tyto závorky jsou určeny pro tvorbu anonymních uživatelských slovníků. Vrátíme se k ním opět v kapitole .
- # Znak používaný k zápisu čísel v různých číselných soustavách.

PostScript je jazyk postfixový, tj. při zadávání výrazů píšeme nejprve operandy a potom operátory. PostScript je navíc jazyk zásobníkový, operandy se tedy ukládají na zásobník, odkud si je operátory odebírají. Každý operátor ví, kolik operandů má odebrat, a výsledek své operace ukládá opět na zásobník. Zásobník je datová struktura funkčně zcela poplatná svému názvu. Výstižná je i anglická zkratka LIFO (Last In First Out), kdo tam vleze jako poslední, vyleze jako první. Přicházející data jsou ukládána na sebe a přístup je umožněn pouze k poslednímu přírůstku. Pokud chceme přistoupit k dřívějším datům, musíme nejprve odstranit vše, co leží nad nimi. Vysvětleme si to na příkladu. (Na tomto místě rozebereme celou věc trochu podrobněji, v dalších případech většinou práci vstupního modulu — úst — přeskočíme.)

```
1 2 add neg
```

V první fázi čte interpret postupně znaky 1 a 2 a konvertuje je na objekty typu číslo, dále identifikuje jména `add` a `neg` a konvertuje je na vykonavatelné objekty — operátory ve významu binárního sčítání a změny znaménka. V druhé fázi uloží číslo 1 a vzápětí 2 na zásobník, pak binární operátor `add` sečte obě čísla, výsledek číslo 3 uloží na zásobník, odkud si jej vezme až unární operátor `neg` (změna znaménka) a výsledek -3 uloží na zásobník. Pro lepší pochopení si to zakresleme:

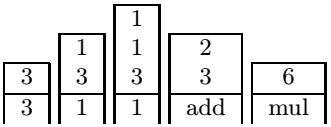


Pod čarou jsou zapsány jednotlivé postupně interpretované objekty, nad čarou jsou zaznamenány změny zásobníku. Nové objekty jsou v tomto zobrazení přidávány zdola.

Takovýto nezvyklý způsob zpracování se může zdát podivný, má však i jisté výhody. Například při zadávání výrazu se zcela obejdeme bez závorek. Výraz $3 \times (1 + 1)$ zapíšeme tedy jako:

```
3 1 1 add mul
```

což se interpretuje jako:



Z příkladů je vidět, že pro programátora je důležité znát aritu každého operátoru a procedury, tj. počet jejich operandů. Pro další zpracování výsledků operace je též nutno vědět, kolik je výsledků a v jakém pořadí se budou zapisovat na zásobník.

PostScript má zásobníků hned několik. Nastiňme si nyní nahrubo jejich význam. (Později se k nim vrátíme.)

zásobník operandů (operand stack) S tímto zásobníkem pracuje většina operátorů, jako jsou matematické operace či operace s řetězci. Operandy, tj. čísla, řetězce a logické hodnoty se ukládají do tohoto zásobníku, odkud si je operátory odebírají. Pro speciální zásobníkové operace slouží sada zásobníkových operátorů umožňující např. měnit pořadí operandů na zásobníku, mazat zbytečné operandy atd. Podrobněji se s ním seznámíme ještě v této kapitole.

zásobník spustitelných objektů (execution stack) Pokud se vykonávání některého vykonavatelného objektu přeruší spuštěním jiné vnořené procedury, je tento objekt uložen na zásobník spustitelných objektů a interpret se k němu vrátí až po dokončení výkonu vnořeného objektu. Tento zásobník je zhusta využíván např. při definicích nových operátorů. Někdy se ovšem spustitelné objekty mohou uložit i na zásobník operandů. V tomto případě je ovšem musíme na chvíli zbavit statusu spustitelnosti. K zásobníku spustitelných objektů má PostScriptový interpret přístup pouze pro čtení. Můžeme si jej tedy např. vypsat, ale neexistují operátory na jeho modifikaci.

zásobník grafických stavů (graphics state stack) PostScript je především jazyk na popis stránky. Proto valná většina jeho operátorů slouží k vykreslování grafických objektů v různých barvách a výplních. Součástí grafického stavu jsou všechna nastavení transformačních matic, nastavení vyplňovacích barev a vzorů, tvary čar, aktuální pozice kreslicího bodu, aktuální kreslená cesta. Jejich okamžitá nastavení lze uložit do zásobníku grafických stavů a později se k nim opět vrátit. Více se grafickým stavům budeme věnovat v kapitole .

zásobník slovníků (dictionary stack) PostScript dovoluje i vlastní definice operátorů. Jednotlivé definice patřící logicky k sobě může uživatel sdružovat do slovníků. Pokud chceme použít operátor definovaný v rámci nějakého slovníku, musíme tento slovník aktivovat. Aktivované slovníky se hromadí na zásobníku slovníků. Potřebná definice se vyhledává v aktivních slovnících v tomto zásobníku shora dolů (tj. nejpozději aktivované slovníky se prohledávají jako první). Podrobnosti najdete v kapitole .

Pokud budeme dále hovořit o zásobníku, bude to zpravidla zásobník operandů. Bude-li řeč o jiném zásobníku, vždy to explicitně zdůrazníme.

Nyní už víme, že programy v jazyku PostScript jsou posloupnosti vykonavatelných a nevykonavatelných objektů. Vykonavatelné objekty jsou např. matematické operátory, funkce na práci se zásobníkem, rozhodovací příkazy, příkazy cyklu, či případně, jak uvidíme později, námi definované operátory. Nevykonavatelné objekty jsou pak jejich data (operandy). Data v PostScriptu můžeme

rozdělit na nevykonavatelné objekty jednoduché a složené. V prvním příkladu jsme se setkali s nevykonavatelným objektem typu číslo. Nyní přesně definujeme některé další nevykonavatelné objekty. Půjde jednak o jednoduché objekty typu: číslo, logická hodnota, značka, jednak o složený objekt typu řetězec.

číslo (number): reálné, či celé číslo se znaménkem nebo bez něho, např.: 12, -30.5, $-12.3\text{E}10$. Celá čísla mohou být zapsána též ve tvaru: základ#číslo, tedy např. 2#1001 označuje číslo 9 v binárním tvaru. Celá čísla jsou povolena v rozsahu $[-2^{32}, 2^{32}]$, reálná čísla v rozsahu $[-10^{38}, 10^{38}]$.

řetězec (string): jakýkoliv znakový řetězec sestávající z posloupnosti písmen, číslic a symbolů. Je omezen délkou 65 535. V programu je vymezen závorkami ().

logická hodnota (boolean): nabývá hodnot `true` (pravda) a `false` (nepravda).

značka (mark): speciální neproveditelný objekt sloužící k označení pozice v zásobníku operandů.

Vidíme tedy, že 12.5E10 se interpretuje jako číslo, kdežto 12.5F10 jako jméno vykonavatelného objektu, protože nevyhovuje syntaxi nevykonavatelných objektů.

Pokud interpret PostScriptu narazí na nevykonavatelný objekt, zpravidla ho uloží na zásobník operandů. U vykonavatelného objektu nejprve zjistí, co má vlastně vykonat (dle aktuální definice), a provede jej.

Proberme si nyní některé vykonavatelné objekty — operátory. Již známe objekt `add`, který vezme dvě čísla z vrcholu zásobníku, sečte je (v pořadí v jakém byla zásobník uložena) a výsledek uloží na zásobník. V dalším použijeme následující konvenci při zápisu syntaxe operátorů:

```
x1 ... xn oper y1 y2 ... yn
```

kde `x1 ... xn` je seznam operandů vyžadovaný operací `oper`, `oper` je jméno operace a `y1 ... yn` je seznam výsledků operace v pořadí, ve kterém se ukládají na zásobník. Pokud budeme chtít upozornit na to, že operand vyžaduje nějaký typ dat, provedeme to náhradou písmena `x` nebo `y` za vhodnou sekvenci. Operaci `add` bychom tedy zapsali takto:

```
num1 num2 add y1
```

Kde zkratkou `num` naznačujeme, že na zásobníku musí být číslo (real nebo integer)

Všimněme si, že tento zápis je ve shodě s činností interpretu PostScriptu. Čísla `num1 num2` se v případě provádění uloží na zásobník, odkud si je operátor `add` vyzvedne.

Matematické operace Sekvencí `num` budeme naznačovat, že se jedná o číslo, sekvence `int` je vyhrazena pro čísla typu integer. Znaménkem `--` vyjádříme, že operátor nevyžaduje žádný vstup, či naopak neprodukuje žádný výstup.

`num1 num2` **`add`** `y1` $\qquad y1 = num1 + num2$

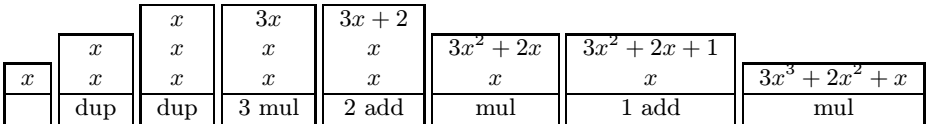
`num1`	`num2`	**sub**	`y1`	$y1 = num1 - num2$
`num1`	`num2`	**mul**	`y1`	$y1 = num1 \times num2$
`num1`	`num2`	**div**	`y1`	$y1 = num1/num2$
`num1`		**abs**	`y1`	$y1 = \|num1\|$
`num1`		**neg**	`y1`	$y1 = -num1$
`num1`		**ceiling**	`int1`	$int1$ je nejbližší vyšší celé číslo k $num1$.
`num1`		**floor**	`int1`	$int1$ je nejbližší nižší celé číslo k $num1$.
`num1`		**round**	`int1`	$int1$ je nejbližší celé číslo k $num1$
`num1`		**truncate**	`int1`	$int1$ je $num1$ po zbavení všech desetinných míst. Všimněte si rozdílného chování této funkce od **floor** na záporných číslech.
`int1`	`int2`	**idiv**	`int3`	celočíselné dělení
`int1`	`int2`	**mod**	`int3`	zbytek po celočíselném dělení
`num1`		**sqrt**	`y1`	$y1 = \sqrt{num1}$
`num1`		**log**	`y1`	$y1 = \log num1$
`num1`		**ln**	`y1`	$y1 = \ln num1$
`num1`	`num2`	**exp**	`y1`	$y1 = num1^{num2}$
`num1`		**sin**	`y1`	$y1 = \sin num1$
`num1`		**cos**	`y1`	$y1 = \cos num1$
`num1`	`num2`	**atan**	`y1`	$y1 = \arctan(num1/num2)$
`int1`		**srand**	`--`	nastaví zárodek (seed) generátoru pseudonáhodných čísel
`--`		**rrand**	`int1`	vrátí zárodek (seed) generátoru pseudonáhodných čísel
`--`		**rand**	`int1`	vrátí pseudonáhodné číslo

Za povšimnutí stojí operátor `atan`, který má dva parametry. Jsou to ony pověstné přepony — protilehlá a přilehlá. Zmíněná funkce vrátí velikost úhlu svíraného pravoúhlým trojúhelníkem s přeponami `num1 num2`.

