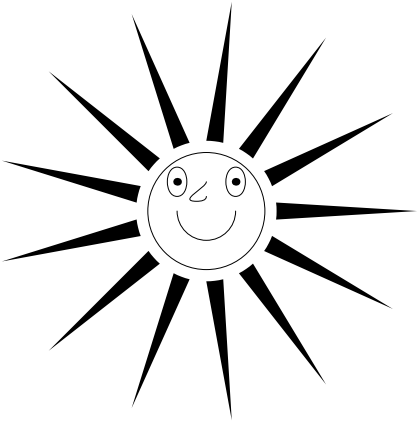
CSTUG



ZPRAVODAJ

Československého sdružení uživatelů TEXu

1–3
2001

ISSN 1211-6661 Ročník 11

OBSAH

Toto číslo obsahuje CD-ROM TEX Live 6.

Zpravodaj Československého sdružení uživatelů TEXu je vydáván v tištěné podobě a distribuován zdarma členům sdružení. Po uplynutí dvanácti měsíců od tištěného vydání je poskytován v elektronické podobě (PDF) ve veřejně přístupném archívu dostupném přes `http://www.cstug.cz`.

Své příspěvky do Zpravodaje můžete zasílat v elektronické podobě anonymním ftp na `ftp.icpf.cas.cz` do adresáře `/wagner/incoming/`, nejlépe jako jeden archivní soubor (`.zip, .arj, .tar.gz`). Současně zašlete elektronickou poštou upozornění na `mailto:bulletin@cstug.cz`. Uvedený adresář je přístupný pouze pro zápis. Pokud nemáte přístup na Internet, můžete zaslat příspěvek na disketě na adresu:

Zdeněk Wagner
Vinohradská 114
130 00 Praha 3

Disketu formátujte nejlépe pro DOS, formáty Macintosh 1.44 MB a EXT2 jsou též přijatelné. Nezapomeňte přiložit všechny soubory, které dokument načítá (s výjimkou standardních součástí CSTEXu), zejména v případě, kdy vás nelze kontaktovat e-mailem.

ISSN 1211-6661

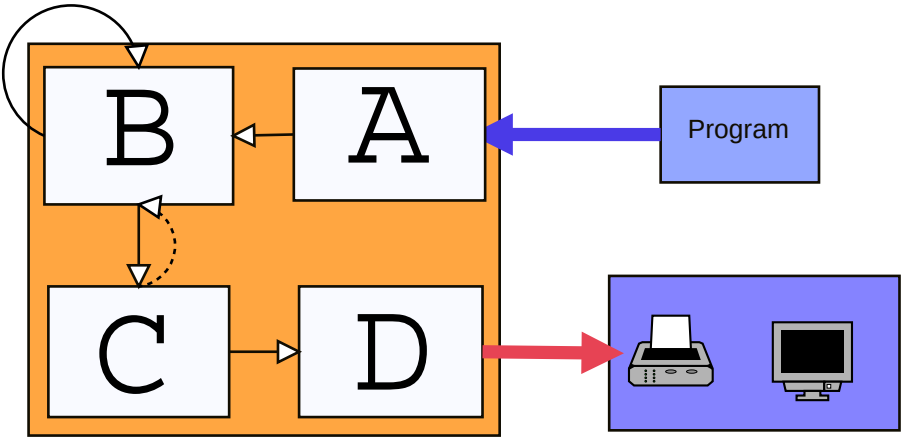
Základní mechanismy PostScriptu

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.

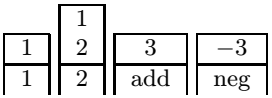
Speciální znaky mají, poplatny svému označení, v PostScriptu zvláštní funkce. Jsou to zejména:

- % 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:

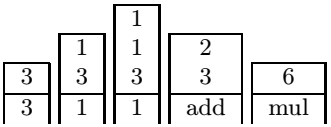


Pod čarou jsou zapsány jednotlivé postupně interpretované objekty, nad čarou jsou zaznamenány změny zásobníku. Nové objekty jsou v tomto zobrazení přidávány zdola.