Nyní se můžeme pustit do programování mnoha matematických výrazů. Záhy však zjistíme, že by se nám líbila možnost měnit i pořadí čísel na zásobníku, například prohodit dva horní záznamy, zduplikovat je atd. Představme si např., že na vrcholu zásobníku se nachází číslo x a chceme vyčíslit polynom $3x^3+2x^2+x$. Kdybychom uměli duplikovat vrchol zásobníku, třeba operátorem `dup`, pak by bylo možné využít Hornerova schématu, tedy faktu, že:

$$3x^3 + 2x^2 + x = ((3x + 2)x + 1)x$$

Přepsáno do PostScriptové notace:
`dup dup 3 mul 2 add mul 1 add mul`
Pro lepší porozumění si nakresleme schéma přesunů na zásobníku:



Z příkladu je vidět užitečnost takovýchto operátorů. Jejich služeb budeme využívat velmi často pro manipulaci s daty za učelem dalšího výpočtu.

Operace se zásobníkem Znakem # označujeme dno zásobníku. Pomocí `mark` označujeme neproveditelné objekty typu značka. Všechny operace se týkají zásobníku operandů.

`x1`	**pop**	`--`	odstraní nejvrchnější prvek zásobníku.
`x1 x2`	**exch**	`x2 x1`	prohodí dva nejvrchnější prvky na zásobníku
`x1`	**dup**	`x1 x1`	duplikuje nejvrchnější prvek na zásobníku
`x1...xn n`	**copy**	`x1...xn x1...xn`	duplikuje n horních prvků na zásobníku
`xn...x0 n`	**index**	`xn...x0 xn`	zkopíruje n-tý prvek shora na vrchol zásobníku
`xn-1...x0 n j`	**roll**	`yn-1...y0`	rotuje horních n prvků j-krát. Směr rotace je shora dolů.
`# x1...xn`	**clear**	`#`	vymaže obsah celého zásobníku
`# x1...xn`	**count**	`# x1...xn n`	spočítá počet prvků v zásobníku a uloží jej na vrchol zásobníku.
`--`	**mark**	`mark`	vloží na vrchol zásobníku pomocnou značku.
`mark x1...xn`	**cleartomark**	`--`	vymaže zásobník až k místu, kde je pomocná značka.

`mark x1...xn`	**counttomark**	`mark x1...xn n`	spočítá počet prvků mezi pomocnou značkou a vrcholem zásobníku.

Zastavme se na chvíli u operátoru `mark`. Ten vloží do zásobníku pomocnou značku. Pokud bychom udělali třeba:

```
1 mark add
```

moc tím interpret PostScriptu nepotěšíme, neboť značka není číslo, nejde tudíž sčítat. Interpret se nám za to odmění hláškou:

```
Error: /typecheck in add
Operand stack:
   1      --nostringval--
```

Proto je třeba u použití značek dávat pozor a případné překážející značky odstaňovat např. operátorem `pop` či operátorem `cleartomark`. Při vlastním programování je dobré snažit se o balancovanost příkazů značky kladoucích s příkazy značky odebírajícími. Objekt typu `mark` nelze vytvořit jinak než operací `mark`.

Důležité pro ladění jsou dva operátory, kterými zjišťujeme stav zásobníku. Jejich výstup není na zásobník, ale do komunikačního kanálu. Proto jim také říkáme interaktivní operátory. Operátor `==` odstraní vrchol zásobníku a vypíše jej (např. na obrazovku) operátor `pstack` vypíše obsah celého zásobníku a navíc (narozdíl od `==`) nechá zásobník v původním stavu.

Vyzkoušejme nyní získané znalosti. Zkusme zjistit, jak se chová operátor `cleartomark` pokud je na zásobníku více značek:

```
1 mark 2 3 mark 4 5 cleartomark pstack
```

Výsledek nás jistě nepřekvapí, zásobník se maže k první nalezené značce (od vrcholu zásobníku), interpret vypíše do komunikačního kanálu:

```
3
2
-mark-
1
```

Shrnutí

Program v PostScriptu je posloupnost znaků. Ta je vstupním procesorem konvertována v posloupnost jmen a speciálních symbolů. Jméno je dále nedělitelný PostScriptový objekt s přiřazeným atributem vykonavatelný/nevykonavatelný. Vykonavatelná jména jsou dále interpretována buď jako operátor, tj. vestavěná PostScriptová funkce, nebo jako uživatelská procedura. Zatím jsme se seznámili s matematickými operátory a s operátory pro práci se zásobníkem.

Speciální symboly slouží k tvorbě komentářů (%) a složených objektů (závorky) a k potlačení atributu vykonavatelnosti (/)

Jednoduché nevykonavatelné objekty jsou např. čísla, logické hodnoty a značky,

příkladem složeného objektu je řetězec. Nevykonavatelné objekty ukládá interpret na zásobník operandů, odkud si je operátory odebírají. Pokud je výsledkem operátoru nevykonavatelný objekt, je samozřejmě uložen na vrchol zásobníku.

Kromě zásobníku operandů existují ještě tři další: zásobník vykonavatelných operací, zásobník grafických stavů a zásobník slovníků, s jejichž funkcí se seznámíme v dalších kapitolách.

K zjištění stavu zásobníku slouží dva operátory: operátor `==` odstraní vrchol zásobníku a vypíše jej do komunikačního kanálu (tj. většinou na obrazovku), operátor `pstack` nechá zásobník tak, jak je, pouze vypíše celý jeho obsah do komunikačního kanálu.

Konstanty, funkce, definice

Prozatím jsme se seznámili s PostScriptovými operátory, tj. s proveditelnými objekty pevně zabudovanými v interpretu. Víme, že na ně interpret převádí některá jména s atributem proveditelnosti. Nyní bychom rádi naučili interpret novým kouskům, tj. aby přiřazoval námi zvolená jména s jiným objektům. Pojem přiřazení vlastně znamená záměnu jména za přiřazený objekt, jednoduchý, či složený. Přiřazení budeme někdy nazývat též expanzí.

PostScript umožňuje definici vlastních objektů několika způsoby. Prozatím si předvedeme dva nejjednodušší.

```
/pi 3.1415926 def
```

definuje novou konstantu `pi`. Pokud dále napíšeme třeba

```
1 pi add
```

`pi` se nám expanduje na 3.1415926 a výsledkem bude 4.1415926 uložené na zásobníku. Je třeba si uvědomit, že vlastní expanze probíhá už v okamžiku definice. Pokud bychom dále definovali např:

```
/ppi pi pi mul def
/pi 1 def
```

pak by hodnota `ppi` byla stále 9.8696041, byť by se `pi` mezitím změnilo. Tuto vlastnost můžeme potlačit takovouto konstrukcí:

```
/ppi { pi pi mul } def
```

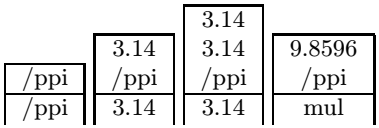
pak se hodnota `ppi` bude měnit v závislosti na hodnotě `pi`, protože expanze se bude provádět až v okamžiku použití[4]. První způsob je vhodný pro definici konstant, druhým způsobem můžeme definovat funkce. V druhém případě se nám jméno `ppi` přiřazuje složenému proveditelnému objektu — proceduře, zatímco v prvém případě se nám stejné jméno přiřazovalo jednoduchému neproveditelnému objektu typu číslo.

Všimněme si ještě jedné charakteristiky PostScriptového způsobu definic. Opravdu se jedná spíše o makra, tedy jakési zkratky. Definujeme-li např:

[4] Uživatelé TEXu tento jev jistě znají.

```
/trikrat { 3 mul } def
```
pak můžeme směle napsat sekvenci typu:
```
1 trikrat trikrat
```
jejímž výsledkem bude $1 \times 3 \times 3 = 9$. Takto definovaný operátor je takříkajíc zleva nenasycený, vlastní `3 mul` by skončilo s chybou; protože však k expanzi dojde až v okamžiku použití, kdy zajistíme potřebný počet operandů na zásobníku, je všechno v pořádku.

Podívejme se trochu podrobněji, co se děje při definicích.
```
/ppi 3.14 3.14 mul def
```
Lomítko před `ppi` říká, že se jedná o neproveditelný objekt, který se jako takový uloží na zásobník (bez lomítka). Poté se normálně provádí výpočet do doby, než se narazí na `def`. Stav zásobníku je tedy postupně:



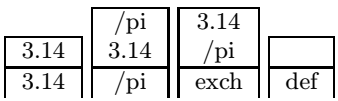
Operátor `def` má tuto syntaxi:
```
jmeno objekt def
```
kde `jmeno` je objekt typu jmeno a `objekt` je libovolný PostScriptový objekt.

Operátor vyžaduje dvě položky na zásobníku a provede asociaci jména a objektu v právě aktuálním slovníku. Teoreticky by bylo možné napsat i např:
```
1 2 def
```
tedy definovat jedničku jako dvojku, ale protože jednička není proveditelný objekt, zapisuje se nadále na zásobník jako jednička. O tom, že je ovšem definována jako dvojka se můžeme přesvědčit například takto:
```
1 2 def 1 load pstack
```
Po provedení těchto operací nám zůstane ná zásobníku dvojka (vysvětlete).

Narazili jsme zde na skutečný význam pojmů proveditelný a neproveditelný. Proveditelné objekty se automaticky nahrazují, zatímco neproveditelné zůstávají beze změny.

K zobrazení definice jsme zde použili nový operátor `load`. Ten vezme nejvrchnější položku zásobníku a vypíše její aktuální definici. V případě neproveditelného objektu, jako třeba jedničky, je to jednoduché; proveditelné objekty musíme stejně jako při definici uvodit lomítkem.

Jiný, praktičtější příklad je tento:
```
3.14 /pi exch def
```



Vidíme, že se `pi` definovalo jako 3.14. Tento způsob je využitelný v případě, kdy si chceme zapamatovat operandy operátoru. Chceme-li třeba navrhnout proceduru `prumery`, která má tři operandy a vrací na zásobník všechny tři aritmetické průměry, postupujeme takto:

```
/prumer{
/prvni exch def
/druhy exch def
/treti exch def
/prum {add 2 div} def
prvni druhy prum
prvni treti prum
druhy treti prum
} def
```

Jednoduché, ne? Druhý, třetí a čtvrtý řádek vytvoří objekty `prvni, druhy, treti` do kterých se „slíznou“ postupně tři čísla z vrcholu zásobníku. Na pátém řádku se deklaruje procedura `prum` počítající aritmetický průměr dvou čísel na zásobníku a vlastní výpočet se provádí na následujících třech řádcích.

Abychom ještě lépe porozuměli, co se děje při definicích, rozeberme si nyní případ se složenými závorkami. Podívejme se, co se děje se zásobníkem při definici naší funkce `trikrat`

```
/trikrat { 3 mul } def
```

	{ 3 mul}	
/trikrat	/trikrat	
/trikrat	{3mul}	def

Vidíme, že složené závorky sdruží obsah mezi sebou do jednoho složeného objektu a ten umístí na zásobník. Pak `def` provede potřebnou asociaci. Použitím už nám známých firem `load` a `pstack` se můžeme přesvědčit o úspěchu cele akce:

```
/tri load pstack
```

vypíše:

```
{3 mul}
```

Nepřímo jsme se seznámili s další vlastností operátoru `load`. Na zásobníku nám zbude nevykonavatelný objekt typu procedura. Pokud bychom ji chtěli spustit, použili bychom operátor `exec`. Tedy například sekvence:

```
2 /tri load exec
```

by umístila objekt `6` na vrchol zásobníku.

Podívejme se nyní podrobněji, co vlastně znamená operace nahrazení. V kapitole jsme si říkali, že definice se sdružují do slovníků. Slovník je další příklad složeného objektu. Můžeme si ho představit jako tabulku o dvou sloupcích, v prvním sloupečku je název a v druhém hodnota. Takové dvojici budeme někdy říkat nahrazovací pár. První objekt budeme navíc nazývat objekt typu jméno.

Napišme si takovou tabulku pro makra `pi` a `trikrat`

název	hodnota
`pi`	3.14
`trikrat`	{3 mul}

Víme také, že většinu operátorů můžeme předefinovat. Zkusme to například s operátorem `add`. Předefinujme ho na chvíli na `mul`

```
/add {mul} def
1 1 add pstack
```

výsledkem použití takto zmateně předefinovaného `add` bude očekávaná jednička. Představme si však, že jsme v situaci, kdy jsme předefinovali nějaký operátor a nyní potřebujeme zpět jeho původní definici. Ta je ovšem ztracena. Pokud jsme ale před tím, než jsme operátor předefinovávali, aktivovali nový slovník, kam se nová definice operátoru uložila, můžeme se ke starým hodnotám vrátit tak, že tento slovník deaktivujeme. Osvětlíme si to na příkladě:

```
1 1 add ==
1 dict begin
/add {mul} def
1 1 add ==
end
1 1 add ==
```

Na prvním řádku pouze zkoušíme operátor `add`, výsledek operace je 2. Druhý řádek vytvoří slovník s kapacitou jedné definice (`1 dict`) a otevře ho (`begin`). Následuje předefinování operátoru `add`, ten nyní dává výsledek 1. Po uzavření slovníku (jeho deaktivaci) se obnoví původní definice operátoru `add`.

Operace aktivace a deaktivace nového slovníku se provádí pomocí zásobníku slovníků. Pomocí `int dict` se vytvoří na zásobníku operandů nový slovník o kapacitě `int` (přesně řečeno jeho identifikátor) a operátorem `begin` se umístí na vrchol zásobníku slovníků. Protože se definice vyhledávají postupně ve všech aktivních slovnících shora dolů, jsou definice vytvořené v tomto nejvyšším slovníku preferovány.[5]

Při vytvoření nového slovníku musíme specifikovat jeho kapacitu, tj. kolik nových definic může obsáhnout. Jistě nás napadne otázka, co se stane, když vytvoříme definic více, než-li je kapacita slovníku. To se liší podle verze PostScriptu. V Level 1 nadbytečné definice vyvolaly chybu, zatímco v Level 2 se zvětší kapacita slovníku a o definice nepřijdeme.

Slovníky většinou používáme ještě k jinému účelu. Definujeme a shromažďujeme si v nich procedury, které budeme později používat. Definice se totiž ve slovnících nezapomínají odstraněním ze zásobníku slovníků.

[5] Slovníky vlastně zajišťují svérázný způsob definice lokálních procedur jak ji známe z jiných programovacích jazyků.

Využijeme dosud nabytých znalostí ke konstrukci slovníku pro počítání s komplexními čísly. Komplexní číslo budeme reprezentovat jako dvojici čísel, nejprve bude reálná, pak komplexní část. Zkusme tedy tuto definici:

```
/complex 5 dict def
complex begin
/cti {/iy exch def /y exch def /ix exch def /x exch def} def
/add {cti x y add ix iy add} def
/sub {cti x y sub ix iy sub} def
/mul {x y mul ix iy mul sub x iy mul ix y mul add} def
/neg {neg exch neg exch} def
end
```

Zavedli jsme slovník `complex`, ve kterém předefinováváme většinu operátorů určených k počítání. Nyní pokaždé, když budeme chtít použít počítání s komplexními čísly aktivujeme slovník `complex`.

```
complex begin
1 1 2 2 add
end
```

Bohužel tento příklad skončí s chybou. Háček je v tom, že v definici `add` používáme `add`, které se interpret snaží rozexpandovat podle aktuální definice a tím dojde k zacyklení. Proto musíme celou definici pozměnit takto:

```
/complex 5 dict def
complex begin
/cti {/iy exch def /y exch def /ix exch def /x exch def} bind def
/add {cti x y add ix iy add} bind def
/sub {cti x y sub ix iy sub} bind def
/mul {x y mul ix iy mul sub x iy mul ix y mul add} bind def
/neg {neg exch neg exch} bind def
end
```

Operátor `bind` zajistí, že se v předcházející proceduře (uvozené složenými závorkami) nahradí jména operátorů skutečnými operátory. Jde o jakýsi lokální zákaz expanze. Jak je taková věc realizována ve skutečnosti, záleží na daném interpretu. Můžeme si například představit, že jména jsou nahrazena ukazateli na funkce. Operátor `bind` použijeme všude tam, kde chceme zajistit, aby námi definované procedury byly stabilní vůči redefinicím operátorů, z nichž jsou složeny. Bohužel funkce operátoru `bind` je omezená. Funguje totiž pouze na jména vestavěných jménem operátorů. Kdybychom např. chtěli znova předefinovat `add` a využit k tomu náš komplexní `add`, operátor `bind` by nám v tomto případě příliš nepomohl, protože `add` v případě aktivního slovníku `complex` není operátor, ale procedura. Příčiny takového chování tkví v tom, že operátor je jednoduchý

objekt a `bind` vlastně nahradí jméno operátoru ukazatelem[6] na operátor, což je opravdu v případě jednoduchých objektů absolutní řešení. Pokud bychom však ukazovali na složený objekt, bylo by nutné, zafixovat všechny složky objektu, aby `bind` dělalo opravdu to co chceme. K tomu je nutno vytvořit kopii původního objektu odlišnou od svého vzoru. Protože však kopírování složených objektů pracuje opět pouze přes ukazatele, není tato operace možná, blíže viz kapitolu

Jiná možnost, jak řídit expanzi objektů, je využití dvojitého lomítka. V kapitole jsme řekli, že pokud je jméno uvozeno dvojitým lomítkem, je okamžitě expandováno, tedy vstupní modul předává k dalšímu zpracování už patřičnou asociaci. Použijeme-li tuto vlastnost v době definice procedury, provede se nám okamžitá expanze o jeden stupeň. Tedy jméno se expanduje na svojí aktuální náhradu ihned a ne až v okamžiku použití. Pokud se jméno nahradí posloupností jiných jmen, na tato jména se již okamžitá expanze nevztahuje. Důležité v tomto případě je, že na rozdíl od operátoru `bind` funguje tato substituce i na jiné objekty, například na slovníky. Představme si situaci, že chceme definovat objekt `ckvadrat` (komplexní kvadrát), který je závislý na slovníku `complex`. Nedokážeme však zajistit, aby se definice ve slovníku během interpretace neměnily. Proto použijeme dvojitého lomítka.

```
/ckvadrat {//complex begin
cti x y x y mul end} def
```

Co se stalo: Objekt (jméno) `complex` se nahradilo již v okamžiku definice procedury `ckvadrat` aktuální hodnotou. Procedura je tedy o něco větší, nese si v sobě celou definici slovníku `complex`, ale zato není závislá na redefinicích jednotlivých objektů ve slovníku.

Na tomto místě bude vhodné si rozebrat práci interpretu se zásobníkem spustitelných objektů. Připomeňme si, že k tomu dochází v „trávicím“ nahrazovacím modulu. Vezměme si jednoduchý příklad:

```
complex begin
1 2 1 2 add
end
```

Nejprve je jméno `complex` rozpoznáno jako proveditelný objekt a nahrazovací modul jej převede na neproveditelný objekt jméno slovníku. Prováděcí modul toto jméno uloží na zásobník operandů. Následující `begin` je nejprve rozpoznáno jako jméno a nahrazovací modul jej nahradí PostScriptovým operátorem, který ze zásobníku operandu odebírá jméno slovníku, a ten umístí na vrchol zásobníku slovníků. Od této chvíle se všechny definice budou hledat nejprve v tomto slovníku a následně ve všech slovnících pod ním v sestupném pořadí dle jejich umístění na zásobníku. Stejně tak se definice nových procedur budou ukládat do tohoto slovníku, nebude-li řečeno jinak.

[6] Byť zde mluvíme o ukazatelích, vězte, že PostScript žádný mechanismus pro práci s ukazateli neposkytuje.

Čísla 1 2 1 2 projdou trávicím traktem vcelku bez potíží. Jsou rozpoznána jako neproveditelné objekty, nahrazovací modul je tedy předává okamžitě prováděcímu modulu, a ten je umístí na zásobník.