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` $\qquad y1 = num1 + num2$

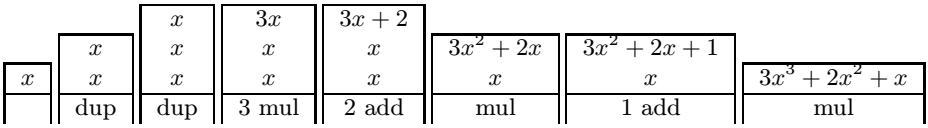
`num1`	`num2`	**sub**	`y1`	$y1 = num1 - num2$
`num1`	`num2`	**mul**	`y1`	$y1 = num1 \times num2$
`num1`	`num2`	**div**	`y1`	$y1 = num1/num2$
`num1`		**abs**	`y1`	$y1 = \|num1\|$
`num1`		**neg**	`y1`	$y1 = -num1$
`num1`		**ceiling**	`int1`	$int1$ je nejbližší vyšší celé číslo k $num1$.
`num1`		**floor**	`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
```

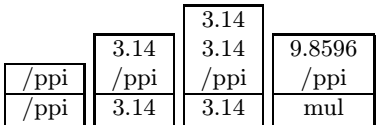
pak se hodnota `ppi` bude měnit v závislosti na hodnotě `pi`, protože expanze se bude provádět až v okamžiku použití[4]. První způsob je vhodný pro definici konstant, druhým způsobem můžeme definovat funkce. V druhém případě se nám jméno `ppi` přiřazuje složenému proveditelnému objektu — proceduře, zatímco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



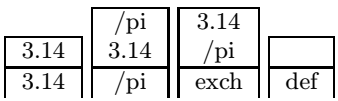
Operátor `def` má tuto syntaxi:
```
jmeno objekt def
```
kde `jmeno` je objekt typu jmeno a `objekt` je libovolný PostScriptový objekt.

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:

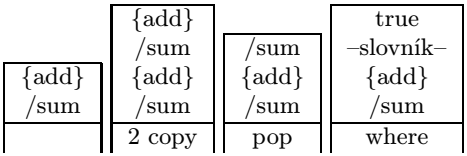
```
/new << /sum {add} /kvadr {dup mul} >> def
```

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:

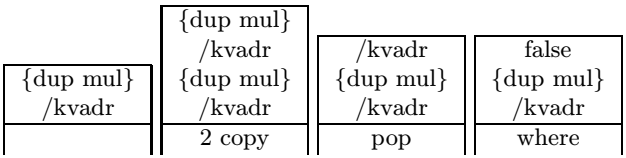
```
<< /sum {add} /kvadr {dup mul} >>
{2 copy pop where {pop pop pop}{def} ifelse} forall
```

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.

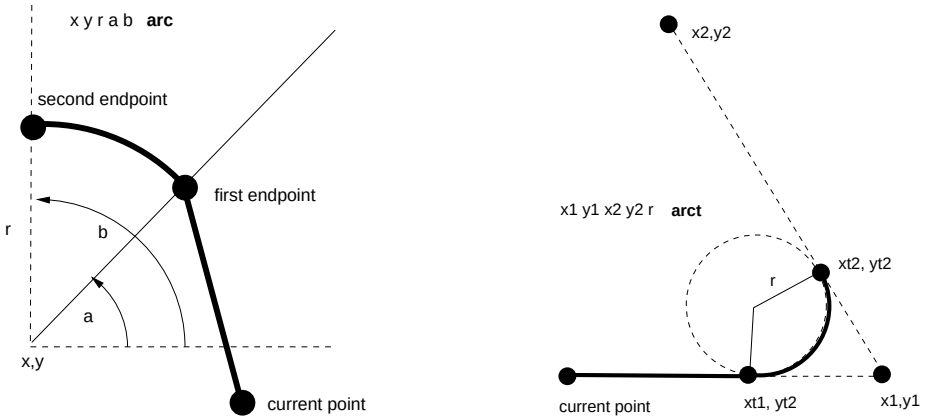
Cesty jsou reprezentovány interními strukturami a nelze k nim přistupovat jinak, než prostřednictvím konstrukčních operátorů (jednou zanesený segment již např nelze příliš editovat). Cesty nejsou ani PostScriptové objekty, takže je nelze triviálně asociovat s nějakým objektem typu jméno (jako například

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).

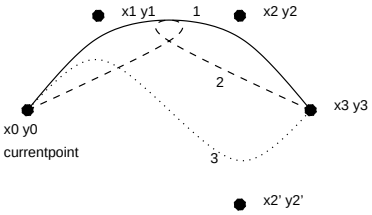
Každá nová cesta musí začínat operátorem `newpath`. Ten vymaže proměnnou *aktuální cesta (current path)*, pokud jsme si neuložili grafický stav, můžeme se s předchozí cestou rozloučit. *Aktuální bod cesty (current point)* je ovšem v tomto okamžiku nedefinován. Proto je nutné ho nastavit pomocí operátoru `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).



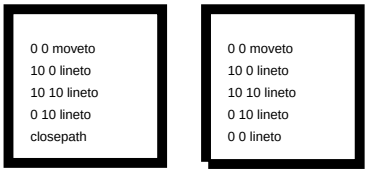
Obrázek 2: Činnost operátorů `arc` a `arct`. Tučnou čarou je zobrazena část cesty, která se přidá od *current pointu*.

- 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`.

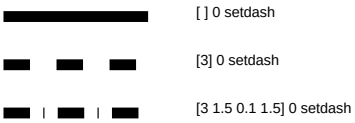
Předpokládejme nyní, že již máme nějakou cestu zkonstruovanou a rádi bychom ji vykreslili. K tomu nám poslouží operátory `stroke`, `fill` a `eofill`. Než je použijeme, musíme ovšem nastavit vlastnosti kreslícího nástroje. Pomocníkem nám bude skupina operátorů:

`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.

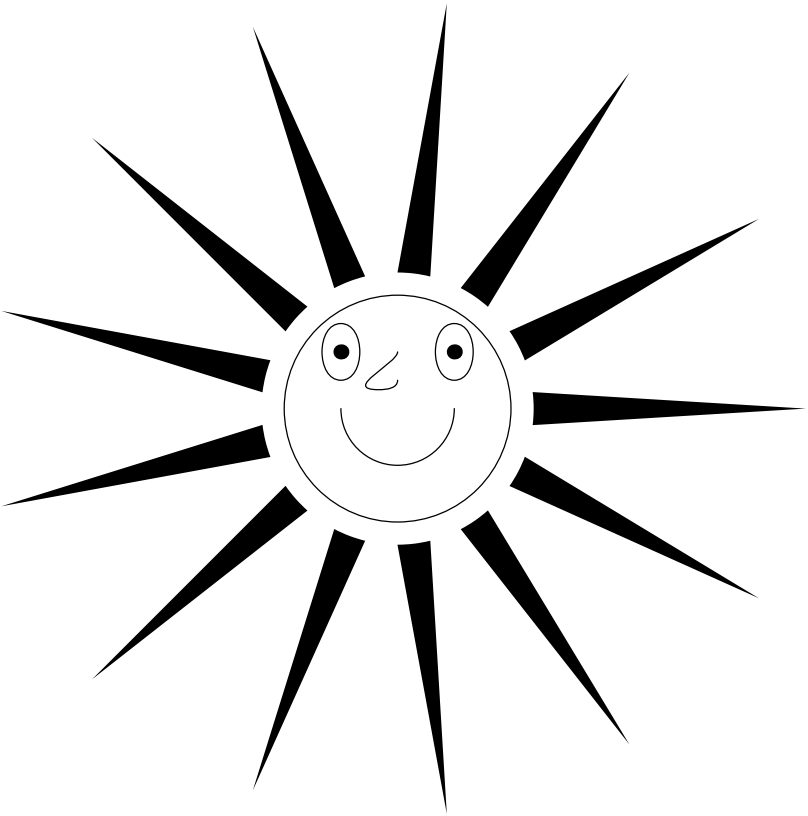
Tabulka 3: Položky fontů Type 1

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

METAPOST a mfpic – první část

Miroslava Krátká



Obsah

1. METAPOST

METAPOST je programovací jazyk určený pro popis (a následné kreslení) obrázků. Jeho autorem je John D. Hobby. METAPOST vznikl z jazyku METAFONT, který již v roce 1977 začal vytvářet společně s typografickým systémem TEX Donald E. Knuth. První zmínka o METAPOSTu se objevila v roce 1989.

Zdrojový text METAPOSTu je posloupnost příkazů oddělených středníky. Přesnější syntaxe souboru bude vysvětlena později. Překladač mívá obvykle podobný název, v operačním systému Linux je to `mpost`. Výstupem překladu je program v jazyce PostScript vykreslující obrázek.

1.1. Datové typy

Každý objekt v METAPOSTu má svůj datový typ. METAPOST podporuje následující datové typy:

1. `numeric` pro uložení čísla,
2. `pair` pro uložení souřadnic,
3. `path` pro uložení cesty, to jest křivky,
4. `transform` pro uložení afinní transformace,
5. `color` pro uložení barvy,
6. `string` pro uložení řetězce,
7. `boolean` pro uložení proměnné booleovského typu,
8. `picture` pro uložení obrázku,
9. `pen` pro uložení pera.

Čísla jsou reprezentována jako k-násobky zlomku $\frac{1}{65536}$, kde k je celé číslo. Jejich absolutní hodnota musí být menší než 4096, ale průběžné výsledky mohou být až osmkrát větší.

Souřadnice bodů jsou popisovány dvojicí čísel. Souřadnice mohou být vzájemně sčítány a odčítány, dále násobeny a děleny číslem.

Cesta reprezentuje lomenou čáru nebo křivku, danou parametrickým vyjádřením.

Transformací může být libovolná kombinace otáčení, stejnolehlosti, zkosení a posunutí. Transformace bývá aplikována na cesty, obrázky a pera.

Datový typ `color` je podobný souřadnicovému typu; nejsou použity dvě komponenty, ale tři, přičemž jednotlivé komponenty (R, G, B) odpovídají postupně podílu červené, zelené a modré barvy. Velikost každé z komponent nesmí překročit meze intervalu $[0, 1]$. Předdefinovány jsou `black`, `white`, `red`, `green` a `blue` pro černou, bílou, červenou, zelenou a modrou barvu. Černá barva odpovídá (0,0,0) a bílá (1,1,1). Barvy se mohou navzájem sčítat a odčítat. Barvu lze také násobit reálným číslem. Je-li požadována šedá barva, například (0.7,0.7,0.7), je možné použít zápis `0.7white`.

Řetězce jsou reprezentovány posloupností písmen v uvozovkách.

Booleovské proměnné nabývají hodnot `true` nebo `false` a existují pro ně operátory `and`, `or`, `not`.