Následuje proveditelný objekt `add`. Interpret najde jeho definici v aktivním slovníku `complex`. Ta vypadá takto:

`cti x y` [`add`] `ix iy` [`add`]

pomocí rámečků jsme označili, že další `add` již prošly péčí operátoru `bind` a že se tudíž nejedná o jména, ale přímo o odkazy na operátory. Během generování tohoto řádku ale trávicí trakt zjistí, že objekt `cti` je možno dále expandovat. Uloží si tedy zbytek na zásobník proveditelných objektů a expanduje jméno `cti`. To je opět obsaženo ve slovníku `complex`:

`/iy` [`exch`] [`def`] `/y` [`exch`] [`def`] `/ix` [`exch`] [`def`] `/x` [`exch`] [`def`]

vidíme, že jedná o posloupnost neproveditelných objektů nebo operátorů, a proto jsou předány vykonávacímu modulu, který provede potřebné definice a zaznamená je do aktivního slovníku `complex`. Po provedení tohoto kroku jsou nové definice tyto:

`iy`	2
`y`	1
`ix`	2
`x`	1

Tím je dokončena expanze objektu `cti` a je možno se vrátit k zásobníku proveditelných objektů. Jak víme, nachází se tam:

`x y` [`add`] `ix iy` [`add`]

jména `x y` jsou nahrazena neproveditelnými objekty, čísly 1, 1 a prováděcí modul je umístí na zásobník. Pak se provede operátor `add`, obě čísla se tedy sečtou a stejný postup se aplikuje na objekty `ix iy add`. Na zásobníku operandů zbudou neproveditelné objekty 3 3. Zásobník proveditelných objektů je prázdný, může se tedy přistoupit ke zpracování proveditelného objektu `end`. Ten se expanduje na operátor, jenž prováděcímu modulu přikáže odstranit slovník `complex` ze zásobníku aktivních slovníků.

Na tomto poněkud komplikovaném případě jsme si ukázali celou škálu situací, s kterými se můžeme během interpretace setkat. Za povšimnutí stojí např fakt, že jménům `x y ix iy` je přidán správný význam až při expanzi jména `cti`, skutečnou definici tedy objekty obdrží až za běhu. Vidíme, že patřičných ladicích nástrojů by bylo programování komplikovaných procedur těžkou záležitostí. PostScript poskytuje dvojici operátorů pro práci se zásobníkem proveditelných objektů. Operátor `countexecstack` vrací počet objektů na zásobníku proveditelných objektů, zatímco operátor `execstack` vezme pole z vrcholu zásobníku operandů a umístí do něj prvky ze zásobníku proveditelných objektů. Velikost tohoto pole by měla samozřejmě odpovídat počtu prvků na zásobníku prove-

ditelných objektů, což většinou zajistíme právě operátorem `countexecstack`. Nejjednodušší použití je následující:

```
countexecstack array execstack
```

Zastavme se na závěr u operátoru `exec`. Zkusme si jednoduchý příklad. Co zbude na zásobníku po provedení sekvence:

```
1 1 /add exec
```

Odpověď je možná překvapivá, ale výsledkem bude opět:

```
/add
1
1
```

operátor `exec` totiž respektuje příznak spustitelnosti. Pokud chceme dosáhnout vykonání objektu `/add`, musíme jej nejprve konvertovat na spustitelný objekt. K tomu nám dopomůže operátor `cvx`. Výsledkem posloupnosti:

```
1 1 /add cvx exec
```

již bude očekávaná dvojka.

Shrnutí

K definici nových, nebo redefinici starých objektů slouží operátor `def`. Očekává dva objekty na zásobníku, první je zpravidla jméno nového resp. starého objektu zbavené pomocí lomítka atributu spustitelnosti, druhý je nejčastěji složený objekt typu procedura. Tato definice se vloží, není-li řečeno jinak, do slovníku z vrcholu zásobníku aktivních slovníků.

Při expanzi jmen se užívá zásobník spustitelných objektů, kam se ukládá nedoexpandovaný zbytek definice. Nejčastěji k tomu dochází, když v definici objektu figurují další objekty, které je třeba expandovat.

K řízené expanzi slouží jednak operátor `bind`, který fixuje význam jména, jedná-li se o jméno vestavěného operátoru, jednak dvojité lomítko `//`, vynucující okamžitou náhradu objektu i případě, kdy je nahrazovací mechanismus potlačen (např. mezi složenými závorkami).

Pro sdružování definic slouží slovníky. Nový slovník vytvoříme operátorem `dict`. K jeho aktivací slouží operátor `begin`, k jeho deaktivaci operátor `end`. Aktivací se slovník uloží na vrchol zásobníku aktuálních slovníků, deaktivací, ze z vrcholu tohoto zásobníku odebere. Po deaktivaci slovníku se definované náhrady neztrácí, pouze nejsou přístupné.

K případnému zkoumání obsahu zásobníku spustitelných objektů slouží dvojice operátorů `countexecstack` a `execstack`. První vrací hloubku tohoto zásobníku, druhý vkládá do vhodného pole na vrcholu zásobníku operandů obsah zásobníku spustitelných objektů.

V PostScriptu lze předefinovat každý objekt, následky si nese ovšem každý sám.

Cykly, smyčky, podmínky

Zatím jsme se nezmiňovali o řídících konstrukcích které známe jako nezbytnou součást jiných programovacích jazyků. Řídící konstrukce jsou v PostScriptu realizovány pomocí několika operátorů. Nejjednodušší z nich je operátor `repeat`. Jeho syntaxe je následující:

```
n { příkazy } repeat
```

Vidíme, že operátor vyžaduje na zásobníku dva objekty, první je typu celé číslo, druhý je typu procedura. Číslo udává počet provedení procedury. Například součet řady čísel od jedné do pěti bychom realizovali pomocí operátoru `repeat` takto:

```
1 2 3 4 5 4 {add} repeat
```

Důležitým rysem operátoru `repeat` je, že, narozdíl od následujícího operátoru nenechává na zásobníku operátorů žádné pomocné údaje, jak se o tom můžeme přesvědčit pomocí `pstack`.

```
4 { pstack } repeat
```

Zcela jinak se z tohoto hlediska chová příkaz for-cyklu. Jeho syntaxe je:

```
x y z~proc for
```

Na začátku se nastaví čítač na hodnotu `x`, při každém průběhu se zvyšuje o `y`, dokud nedosáhne hodnoty `z`. Aktuální hodnota čítače se pokaždé uloží na zásobník operandů a spustí se procedura `proc`. Např. součet řady $\sum_{i=1}^{10} i^2$ můžeme tedy naprogramovat jako:

```
0 1 1 10 { 2 exp add } for
```

Při při použití `for` je potřeba myslet i na zásobník. Např. po provedení

```
1 1 10 {} for
```

nám na zásobníku „zbudou“ hodnoty 10 9 8 7 6 5 4 3 2 1.

Z předvedených příkladů je zřejmé, že operátor `repeat` použijeme v případě kdy chceme pouze opakovat jistý úkon, a operátor `for` v případě, když nás zajímá řídicí proměnná. Operátor `repeat` navíc vyžaduje jako počet opakování celé číslo, kdežto u `for` mohou být všechna čísla reálná. Pak ovšem musíme myslet i na zaokrouhlovací chyby, které mohou narušit průběh výpočtu.

V PostScript existuje ještě jeden důležitý příkaz cyklu `foreach`, který slouží k procházení složených objektů. Více si o něm povíme v kapitole .

Dalšími důležitými operátory jsou příkazy pro větvení výpočtu `if` a `ifelse`.

```
bool proc if
bool proc1 proc2 ifelse
```

kde `bool` je pravdivostní hodnota a `proc, proc1, proc2` jsou procedury. Pokud je `bool` rovno `true`, vykoná se v případě operátoru `if` procedura `proc`. V případě operátoru `ifelse` se vykoná procedura `proc1`. Pokud je `bool` rovno `false`, operátor `false` neudělá nic a operátor `ifelse` vykoná proceduru `proc2`.

Jak již jistě tušíte, PostScript oplývá řadou operátorů, které jako výstup produkují zmíněné logické hodnoty. Seznámíme se nyní s těmi, které realizují porovnávání objektů.

První z nich je operátor `eq`. Porovnává dva nejvrchnější objekty na zásobníku operandů a vrací hodnotu `true`, pokud jsou shodné, a hodnotu `false`, pokud nejsou. Pojem shodnosti se ovšem v PostScriptu liší objekt od objektu. U čísel se porovnává skutečná hodnota, což znamená, že např:

```
1.00 1 eq
```

vrátí hodnotu `true`, i když první číslo je reálné a druhé celé. Složené objekty se chovají vůči operátoru `eq` mnohem záludněji. Řetězce se porovnávají znak po znaku a shodují-li se všechny, je výsledná hodnota `true`, v opačném případě `false`. Ostatní složené objekty jsou shodné pouze pokud vznikly z jednoho složeného objektu operátory `dup` či `copy`. Ukažme si to na příkladech:

```
1.0 1 eq           true
(abc) (abc) eq     true
[1 2 3] dup eq     true
[1 2 3] [1 2 3] eq false
```

Operátor `eq` má navíc ještě jednu vlastnost. Řetězce a jména, pokud se shodují obsahově, považuje též za shodné:

```
(abc) /abc eq true
```

Negací operátoru `eq`, operátor `ne`. Vrací `true` v případě, kdy `eq` vracel `false` a obráceně.

Trochu jinak se chovají operátory pro porovnání velikostí. Jedná se o operátory `gt` ($>$), `ge` ($\geq$), `lt` ($<$) a `le` ($\leq$). Ty fungují pouze v případě, kdy na vrcholu zásobníku je dvojice čísel, nebo dvojice textových řetězců. V prvním případě provedou zmíněné operátory číselné porovnání a jeho výsledek uloží na zásobník, v druhém případě provedou porovnání lexikální. Pokud se na vrcholu zásobníku nachází jiné typy objektu, případně objekty dvou různých typů, ohlásí interpret PostScriptu chybu **typecheck**.

Pro kombinaci různých podmínek slouží logické operátory `and`, `not` `or` `xor`. Ty mohou pracovat jednak na logických hodnotách, jednak na celých číslech. V případě, že je vstupem celé číslo, pracují po bitech (nula má význam `false`, jednička `true`). Místo zdlouhavého vysvětlování snad lépe poslouží následující tabulka:

```
true false and  false
1    1     and  1
1          not  -2
```

Vidíme, že celá čísla jdou výhodně použít k reprezentací více nezávislých podmínek. K výstavbě takovýchto multipodmínek musíme být ovšem schopni pohybovat jednotlivými bity. To nám umožní operátor `bitshift`. Ten očekává na zásobníku dvě celá čísla — operand a posun:

```
int posun bitshift
```

Pokud je `posun` kladný, provede operátor bitový posun doleva o specifikovaný počet bitů. Vlevo vytlačené bity jsou ztraceny, zprava se bitová reprezentace čísla

`int` doplňuje nulami. Při záporném posunu se provede obdobný bitový posun směrem doprava.

Shrnutí
PostScript poskytuje standardní sadu řídících struktur. Pro realizaci cyklů slouží operátory `repeat`, `for`, `foreach`, pro podmíněné příkazy `if`, `ifelse`. PostScript disponuje operátory porovnání objektů jako `eq`, `ge`, `gt`, `ge`, `gt`. Z jednoduchých podmínek lze pomocí logických operátorů `and`, `not`, `or`, `xor` vytvořit podmínky složené. Zmíněné operátory jsou většinou polymorfní a lze je použít s různými výsledky na více typů objektů.

Složené objekty

V předchozím výkladu jsme se již s některými složenými objekty seznámili. Víme, že se tvoří pomocí závorek všech druhů a že na zásobník se ukládají vcelku. Nám známé složené objekty jsou řetězec, slovník a procedura. První dva jsou neproveditelné, třetí je proveditelný. Zabývejme se nyní složenými objekty trochu hlouběji.

Složené objekty mohou být v PostScriptu dvojího druhu. Anonymní a neanonymní. Neanonymní se vytváří vždy některým z operátorů `array`, `packedarray`, `string`, `dict`. Ty jednak vytvoří ve virtuální paměti daný objekt a navíc na zásobník umístí tzv. ID tohoto objektu (můžeme si ho představit jako odkaz), kterým se můžeme později na něj odkázat, i když jej ze zásobníku odstraníme. Anonymní objekty vytváříme pomocí závorek a nejčastěji slouží v programu jako konstanty. Pokud je ze zásobníku odstraníme, nelze se k nim později vrátit.

Pole
Anonymní objekt typu pole se vytvoří pomocí hranatých závorek `[]`. Tedy například:

```
[25 1.8 (abc) {add sub}]
```

vytvoří na zásobníku jeden složený objekt typu pole obsahující postupně čtyři objekty. První jsou dva jednoduché objekty typu číslo, třetí je složený objekt typu řetězec a čtvrtý složený objekt typu procedura (přesvědčit se o tom můžeme pomocí operátoru `==`).

Pole můžeme opět rozbít na jednotlivé součástky operátorem `aload`. Ten umístí na zásobník postupně všechny složky pole a na vrchol položí opět celé pole. Tedy např. výsledkem řádek:

```
[25 1.8 (abc) {add sub}]
aload
```

```
pstack
```
bude výstup do komunikačního kanálu:
```
[25 1.8 (abc) {add sub}]
{add sub}
(abc)
1.8
25
```
Chceme-li tedy přistoupit k položkám pole pomocí operátoru `aload`, musíme většinou vrchol zásobníku smazat.

Počet prvků budeme nazývat délkou pole. Pro dané pole jeho délku zjistí operátor `length`.

Často budeme chtít vytvořit prázdné pole s určitou délkou, do kterého budeme později zapisovat. K tomu je určen operátor `array`, odebírající ze zásobníku celé číslo udávající délku vytvořeného zásobníku. Tedy například:
```
4 array
```
vytvoří na zásobníku neanonymní prázdné pole délky 4 (resp. ID tohoto pole), přičemž prázdností se míní, že pole je složeno z nulových objektů. Délku pole po jeho vytvoření již nelze změnit.

Pohodlnější přístup k prvkům pole zajišťuje dvojice operátorů `put` a `get`, z nichž `put` zapisuje objekty na dané místo v poli a `get` je čte. Syntaxe obou operátorů je stejná:
```
pole index objekt operátor(put/get)
```
Nutno míti na paměti, že pole je indexováno od nuly. Pole ze začátku kapitoly bychom tedy mohli vytvořit i takto:
```
/a 4 array def
a 0 25 put
a 1 1.8 put
a 2 (abc) put
a 3 {add sub} put
```
Případně pomocí operátoru `astore` fungující opačně nežli `aload`
```
/a 4 array def
25 1.8 (abc) {add sub} a astore
```
Proveďme nyní pokus:
```
[ 1 2
pstack
```
na výstupním kanálu se objeví:
```
2
1
-mark-
```
z čehož je patrné, že otvírací hranatá závorka je náhražkou za operátor `mark`. Například tedy konstrukce:

```
[ 1 2 mark 3 4 ]
```
vytvoří na zásobníku situaci:
```
[3 4]
2
1
```

Z experimentu vyplývá, že pomocí konstrukce se složenými závorkami jen těžko vytvoříme pole obsahující značku. Naštěstí tuto nepříjemnost lze obejít operátorem `put`.

Levou hranatou závorku můžeme tedy nahradit operátorem `mark`, pravou hranatou závorku lze naopak nahradit procedurou založenou na operátoru `astore`. Pole `[1 2 3]` můžeme vytvořit i konstrukcí:
```
mark 1 2 3 counttomark array astore exch pop
```

Posledení tvrzení není zcela pravdivé, Neboť zmiňovanou konstrukcí vznikne pole neanonymní.

Se složenými objekty se pojí také zvláštní operátor cyklu `forall`. Jeho syntaxe je pro objekt typu pole:
```
pole procedura forall
```
Tento operátor aplikuje proceduru pro každý prvek pole v pořadí, v jakém se v poli nacházejí. Například součet $\sum_{i=1}^{10} i^2$ bychom realizovali takto:
```
0 [1 2 3 4 5 6 7 8 9 10] {dup mul add} forall
```
Případně lze použít rafinovanější konstrukci:
```
0 [1 1 10 {} for ] {dup mul add} forall
```
kde se vlastní pole vygeneruje až příkazem for-cyklu.

Důležitý pro práci se složenými objekty je operátor `copy`. Již jsme se s ním setkali v kapitole , kde kopíroval vrchních n prvků zásobníku na vrchol zásobníku. V případě složených objektů funguje poněkud jinak. Pro tuto vlastnost (odlišná funkce závislá na typu objektu) nazýváme operátor `copy` polymorfní. Syntaxe operátoru je v tomto případě:
```
pole1 pole2 copy pole1
```
Délka `pole2` musí být alespoň jako délka `pole1`. Operátor kopíruje složky `pole1` do pole `pole2` počínaje nultým prvkem. Prvky v poli `pole2` nacházející se mimo rozsah definovaný délkou `pole1` zůstanou nezměněna. Například:
```
/pole 6 array def
[1 2 3 4] pole copy
```
vytvoří pole `pole` a naplní ho obsahem: `[1 2 3 4 null null]`.

U operátoru `copy` je potřeba se ještě na chvíli zastavit. Zkusme si trochu zaexperimentovat:
```
/pole 6 array def
[1 2 3 4] pole copy
[pole]
[3 4] pole copy
pstack
```

První dva řádky vytvoří nejprve pole délky 6 a následně jej naplní prvky 1234. Na dalším řádky vytvoříme pole obsahující pouze jeden prvek a to pole `pole`. Následně přepíšeme operátorem `copy` první dvě položky pole `pole`. Situace na zásobníku by měla být:

```
[3 4]
[[1 2 3 4 null null]]
[1 2 3 4]
```

operátor `pstack` však ukáže:

```
[3 4]
[[3 4 3 4 null null]]
[1 2 3 4]
```

čili změna prvků pole `pole` vyvolala i změnu ve všech polích pole `pole` obsahující. Složené objekty se tedy tvoří výhradně pomocí odkazů na objekty a změna objektu se tudíž projeví ve všech složených objektech tento objekt obsahující.

Poslední dva operátory pro práci se složenými objekty jsou `putinterval` a `getinterval`. Slouží k podobnému účelu jako operátor `copy`, pouze umožňují stanovit pozici, od které bude v poli `pole2` kopírování zahájeno. Syntaxe operátoru `putinterval` je:

```
pole2 index pole1 putinterval
```

Připomeňme si, že pole je v PostScriptu indexováno od nuly. Též je dobré si povšimnout, že operátor `putinterval` narozdíl od `copy` nezanechává na zásobníku žádné pole a pořadí polí je obrácené, tj. níže na zásobníku je pole do kterého se kopíruje.

Syntaxe operátoru `getinterval` je:

```
pole1 index délka getinterval pole2
```

Operátor vyjme z pole `pole1` podpole `pole2` o délce `délka` a začínající na pozici `index`. To uloží na zásobník.

Zhuštěná pole

PostScript zahrnuje datovou strukturu zhuštěné pole. Rozdíly vůči normálnímu poli jsou dva. První spočívá ve způsobu uložení v paměti. Ten je úsporný a zhuštěné pole zabírá méně místa. Což se ovšem projeví negativně v rychlosti přístupu k jednotlivým položkám. Výjimkou je operátor `forall` přistupující k jednotlivým prvkům postupně — rychlost tohoto operátoru je srovnatelná s normálním polem. Druhý rozdíl spočívá v tom, že do jednou vytvořeného zhuštěného pole nelze zapisovat, je určeno pouze ke čtení. To znamená, že operátory `put`, `putinterval`, `astore` nelze v souvislosti se zhuštěným polem použít.

Objekt zhuštěné pole vytvoříme pomocí operátoru `packedarray` následovně:

```
objekt1 ... objektn n packedarray pole1
```

Od této chvíle již k objektu můžeme přistupovat pouze čtením. Kromě výše uvedených tří operátorů se zhuštěné pole chová podobně jako nezhuštěné pole.

Řetězce

O tomto objektu víme, že je složen z jednotlivých znaků a většinou reprezentuje nějaký text. Později, v kapitole si ukážeme, jak řetězec vytisknout. Zatím se však pouze naučíme různým jiným operacím.

Stejně jako objekt typu pole, lze řetězce vytvářet dvěma způsoby. Konstantní řetězce pomocí závorek:

```
(retezec)
```

nebo „proměnné“ řetězce pevné délky operátorem `string`, přičemž délkou řetězce míníme počet znaků, z nichž je řetězec složen. Operátory `get, put, copy, length, forall, getinterval, putinterval` fungují podobně jako u polí. Příklad:

```
/a 5 string def
(abcd) a copy
```

Čili vytvoříme objekt `a` jako řetězec délky 5 a následně ho naplníme postupně znaky abcd. Při definici řetězce operátorem `string` se vytvoří „prázdný“ řetězec tvořený znaky `\000` (v oktatovém zápisu znak číslo 0).

Pozornost je třeba věnovat operátoru `forall`, který v případě řetězce nejprve převádí jednotlivé znaky na celá čísla dle jejich ASCII kódu.