Výsledky příkazů pro kreslení jsou ukládány do speciálních proměnných typu `picture`, například u příkazu `draw` je to `currentpicture`. Obrázky lze přidávat k jiným obrázkům a je možné na ně aplikovat afinní transformace.

Datovému typu `pen` je věnován odstavec Pera na straně 45.

Deklarace proměnných

K deklaraci proměnné se v METAPOSTu používá příkaz

`typ` *název_proměnné* .

Typ proměnné je jeden z výše uvedených datových typů. Pro deklaraci celého pole proměnných se použije *název_proměnné*`[]`. Proměnné, kterým není přiřazen typ, jsou METAPOSTem chápány, jako by byly typu `numeric`. Je-li *název_proměnné* přiřazen nějakému typu, je tato deklarace platná v celém zdrojovém textu, ne pouze v jednotlivých obrázcích (zdrojový text totiž smí obsahovat popisy více obrázků, viz odstavec Vstupní soubor na straně 43).

Proměnné `z`*přípona* jsou předdefinovány jako dvojice (`x`*přípona*,`y`*přípona*), přičemž *přípona* je složena z tokenů, o kterých se zmiňuji v následujícím odstavci. Na začátku každého obrázku, to znamená vždy po příkazu `beginfig`, nejsou dvojicím `z`*přípona*=(`x`*přípona*,`y`*přípona*) přiřazeny hodnoty.

Ke zjištění typu proměnné nebo její hodnoty slouží příkaz

`show` *název_proměnné* .

Tokeny

Tokeny jsou základními stavebními prvky METAPOSTu. Vstupní soubor se skládá z číselných tokenů, řetězcových tokenů a symbolických tokenů.

Nejpoužívanějšími symbolickými tokeny jsou velká a malá písmena anglické abecedy a znak podtržítka (_). Mezi významné tokeny patří znak procenta (%), který způsobí ignorování zbývajícího kódu na aktuálním řádku, a také znak tečky (.), záleží však na počtu teček vyskytujících se za sebou. Dvě a více teček dohromady tvoří symbolický token (například `..` nebo `...`), tečka stojící před a za číslicemi je součástí číselného tokenu. V případě, že jedna tečka není obklopena číslicemi, a tedy není částí čísla, je tato tečka ignorována. Tohoto lze využít pro přehledné pojmenování proměnných, například `m.a`, přičemž se tento název skládá ze dvou tokenů `m` a `a`. Znaky `,;()` – čárka, středník a kulaté závorky – jsou považovány za samostatný token, i když stojí těsně za sebou.

Způsob, jak METAPOST zachází s tokeny, a tabulku tokenů najdeme v knize METAFONTbook ([3, str. 50]) a v manuálu k METAPOSTu ([2, str.16]).

Symbolické tokeny rozdělujeme do dvou skupin. Symbolický token bez speciálního významu, například jméno námi nadefinované proměnné, se nazývá **tag**. Symbolický token, který nese název primitivního příkazu (viz strana 56) nebo byl definován pomocí `def` jako makro (viz odstavec Makra na straně 56), se nazývá **spark**.

1.2. Vstupní soubor

Vstupní soubor mívá obvykle příponu `.mp`. Soubor `obrazek.mp` může vypadat například takto:

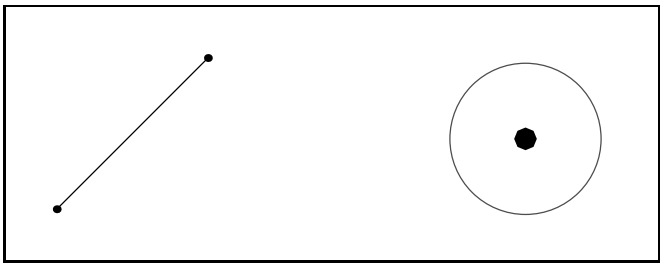
```
prologues:=1;
u:=1cm;
pen tluste;