S řetězci se pojí dvojice vyhledávacích operátorů `search` a `anchorsearch`. Operátor `search` testuje řetězec na existenci podřetězce. Jeho syntaxe je:

```
řetězec1 řetězec2 search výsledek
```

v případě, že `řetězec2` je podřetězcem řetězce `řetězec1`, je výsledek hledání ve tvaru:

```
řetězec3 řetězec2 řetězec4 true
```

kde `řetězec3` je část řetězce `řetězec1` nacházející se za prvním výskytem řetězce `řetězec2` (někdy nazývaná postfix) a `řetězec4` je část řetězce `řetězec1` nacházející se před prvním výskytem řetězce `řetězec2`.

V případě, že `řetězec2` není obsažen v řetězci `řetězec1` (tzv. prefix)je výsledkem

```
řetězec1 false
```

Ukažme si na to několika příkladech.

```
(abcd)(cd) search   => ()(cd)(ab) true
(abcdab)(ab) search => (cdab)(ab)() true
(abcd) (ba) search  => (abcd) false
```

Znakem `=>` označujeme výsledek operace.

Operátor `anchorsearch` narozdíl od `search` kontroluje, zda `řetězec1` začíná řetězcem `řetězec2`. Pokud ano, je výsledek ve tvaru:

```
řetězec3 řetězec2 true
```

kde `řetězec3` je zbytek řetězce `řetězec1` následující po prvním výskytu řetězce `řetězec2`. V opačném případě je výsledek:

```
řetězec1 false
```

Ilustrujme si to opět na příkladu:

```
(abcd)(ab) anchorsearch => (cd)(ab) true
(abcd)(de) anchorsearch => (abcd)  false
(abcd)(cd) anchorsearch => (abcd)  false
```

Slovníky

Se slovníky jsme se již setkali v kapitole Nyní se budeme o slovníky zajímat spíše v kontextu ostatních složených objektů.

V kapitole jsme poznali operátor `dict` na založení slovníku a dvojici operátorů `begin` a `end` aktivující a deaktivující slovník.

Stavebními kameny slovníků jsou páry objektů, z nichž první je typu `jméno` a druhý je libovolný objekt. Tyto páry již nejsou ve slovníku nijak sekvenčně uspořádány, proto přístup k nim dle pořadí, jak tomu bylo např. u polí či řetězců, není z principiálního hlediska možný. Přistupovat k jednotlivým párům můžeme pouze pomocí první položky. Tu budeme nazývat `klíč`. Z výše uvedených skutečností je zřejmo, že se syntaxe operátorů `get`, `put` bude při použití se slovníky odlišná.

```
slovník klíč objekt put
slovník klíč get objekt
```

Operátor `put` vyhledá ve slovníku `slovník` klíč `klíč`. Pokud existuje, asociuje ho s objektem `objekt`. Pokud neexistuje, vytvoří nahrazovací pár `klíč objekt`.

Operátor `get` se pokusí vyhledat klíč `klíč` ve slovníku `slovník`. Pokud existuje, vrátí s ním asociovaný objekt. V opačném případě spustí chybovou proceduru `undefined` a oznámí (samozřejmě pokud neexistuje definice tohoto objektu v dotyčném slovníku):

```
Error: /undefined in --get--
```

Vzhledem k tomu, že slovníky nejsou sekvenčně uspořádány, nelze na ně aplikovat operátory `putinterval` a `getinterval`.

Operátor `copy` naopak funguje identicky jako u polí a řetězců, stejně jako operátor `length` vracející počet všech definovaných náhrad v daném slovníku. Se slovníky se navíc pojí operátor `maxlength` informující o tom, s jakou velikostí byl slovník vytvořen. V kapitole jsme se dozvěděli, že při překročení maximální kapacity se v PostScriptu Level 2 automaticky zvětší i hodnota `maxlength`, zatímco v případě Level 1 interpret oznámí chybu `dictfull`.

Z kapitoly též známe operátor `load`, který hledá nahrazovací pár k danému klíči ve všech aktivních slovnících (tj. všech na zásobníku slovníků) a vrací první nalezenou nahrazovací hodnotu, kterou uloží na zásobník operátorů. Opačně funguje operátor `store`. Jeho syntaxe je:

```
klíč objekt store
```

Nejprve vyhledá na zásobníku slovníků první (opět ve smyslu shora dolů) takový, ve kterém existuje asociační pár ke klíči `klíč`. Tento pár pak v nalezeném slovníku nahradí párem `klíč objekt`. Pokud nenajde odpovídající klíč v žád-

ném z aktivních slovníků, definuje jej v aktuálním (tj. nejvyšším) slovníku. Tedy provede:

```
/klíč objekt def
```

Někdy je potřeba zjistit, zda je klíč definován v daném slovníku. K tomu je určen operátor `known` se syntaxí:

```
slovník klíč known bool
```

kde `bool` je `true` v případě, že ve slovníku `slovník` existuje asociační pár ke klíči `klíč`, případně `false`, když neexistuje.

Hledáme-li definici konkrétního klíče v aktivních slovnících, využijeme operátor `where`:

```
klíč where výsledek
```

kde `výsledek` je v případě nalezení definice:

```
slovník true
```

a v případě, že klíči nepřísluší žádný asociační pár v žádném z aktivních slovníků:

```
false
```

Ostatní složené objekty disponovaly operátorem `forall`. Ani slovníky nejsou výjimkou. Syntaxe je opět:

```
slovník procedura forall
```

Jednotlivé asociační páry slovníku `slovník` jsou postupně umísťovány na zásobník operandů v pořadí `klíč objekt` a poté je spuštěna daná procedura `procedura`. Například operátor `copy` bychom mohli pomocí operátoru `forall` realizovat takto (promyslete):

```
slovník1 begin slovník2 {def} forall end
```

Při použití operátoru `forall` se slovníky musíme vědět následující moudra:

1. Pořadí, ve kterém budou jednotlivé páry zpracovávány, není stanoveno (narozdíl od ostatních složených objektů) a závisí na konkrétním interpretu.
2. Pokud vzniknou během aplikace procedury na jednotlivé páry některé nové definice ve zpracovávaném slovníku, operátor `forall` je může, ale nemusí zahrnout do své práce.

Jak vidno, snadno lze nevhodným použitím tohoto operátoru napsat program, který dva různé interprety PostScriptu interpretují odlišně.

Nakonec si ještě povězme o jedné vymoženosti PostScriptu Level 2, která nám umožňuje pohodlnější tvorbu uživatelských slovníků pomocí dvojitých lomených závorek `<<` a `>>`. Slovníky tak můžeme vytvářet podobně jako pole. Například:

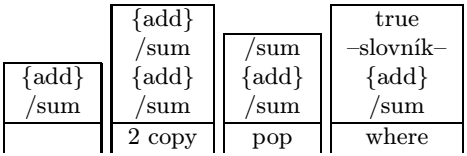
```
/new << /sum {add} /kvadr {dup mul} >> def
```

Tímto vytvoříme objekt `new` asociovaný se slovníkem obsahujícím definici jmen `sum` a `kvadr`. Takto vytvořené slovníky nazýváme anonymní. Vidíme, že asociační páry jsou ve slovníku uváděny postupně, nejprve klíč, pak asociovaný objekt. Tato konstrukce neumožňuje použití operátoru `bind`, které lze však nahradit užitím dvojitého lomítka `//`. Častější použití tohoto způsobu zadávání slovníků je u operátorů, které slovník vyžadují jako jeden z operandů, napří-

klad operátor `setpagedevice`, případně `sethalftone`. Užitečná může být také následující konstrukce:

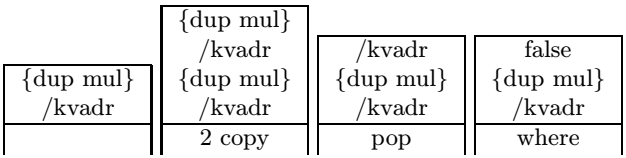
```
<< /sum {add} /kvadr {dup mul} >>
{2 copy pop where {pop pop pop}{def} ifelse} forall
```

která zjistí, zdali jsou definice v anonymním slovníku obsaženy již v některém z aktivních slovníků. Pokud ano, nestane se nic, pokud ne, vytvoří se nové definice v aktuálním slovníku. Podívejme se, co se vlastně děje. Uvažujme situaci, kdy v aktivních slovnících je jméno `sum` již definováno a jméno `kvadr` dosud ne. Operátor `forall` klade postupně asociační páry na zásobník a spouští definovanou proceduru. Nejprve, dejme tomu, pro dvojici `/sum` {`add`}:



kde `slovník` je aktivní slovník, v němž byla definice nalezena. Protože je na vrcholu zásobníku `true`, operátor `ifelse` zvolí větev výpočtu {`pop pop pop`} která smaže tři horní položky zásobníku (samotné `true` je odebráno operátorem `ifelse`).

V případě `kvadr` probíhá výpočet takto:



V tomto případě zvolí operátor `ifelse` větev {`def`}, která pouze aplikuje operátor `def`.

Stejně jako u polí levá závorka `<<` pracuje jako operátor `mark`, kdežto pravou závorku `>>` lze nahradit konstrukcí:

```
counttomark 2 idiv dup dict begin {def} repeat
pop currentdict end
```

Shrnutí

Složenými objekty v PostScriptu míníme pole (`array`), zhuštěné pole (`packed array`), řetězec (`string`) a asociativní pole (`dict`).

Jednoduché grafické práce

Až dosud to vypadalo, že PostScript je pouze poněkud podivný programovací jazyk. Teď si povíme něco o operátorech, které z PostScriptu dělají jazyk pro popis stránky. Nejprve si ale musíme ujasnit několik dalších konceptů.

Jedním z pilířů vektorové grafiky je v PostScriptu pojem *cesta*. Tedy čára, kterou lze následně vykreslovat, vyplňovat barvou nebo vzorem, či použít jako masku (ohraničuje-li nějakou uzavřenou oblast). Cesta může sestávat z úseček, kružnic či Bézierových křivek třetího stupně. Navíc může být i přerušená, či sestrojena z několika podcest (uzavřených i neuzavřených). K manipulaci s cestami je určeno několik specializovaných operátorů. Vysvětleme si jejich činnost na jednoduchém příkladu:

```
newpath
0 0     moveto
0 100   lineto
100 100 lineto
100 0   lineto
0   0   lineto
4 setlinewidth
stroke
showpage
```

Operátor `newpath` nejprve vynuluje proměnnou obsahující aktuální cestu. Dále je nastaven první bod cesty operátorem `moveto`. Tento operátor je nutno použít vždy po operátoru `newpath`; pokud tak neučiníte, interpret nahlásí z pochopitelných důvodů chybu. Aktuální bod cesty je v tomto okamžiku nastaven na souřadnice $(0, 0)$. Operátor `lineto` připojí nejprve k aktuální cestě úsečku z aktuálního bodu cesty do bodu daného dvojicí čísel z vrcholu zásobníku a zároveň na tento bod nastaví aktuální bod cesty. Posledním operátorem `lineto` končí konstrukce cesty. Operátorem `setlinewidth` nastavíme tloušťku pera na čtyři PostScriptové body a pomocí `stroke` aktuální cestu tímto perem vykreslíme. Obrázek se ovšem zobrazí až operátorem `showpage`.