beginfig(1);
z1=(u,u);
z2=(3u,3u);
draw z1--z2;
pickup pencircle scaled 3pt;
draw z1;
draw z2;
endfig;

beginfig(3);
draw fullcircle scaled 2u shifted (5u,5u) withcolor 0.3white;
tluste:=makepen(fullcircle scaled 0.3u);
```

```
draw (5u,5u) withpen tluste;
endfig;

end
```

Význam prvního řádku obsahujícího přiřazení proměnné `prologues` vysvětluje odstavec Prologues (strana 62). Dvojice příkazů `beginfig`(*číslo*) ... `endfig` ohraničuje příkazy popisující jeden obrázek.V souboru může být popsáno obrázků několik. Má-li více obrázků stejné *číslo*, bude výsledný obrázek s daným *číslem* odpovídat popisu v pořadí posledního obrázku, který se ve vstupním souboru objevuje v okolí `beginfig`(*číslo*) ... `endfig`. Vstupní soubor ukončuje příkaz `end`.

Při spuštění překladače je možné u jména souboru vynechat příponu `.mp`. Pro překlad souboru `obrazek.mp` tedy postačí zadat `mpost obrazek`.

Uvedeným postupem ze vstupního souboru `obrazek.mp` vzniknou tři soubory, a to `obrazek.1`, `obrazek.3`, které obsahují program v jazyce PostScript vykreslující dané obrázky, a soubor `obrazek.log`, v němž je zaznamenán postup překladu a případná chybová hlášení.



Obrázek 1: obrazek.1 a obrazek.3

Obrázek si prohlédneme například pomocí programu Ghostview, nebo jej vložíme do TEXovského dokumentu; ten zpracujeme odpovídajícím způsobem a prohlédneme vhodným prohlížečem. Vkládání obrázku je závislé na použitém formátu TEXu:

- v plain TEXu, $\mathcal{AMS}$-TEXu a LATEXu 2.09: \`epsfbox` {*jméno_souboru*} (potřebujeme soubor `epsf.tex`),
- v pdfTEXu: \`convertMPtoPDF` {*jméno_souboru*}{1}{1} (jsou nezbytné soubory `supp-pdf.tex` a `supp-mis.tex`),
- v LATEX 2$_\varepsilon$a pdfLATEXu: \`includegraphics` {*jméno_souboru*} (předpokládá načtení balíků `graphics` nebo `graphicx`, pro pdfLATEX s volbou `pdftex`, přičemž jsou ještě potřebné soubory `pdftex.def`, `supp-pdf.tex` a `supp-mis.tex`).

Protože METAPOST vytváří soubory s příponou *.číslo*, je při použití pdfLaTeXu nutno uvést příkaz

`\DeclareGraphicsRule {*} {mps} {*} {}` .

Po překladu dostaneme soubory s příponou `.dvi`, respektive `.pdf`. První z nich ještě zpracujeme programem `dvips` (neboť při prohlížení `.dvi` souboru nemusí být viditelné obrázky) a vzniklý postscriptový soubor prohlédneme pomocí Ghostview; soubor `.pdf` prohlédneme například prohlížečem Acrobat Reader.

1.3. Základní příkazy kreslení

Jednotky

METAPOST užívá základní systém souřadnic shodný se souřadným systémem jazyka PostScript. Jednotkou tohoto systému je postscriptový bod o velikosti $\frac{1}{72}$ palce, bývá označován **bp**. Dalšími jednotkami, které METAPOST zná, jsou **pt** o velikosti $\frac{1}{72{,}27}$ palce, **in** pro palec, **cm** pro centimetr a **mm** pro milimetr.

Pro jednodušší úpravu měřítka je výhodné přiřazení jednotky nějaké proměnné, `u:=cm` apod. Rozlišujeme `u=cm` (rovnice) a `u:=cm` (přiřazení). Následné `u=2cm` by dalo chybu, zatímco `u:=2cm` by změnilo hodnotu `u`.

Body

Ve všech příkazech kreslení je možno používat přímo souřadnicový zápis bodů, ale pohodlnější a přehlednější je nadefinování bodů v úvodu vstupního souboru, například `z1=(3cm,1cm)`. Bod `z`*číslo* reprezentuje bod o souřadnicích (`x`*číslo*,`y`*číslo*). Souřadnice bodů je možné navzájem sčítat a odčítat. Souřadnice bodu lze násobit a dělit číslem. METAPOST také řeší lineární rovnice. Je tedy možné zadat body $Z_1 = [3, 1]$ a $Z_2 = [-3, 7]$ například takto:

```
x1=-x2=3cm;        x1+y1=x2+y2=4cm;
```

Střed úsečky lze zjistit jednoduše, pomocí zápisu `z3=1/2[z1,z2]`.

Jako pomocnou neznámou, jejíž přesná hodnota pro nás v danou chvíli není důležitá, používáme `whatever`. Velice vhodné je její použití při hledání průsečíku dvou přímek, tj. řešení soustavy lineárních rovnic:

```
z5=whatever*[z1,z2]=whatever*[z3,z4];
```

Hledaným průsečíkem je zde `z5`. Neznámou `whatever` lze použít vícekrát, jednotlivé hodnoty jsou na sobě nezávislé.

Pera

Jedním ze základních požadavků pro kreslení je umožnění změny tloušťky pera. Příkazem

`pencircle scaled` *číslo*

zvolíme kruhové pero zadané tloušťky. Na začátku každého obrázku, tedy po příkazu `beginfig`, je nastaveno pero o tloušťce 0,5 bp. Je-li třeba získat například pero s kaligrafickým efektem, může být použita některá z lineárních transformací. Vlastní nastavení požadovaného pera se provede příkazem

`pickup` *název_pera* .

K opětnému získání přednastavené hodnoty tloušťky pera lze použít příkazu

`pickup defaultpen` .

METAPOST disponuje také perem `pensquare`, jež je vytvořeno příkazem

`makepen((-.5,-.5)--(.5,-.5)--(.5,.5)--(-.5,.5)--cycle)` .

Je tedy zřejmé, že příkaz `makepen` *cesta* vytvoří pero tvaru dané cesty. Opačným příkazem je `makepath` *pero*, který danému peru přiřadí jako výstup odpovídající cestu.

Body, lomené čáry a křivky

Bod se nakreslí příkazem

`drawdot` *bod* .

Lomená čára se vykreslí příkazem

`draw` *bod*`--`*bod*`--`*bod* ,

uzavřená lomená čára potom

`draw` *bod*`--`*bod*`--`*bod*`--cycle` .

METAPOST kreslí Bézierovy kubiky procházející zadanými body. Sám si vypočítá kontrolní body tak, aby se křivka jevila co „nejhezčí“. Příkaz

`draw` *bod*`..`*bod*`..`*bod*

popisuje Bézierovu křivku a příkaz

`draw` *bod*`..`*bod*`..`*bod*`..cycle`

udává uzavřenou křivku.

Dále máme několik možností, jak tvar křivky upravit podle svých představ. Sklon tečny v bodě křivky ovlivníme příkazem

`draw` *bod*`..`*bod*`{dir` *číslo*`}..{dir` *číslo*`}`*bod* .

Můžeme také použít předdefinované směry `up`, `down`, `left`, `right`.

Napětí křivky se upraví příkazem

`draw` *bod*`..`*bod*`..`*bod*`..tension` *číslo* `and` *číslo*`..`*bod*`..`*bod* .

Číslo je desetinné číslo větší než $\frac{3}{4}$; hodnota 1 odpovídá použití „..“. Pro `tension` *nekonečno* je kromě `tension infinity` k dispozici „...“.

Zakřivení cesty lze ovlivnit příkazem

`draw` *bod*`{curl` *číslo*`}..`*bod*`..{curl` *číslo*`}..`*bod* .

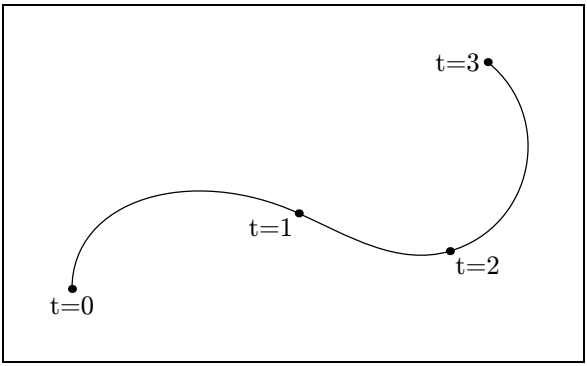
Číslo menší než 1 snižuje zakřivení, *číslo* větší než 1 jej zmenšuje. Poslední možností je přímé zadání kontrolních bodů příkazem

`..controls` *kontrolní_bod* `and` *kontrolní_bod*`..` .

Pro vymazání slouží příkazy `undrawdot` a `undraw` se stejnou syntaxí.

Parametrizace křivek

METAPOST pracuje s cestami jako parametrizovanými křivkami a tím umožňuje zjišťovat o cestách (křivkách) mnoho užitečných informací. Křivka definovaná parametricky je zapisována jako množina bodů $(X(t), Y(t))$, kde t se pohybuje od nuly k číslu udávajícímu počet křivkových segmentů (tj. k číslu o jedničku menšímu než počet zadaných bodů křivky). Hodnota t se nazývá čas.



Obrázek 2: Křivka definovaná parametricky

Průsečík dvou křivek se zjišťuje příkazy

a `intersectionpoint` *b* ,

a `intersectiontimes` *b* .

kde *a* a *b* jsou cesty. Výsledkem prvního příkazu jsou souřadnice průsečíku (neprotínají-li se zadané cesty, METAPOST hlásí chybu); výsledkem druhého je dvojice (t_a, t_b), t_a je čas na cestě *a* v průsečíku cest *a* a *b*, t_b je odpovídající čas na cestě *b*. Je-li výsledkem $(-1, -1)$, cesty se neprotínají. Může se stát, že se dvě křivky protínají ve více bodech; METAPOST v tomto případě vybírá (t_a, t_b) s nejmenším t_a, to je ovšem velmi zjednodušeně řečeno, ve složitějších případech jsou prováděna ještě další porovnávání. Podrobnější vysvětlení najdeme v knize METAFONTbook ([3]).

Pro zjištění souřadnice bodu na křivce slouží příkaz

`point` *t* `of` *cesta* .

Čas celé křivky udává příkaz

`length` *cesta* .

Příkaz

`subpath` (t_1, t_2) `of` *cesta*

vyjme z *cesty* úsek mezi časy t_1 a t_2.

S operací `subpath` pracují další dva příkazy

a `cutbefore` *b* ,

a `cutafter` *b* ,

kde *a* a *b* jsou cesty, jejichž průsečík je bod *P*. První z příkazů vybere úsek cesty *a* od bodu *P* ke konci, druhý vybere naopak začátek *a* až k bodu *P*.

Tečný vektor křivky v jejím libovolném *t* vrací příkaz

`direction` *t* `of` *cesta* ,

t příslušný zadanému tečnému vektoru zase příkaz

`directiontime` *vektor* `of` *cesta*

a bod odpovídající tečnému vektoru příkaz

`directionpoint` *vektor* `of` *cesta* .

Příkazem

`arclength` *cesta*

se zjišťuje oblouková míra křivky.

Pro zjištění času odpovídajícího obloukové míře *a* na křivce *p* je připraven příkaz

`arctime` *a* `of` *p* .

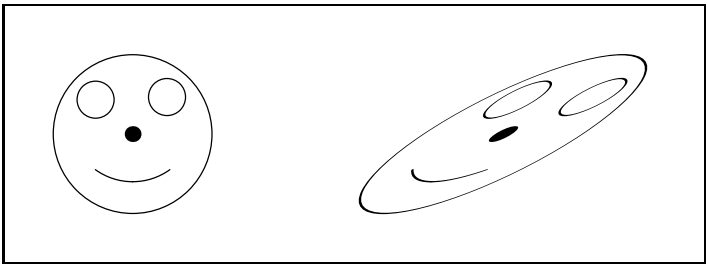
Afinní transformace

METAPOST disponuje následujícími transformacemi:

- posunutí o vektor (a, b): (x, y) `shifted` $(a, b) \longrightarrow (x + a, y + b)$,
- otočení o úhel θ, který je zadaný ve stupních, kolem středu $(0, 0)$ proti směru hodinových ručiček: (x, y) `rotated` $\theta \longrightarrow (x \cos\theta - y \sin\theta, x \sin\theta + y \cos\theta)$,
- zkosení s koeficientem a: (x, y) `slanted` $a \longrightarrow (x + ay, y)$,
- stejnolehlost s koeficientem a: (x, y) `scaled` $a \longrightarrow (ax, ay)$,
- změna měřítka na ose x: (x, y) `xscaled` $a \longrightarrow (ax, y)$,
- změna měřítka na ose y: (x, y) `yscaled` $a \longrightarrow (x, ay)$,
- rotace a stejnolehlost: (x, y) `zscaled` $(a, b) \longrightarrow (ax - by, bx + ay)$ (což odpovídá násobení komplexních čísel),
- osová souměrnost podle přímky procházející body a a b: (x, y) `reflectedabout` (a, b),
- otočení o úhel θ, který je zadaný ve stupních, kolem středu (a, b) proti směru hodinových ručiček: (x, y) `rotatedaround` $((a, b), \theta)$.

Použití transformací ukazuje obrázek 3.