Jak vidno, výše uvedené operátory lze rozdělit na ty, kterými cestu konstruujeme, a ty, kterými cestu vykreslujeme či vybarvujeme. Zastavme se na chvíli u těch konstrukčních.

Cesty jsou sestaveny ze segmentů, které mohou i nemusí být navzájem propojeny, mohou se překrývat, protínat, mohou být konvexní i konkávní. Základními konstrukčními prvky segmentů jsou přímky, Bézierovy křivky, kruhové oblouky a obrysy jednotlivých liter dostupných znakových sad.

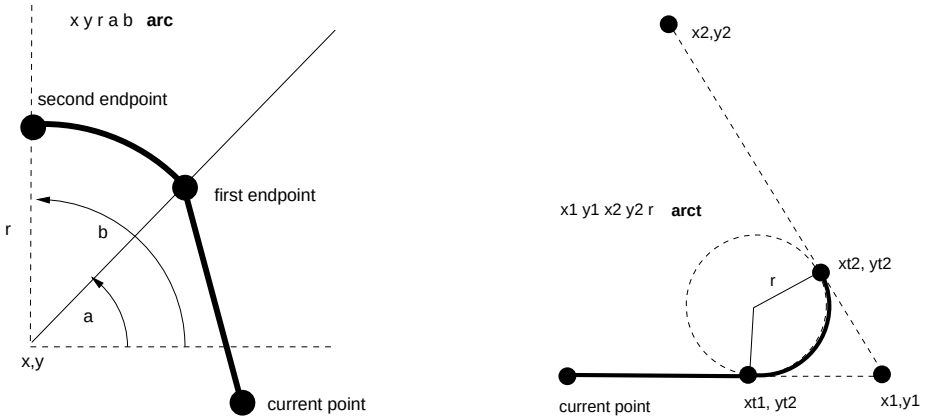
Cesty jsou reprezentovány interními strukturami a nelze k nim přistupovat jinak, než prostřednictvím konstrukčních operátorů (jednou zanesený segment již např nelze příliš editovat). Cesty nejsou ani PostScriptové objekty, takže je nelze triviálně asociovat s nějakým objektem typu jméno (jako například

slovníky, pole, atd.), nelze je tedy ani ukládat na zásobník operandů a následně s nimi provádět operace typu skládání jako např. v Metafontu. Standardním prostředkem, jak si ovšem části cesty uchovat, bude pro nás *zásobník grafických stavů*. Jisté, speciálně konstruované cesty si ovšem můžeme uložit jako takzvané uživatelské cesty (user paths).

Body zadáváme v PostScriptu pomocí jejich souřadnic (za chvíli uvidíme, že souřadnicový systém můžeme dost razantně změnit). Souřadnice jsou dvojice reálných čísel x, y udávající polohu bodu na aktuální stránce. Základní jednotkou jsou PostScriptové body (72 do palce).

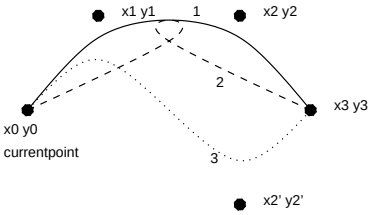
Každá nová cesta musí začínat operátorem `newpath`. Ten vymaže proměnnou *aktuální cesta (current path)*, pokud jsme si neuložili grafický stav, můžeme se s předchozí cestou rozloučit. *Aktuální bod cesty (current point)* je ovšem v tomto okamžiku nedefinován. Proto je nutné ho nastavit pomocí operátoru `moveto`. Tento operátor použijeme vždy, když bude třeba začít novou podcestu. Dalšími užitečnými operátory jsou:

- Operátor přidání úsečky z aktuálního bodu do bodu x, y `x y lineto`. Není-li nastaven aktuální bod, nahlásí chybu `nocurrentpoint`, podobně jako u ostatních operátorů tohoto typu.
- Další důležitou skupinou operátorů je tlupa `arc`, `arcn`, `arc`, `arcto` přidávající na konec cesty kruhový oblouk. Činnost operátorů `arc` a `arct` je nakreslena na obrázku 2. Varianta `acrn` funguje podobně jako `arc`, pouze se oblouk přidává po směru hodinových ručiček. Operátor `arcto` je variantou `arct`, pouze zanechá na zásobníku operandů hodnoty $xt1$ $yt1$ $xt2$ $yt2$ (tečné body).



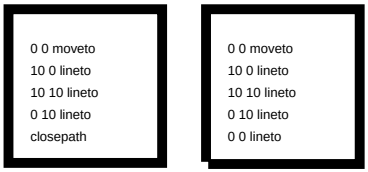
Obrázek 2: Činnost operátorů `arc` a `arct`. Tučnou čarou je zobrazena část cesty, která se přidá od *current pointu*.

- Ke konstrukci Bézierových křivek slouží operátor `curveto`. Schéma jeho činnosti je nakresleno na obrázku 3. Bodem $x0, y0$ je *current point*. O konstrukcích Bézierových křivek více v dodatcích.



Obrázek 3: Činnost operátoru `curveto` a různé tvary Bézierových křivek. 1. `x1 y1 x2 y2 x3 y3 curveto` 2. `x2 y2 x1 y1 x3 y3 curveto` 3. `x1 y1 x2' y2' x3 y3 curveto`

- Posledním operátorem zde zmíněným bude `closepath`. Ten uzavře právě kreslenou podcestu úsečkou z *current pointu* do posledního bodu nastaveného pomocí `moveto`. Nejvíce tento operátor využijeme v součinnosti se `stroke`. Na obrázku 4 vidíme jak zafunguje vykreslení cesty uzavřené pomocí `closepath` a co se stane při neuzavření cesty.



Obrázek 4: Chování operátoru `stroke` při použití a nepoužití `closepath`.

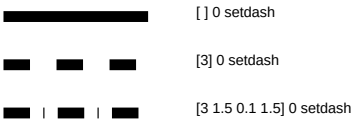
Předpokládejme nyní, že již máme nějakou cestu zkonstruovanou a rádi bychom ji vykreslili. K tomu nám poslouží operátory `stroke`, `fill` a `eofill`. Než je použijeme, musíme ovšem nastavit vlastnosti kreslícího nástroje. Pomocníkem nám bude skupina operátorů:

`num`	**setlinewidth**	Nastaví tloušťku pera.
`num`	**setgray**	Nastaví stupeň šedi. 0 je černá, 1 je bílá

`num1 num2 num3`	**setrgbcolor**	Nastaví barvu v RGB.
`[0 1 2]`	**setlinejoin**	Nastaví tvar spojů elementů cesty. 0 značí hranaté spoje, 1 kulaté, 2 s osekanými rohy viz 5
`[0 1 2]`	**setlinecap**	Nastaví tvar ukončení čar. 0 rovné, ukončené v koncových bodech , 1 kružnice se středem v koncových bodech a s poloměrem poloviny tloušťky pera, 2 rovné, ukončené ve vzdálenosti poloviční tloušťky pera za koncovým bodem.
`array num`	**setdash**	Nastavení vzoru čáry při použití operátoru `stroke`.

Obrázek 5: Ukázky různých nastavení `setlinejoin`.

Vysvětlení si zasluhuje snad jen operátor `setdash`. Při normálním kreslení čáry je čára vykreslována nepřerušovaně. Operátor `setdash` nastaví vzor čáry podle obsahu pole na zásobníku. Čísla obsažená v poli značí střídavě délku čáry a délku mezery. Tyto údaje se cyklicky probírají do konce vykreslování. Číslo na zásobníku udává tzv. *offset*, tedy délku, o kterou se zkrátí první kreslená čárka. Na obrázku 6 je uvedeno několik příkladů.



Obrázek 6: Použití `setdash`

Teď již jistě tušíte, jak funguje operátor `stroke`. Prostě obkreslí aktuální cestu pomocí specifikovaného nástroje. K vyplňování pomocí `fill` je třeba říci ještě pár maličkostí. Protože lze vyplňovat pouze uzavřené cesty, `fill` uzavře implicitně všechny podcesty. To se děje tak, že spojí přímkou konec podcesty s jejím začátkem. Začátek podcesty je dán vždy operátorem `moveto`, konec je dán posledním bodem před počátečním bodem následující podcesty. Dále provede vyplnění podle následujícího pravidla: *Bod bude vybarven, pokud součet orientovaných průsečíků po náhodně zvoleném paprsku z tohoto bodu do nekonečna je*

nenulový. Vidíme, že v tomto případě je důležitá orientace cesty — tomuto způsobu někdy říkáme *non-zero winding number rule*. V případě operátoru `eofill` se vyplňování řídí jinou strategií. Orientace při ní není nijak důležitá, počítá se pouze parita průsečíků. Při sudé paritě se bod vykreslí, při liché nikoliv. Tradiční název pro tuto strategii je *even-odd rule*.

Po použití vykreslovacích operátorů se implicitně zavolá `newpath`. Znamená to, že naše pracně zkonstruovaná cesta je navždy zapomenuta.

Kreslení by bylo v PostScriptu smutnou záležitostí, kdyby neexistoval pojem *grafický stav* a hlavně s ním související *zásobník grafických stavů*. Jedná se o soubor proměnných souvisejících s grafickými operátory. Asi tušíte, že jednou z proměnných bude aktuální cesta. Chceme-li proto cestu obkreslit i vyplnit, provedeme to posloupností:

```
gsave
stroke
grestore
fill
```

Operátor `gsave` uloží aktuální grafický stav na zásobník grafických stavů, odkud si jej můžeme vyzvednout pomocí `grestore`. Pokud nám současný grafický stav zvláště přirostl k srdci a chceme ho vyvolávat i v budoucnu, můžeme v Level 2 PostScriptu použít operátor `gstate`, který grafický stav převede na objekt (složený) a uloží jej na zásobník operandů. Obvyklé použití může být např.:

```
gstate
/mujgstate
exch
def
```

Stav dostaneme zpět prostřednictvím `setgstate`. tedy v našem případě:

```
mujgstate
setgstate
```

Následující tabulka obsahuje proměnné, které tvoří grafický stav. V této tabulce jsou zahrnuty pouze ty, které jsou součástí PostScriptu Level 2 a jsou nezávislé na výstupním zařízení.

proměnná	typ	popis
Transformační matice (CTM)	pole	Aktuální transformační matice zobrazení z uživatelského prostoru do prostoru zařízení.

Barva (color)	neurčeno	Aktuální barva pera určená k vyplňování, či vykreslování cest. Je závislá na hodnotě nastavení barevného prostoru.
Barevný prostor (color space)	pole	Určuje v jakém barevném prostoru se aktuální barvy nachází.
Aktuální bod (current point)	dvojice čísel	Souřadnice aktuálního bodu.
Aktuální cesta (current path)	interní	Aktuální cesta, tedy ta od posledního použití operátoru `newpath`.
Aktuální šablona (clipping path)	interní	Cesta která určuje co se nakonec skutečne zobrazí.
Aktuální font	slovník	Definice fontu, který je nastaven jako aktivní.
Šířka čáry (line width)	číslo	Šířka čáry, která se použije pro operátor `stroke`.
Ukončení čáry (line cap)	celé číslo	Nastavení tvaru konců čar.
Spojení čar (line join)	celé číslo	Nastavení spojovacích tvarů čar.
Miter limit	číslo	Omezení vzniku špiček při spojování čar pod malými úhly
Vzor čáry (dash pattern)	pole	Šablona, podle které se vykresluje čára.
Stroke adjust	true false	Určení, má-li dojít ke kompenzaci efektů způsobených malým rozlišením výstupního zařízení.

Shrnutí
Základem vektorové grafiky v PostScriptu je cesta. Každá cesta se může skládat z podcest. Podcestu zahájíme specifikací jejího počátečního bodu operátorem `moveto` a následně její tvar určíme posloupností operátorů `curveto`, `lineto`, `arc`, `arcn`, `arct`. Cesty mohou být uzavřené, nebo otevřené. Uzavřené cesty se specifikují operátorem `closepath`. Ten spojí úsečkou poslední bod cesty s bodem určeným posledním operátorem `moveto`

Na již vytvořené cesty můžeme aplikovat operátor obkreslení `stroke`, nebo vyplnění barvou `fill`. Před vyplněním se na všechny neuzavřené cesty aplikuje operátor `closepath`. K určení barvy či vlastností pera slouží např. operátory `setgray`, `setlinewidth`, `setdash` aj.

Většina zmíněných operátorů mění obsah proměnných tvořících dohromady tzv. grafický stav. Grafické stavy se mohou ukládat pro pozdější využití na zásobník grafických stavů. K manipulaci s tímto zásobníkem slouží operátory `gsave` a `grestore`.

Fonty a PostScript

Základní použití fontů

Začněme opět malým příkladem.

```
/Helvetica findfont
12 scalefont setfont
288 720 moveto
(Nějaký text) show
```

Jednotlivé řádky dělají následující:

- vybere se font
- zvětší se
- aktuální bod se nastaví na zadanou pozici
- vypíše se text

Interpret PostScriptu má fonty uloženy v adresáři fontů *(font directory)*, tak že asociuje vždy název fontu s jeho definicí. Operátor `findfont` má jeden parametr — název fontu a vrací odkaz na definici fontu. Operátor `scalefont` zvětšuje velikost fontu, má dva parametry: původní definici fontu a měřítko, vrací novou definici fontu. `setfont` nastaví svůj parametr jako aktuální font. `show` z vrcholu zásobníku získá řetězec a vypíše ho.

Efekty s fonty

```
/Helvetica-Bold findfont 48 scalefont setfont
20 40 moveto
.5 setgray
(XYZ) show
```

Tedy celkem normální použití operátoru `setgray`.

```
/Helvetica findfont 48 scalefont setfont
20 40 moveto
(ABC) false charpath
2 setlinewidth stroke
```

Operátor `charpath` vytváří cestu (path) z obrysu definice písma. S touto cestou pak může být nakládáno jako s jinou cestou, tedy vykreslení třeba operátorem `stroke` nebo se touto cestou může ořezat výstup následujících operátoru jako je tomu v následujícím příkladu. Upozornění: z důvodu omezeného počtu elementů v cestě nepoužívejte `charpath` na více než jen několik znaků.

```
/Helvetica findfont 48 scalefont setfont
newpath 20 40 moveto (PQR) false charpath clip
...
```

Opět vcelku normální použití tentokráte operátoru `clip`

Definice fontu

Každý font je popsán v objektu typu slovník *(font dictionary)*. Je to normální slovník, až na to, že musí obsahovat určené páry klíč-hodnota. PostScript rozlišuje několik druhů fontů, a to podle hodnoty klíče *FontType*.

- Typ 0 je složený font *(composite font)*, definovaný pomocí odkazů na jiné základní typy
- Typ 1 je základní typ fontu, podrobnosti kódování viz Adobe Type 1 Font Format
- Typ 3 uživatelem definovaný typ fontu a to tak, že tvar každého znaku je popsán jako PostScriptová procedura

PostScriptový program vytváří slovník fontu pomocí standardních prostředků pro práci se slovníky (dict,begin,end,def). Po vytvoření slovníku oznámí jeho existenci interpretu voláním definefont (s parametry jméno a slovník).

Kódování znaků

Ve slovníku fontu mají popisy znaků jako klíč název znaku a ne jeho číselný kód. Písmena mají jako název sama sebe (třeba 'A'), ostatní znaky mají název složený ze slov (třeba 'three' nebo 'ampersand'). Číselné kódy znaků se na jména převádějí pomocí pole *Encoding* ze slovníku fontu. Jedná se o pole indexované

Klíč	Typ	Povinný	Význam
FontType	integer	+	typ fontu
FontMatrix	array	+	transformační matice ze souřadného systému popisu znaků do uživatelského souřadného systému, například fonty Type 1 od Adobe jsou obvykle definovány jako znaky veliké 1000 jednotek, potom transformační matice vypadá takto [0.001 0 0 0.001 0 0]
FontName	name	–	jméno fontu, interpret PostScriptu jej nijak nepoužívá
FontInfo	dictionary	–	viz dále
LanguageLevel	integer	–	1 nebo 2 podle požadované minimální úrovně interpretu PostScriptu nutné k použití tohoto fontu
WMode	integer	–	přepínač která ze dvou metrik se použije při výpisu znaků tohoto fontu, Level 1 interpret tuto položku ignoruje

Tabulka 1: Položky společné všem typům fontů

čísly 0 až 255, prvky pole jsou jména znaků. Důvodem, proč se používá takovéto schéma, je umožnit snadnou změnu kódování znaků. Skutečně, při změně kódování je třeba změnit jen pole Encoding mapující číselné kódy na jména, samotné definice obrazů znaků zůstávají beze změny. Nepoužité pozice ve vektoru Encoding musí být vyplněny jménem .nodef.

Metrika fontů

Tvar znaku je definován křivkami v souřadné soustavě znaku (character coordinate system). Počátek, nebo také referenční bod, znaku je bod (0,0) v souřadné soustavě znaku. `show` a jiné operátory kreslící znaky umístí počátek prvního znaku na pozici aktuálního bodu v uživatelském systému souřadnic. *Šířka* (width) znaku je vektor z počátku znaku do pozice, kam se má umístit počátek následujícího znaku. Většina indoevropských abeced má kladnou složku x a nulovou složku y. *Meze* (bounding box) znaku tvoří nejmenší obdélník (orientovaný rovnoběžně s osami souřadné soustavy znaku), který obsahuje celý tvar znaku. Meze jsou vyjádřeny pomocí dolního-levého a horního-pravého rohu relativně

Klíč	Typ	Povinný	Význam
Encoding	array	+	Pole jmen znaků sloužící k mapování číselných hodnot na znaky.
FontBBox	array	+	Pole čtyř hodnot v souřadném systému popisu znaků popisující nejmenší obdélník obsahující všechny znaky fontu nakreslené v počátku. Používá se jako pomocná informace při kešování a ořezávání fontů. Pořadí hodnot v poli je: dolní-levé x,dolní-levé y,horní-pravé x,horní-pravé y.
UniqueID	integer	−	Číslo od 0 do 16777215 jednoznačně identifikující font.
XUID	array	−	Pole celých čísel jednoznačně identifikující font nebo jeho variantu. Level 1 ignoruje tuto položku.

Tabulka 2: Položky základních typů fontů (tedy bez FontType=0)

k počátku souřadné soustavy znaku. *Odsazení* (left sidebearing) znaku je vzdálenost počátku od průsečíku mezí a základní čáry. Pokud znak přesahuje vlevo od svého počátku, může být x složka záporná. y složka je téměř vždy nulová. V některých (například asijských) jazycích je běžné psát text ve dvou různých směrech (vodorovně a svisle). Font, který má být použit pro oba směry, musí obsahovat dvě různé sady informací o počátcích a šířkách znaků. K přepnutí způsobu psaní slouží položka *WMode* fontu.

Arnošt Štědrý

Klíč	Typ	Povinný	Význam
PaintType	integer	+	Určuje způsob kreslení písmen. 0 — obrys je vyplněn, 2 — pouze obrys
StrokeWidth	number	–	Šířka obrysu v jednotkách souřadného systému znaků.
Metrics	dictionary	–	Informace o šířce a odsazení. Pokud je tato položka uvedena, má přednost před definicemi uvedenými v popisu znaků.
Metrics2	dictionary	–	Totéž co Metrics jen pro WMode rovno 1.
CDevProc	procedure	–	Procedura pro globální změny metriky fontu. Level 1 ignoruje tuto položku.
CharStrings	dictionary	+	Asociuje jména znaků s jejich definicemi. Definice jsou kódovaný dle Adobe Type 1 Font Format nebo jsou to PostScriptové procedury.
Private	dictionary	+	Obsahuje další informace o fontu. Pro detaily viz Adobe Type 1 Font Format.

Tabulka 3: Položky fontů Type 1

Klíč	Typ	Povinný	Význam
FamilyName	string	–	Jméno skupiny fontů.
FullName	string	–	Jednoznačné jméno fontu.
Notice	string	–	Ochraná známka, copyright a podobně.
Weight	string	–	Jméno tloušťky písma.
version	string	–	Verze.
ItalicAngle	number	–	Úhel ve stupních určující naklonění italiky.
isFixedPitch	boolean	–	True znamená, že font má všechny znaky stejně široké.
UnderlinePosition	number	–	Doporučená vzdálenost podtržení od základní čáry.
UnderlineThickness	number	–	Doporučená šířka podtržení.

Tabulka 4: Položky slovníku FontInfo