```
u=.7cm;
beginfig(1);
path p[]; transform t[]; picture obrazek[];

p1 = fullcircle scaled 3u;
p2 = subpath (5.2,6) of p1;

t1 = identity scaled .8 shifted (0,.3u);
```



Obrázek 3: Použití afinních transformací

```
p3 = p2 transformed t1;

p4 = fullcircle scaled .7u shifted (.65u,.7u);
t2 = identity rotated 90;
p5 = p4 transformed t2;

t3 = identity reflectedabout ((0,-2u),(0,2u));
p6 = p3 transformed t3;

pickup pencircle scaled .3u;
draw (0,0);
pickup defaultpen;

draw p1; draw p3; draw p4; draw p5; draw p6;

obrazek1:=currentpicture;
t4 = identity slanted 1.5 shifted (7u,0);
obrazek2 = obrazek1 transformed t4;
draw obrazek2;
endfig; end
```

Chceme-li k zadané transformaci t provést transformaci inverzní, použijeme příkaz:

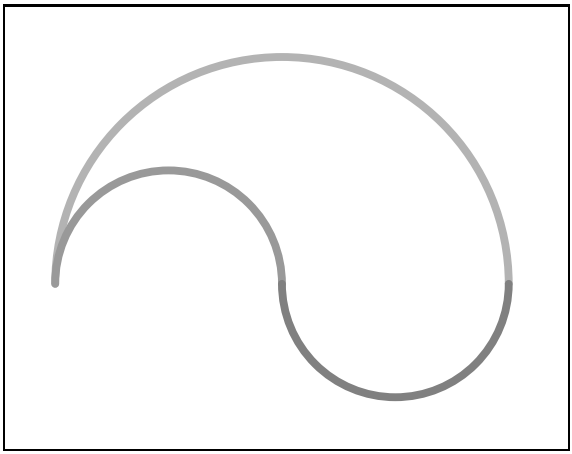
$p = q$ `transformed inverse` t .

Kružnice

Pro kreslení kružnice je předdefinována cesta `fullcircle`, která popisuje kružnici o jednotkovém průměru a středu v počátku. Dále nám METAPOST nabízí cestu `halfcircle`, což je část kružnice se středem v počátku a jednotkovým průměrem nad osou x, a `quartercircle` odpovídající části kružnice o jednotkovém průměru se středem v počátku ležící v prvním kvadrantu.

Kružnice má parametrickou délku 8.

```
u:=cm;
path p[];

p1 = halfcircle scaled 6u;
p2 = halfcircle scaled 3u shifted (-1.5u,0);
p3 = p2 rotated 180;

beginfig(1);

pickup pencircle scaled 3pt;
draw p1 withcolor 0.7white;
draw p2 withcolor 0.6white;
draw p3 withcolor 0.5white;

endfig;
end
```



Obrázek 4: Obrázek odpovídající předchozímu zdrojovému kódu

Přerušované čáry, krajní body čar a šipky

METAPOST poskytuje kromě plné čáry i čáry přerušované, a to tečkované a čárkované. Pro jejich vykreslení používáme příkaz

`draw` *cesta* `dashed` *vzorek* ,

kde *vzorek* je typu `picture`. Předdefinovány jsou *vzorky* `evenly` a `withdots`.

První z nich vykresluje čárkovanou čáru tvořenou čárkami délky 3 pt a mezerami o velikosti 3 pt, druhý kreslí tečkovanou čáru s tečkami vzdálenými od sebe 5 pt.

Vzhled přerušované čáry si můžeme přizpůsobit svému přání jedním z následujících způsobů:

- prodloužením čárek nebo mezer mezi tečkami – transformace `scaled`
- posunutím čárek, respektive teček – transformace `shifted`
- nadefinováním mezer a délky čárek – příkaz `dashpattern(`*zápis* `on|off)`

Vnitřní proměnné `linecap`, ovlivňující tvar čáry v jejích krajních bodech, můžeme přiřadit jedno ze tří možných nastavení – `rounded` pro zaoblení čáry v krajních bodech, `butt` pro „rovné“ zakončení čáry a `squared`. V případě `butt` je čára dlouhá přesně tak, jak je zadána, v případě `rounded` a `squared` se délka čáry prodlouží na obou koncích o tloušťku pera.

Jiná vnitřní proměnná, `linejoin`, ovlivňuje rohy při změně směru lomené čáry. Může nabývat hodnot `rounded`, `beveled`, `mitered`.

Chceme-li čáru zakončit šipkou, použijeme příkaz

`drawarrow` *cesta*

na místě příkazu `draw` *cesta*. Bude vykreslena zadaná cesta se šipkou v posledním bodu křivky. Pro šipku na začátku křivky využijeme příkaz

`drawarrow reverse` *cesta*

a šipky na obou koncích křivky vykreslíme pomocí příkazu

`drawdblarrow` *cesta* .

Velikost šipky ovlivňuje hodnota vnitřní proměnné `ahlength`; úhel, který svírá šipka s křivkou, vyjadřuje proměnná `ahangle`.

Podrobnosti a ukázky všech předešlých proměnných a příkazů v odstavci najdeme v manuálu k METAPOSTu ([2, str. 32]).

1.4. Vkládání textu a obrázků

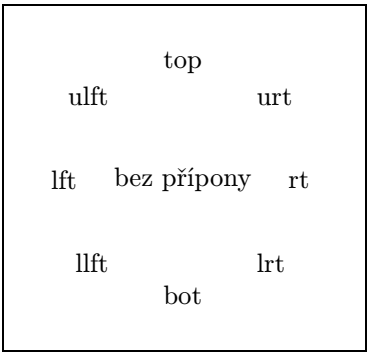
Nejjednodušší způsob, jak v METAPOSTu vložit do obrázku text nebo obrázek, je použitím příkazu

`label.`*přípona_pro_umístění*`(`*řetězec_nebo_obrázek*`,`*souřadnice*`)` ,

kde *řetězec* představuje text v uvozovkách. Následující obrázek ukazuje polohu textu nebo obrázku odpovídající jednotlivým *příponám_pro_umístění*.

Kód tohoto obrázku vypadá takto (je uveden bez `beginfig`, `endfig` a `end`):

```
defaultfont:="csr10"; u:=.7cm; labeloffset:=2u; z1=(0,0);
verbatimtex \font\muj=csr10 \muj etex;
label(btex bez přípony etex,z1);
label.rt("rt",z1);              label.urt("urt",z1);
label.lft("lft",z1);            label.ulft("ulft",z1);
label.top("top",z1);            label.llft("llft",z1);
label.bot("bot",z1);            label.lrt("lrt",z1);
```



Obrázek 5: Přípony používané pro umisťování textu nebo obrázku

Vnitřní proměnná `labeloffset` určuje vzdálenost vkládaného textu od bodu `z1`. Přednastavená hodnota této proměnné je 3bp.

Nahradíme-li `label` příkazem `dotlabel`, umístí se na zadané souřadnici tečka.

Další způsob, jak umístit text nebo obrázek, je pomocí příkazu `thelabel`, který má obdobnou syntaxi jako `label` a `dotlabel`. Výstupem příkazu `thelabel` je datový typ `picture`, což znamená, že `thelabel` přímo nevykreslí požadovaný text nebo obrázek.

Příkaz

```
label.bot("M",(0,0));
```

je tedy ekvivalentní příkazu

```
draw thelabel.bot("M",(0,0));
```

a také příkazu

```
picture a;
a = thelabel.bot("M",(0,0));
draw a;
```

Font textu můžeme změnit přiřazením

`defaultfont := "`*název_fontu*`"` .

Přednastavený je font `cmr10`. Pro kreslení našich obrázků zvolíme font `csr10`. V Linuxu pak můžeme použít české znaky, neboť používá systémové kódování ISO-8859-2, jež se shoduje s kódováním fontu `csr10`; v jiných operačních systémech (Windows, OS/2) jsou některé znaky s diakritikou chybné, protože systémové kódování se neshoduje s kódováním fontu `csr10`.

Další nevýhodou je, že tento font neobsahuje znak `space`, takže v řetězci nemohou být mezery. V tom případě jsou vhodnější například fonty `rphvr` (Helvetica) nebo `rptmr` (Times Roman), ale ty zase nemají všechny české znaky. Pokud potřebujeme obojí, musíme mít buď vhodný český postscriptový font

včetně TEXovské metriky (například `phvr8z` či `ptmr8z`), nebo raději použijeme okolí `btex ... etex` (viz dále).

Vnitřní proměnné `defaultscale` je přiřazen koeficient stejnolehlosti velikosti fontu. Předdefinovaná hodnota `defaultscale` je 1, což většinou odpovídá velikosti 10 bodů. Jestliže neznáme základní velikost fontu a chceme-li zvětšit písmo například na 12pt, použijeme přiřazení:

```
defaultscale := 12pt/fontsize defaultfont;
```

Text v METAPOSTu

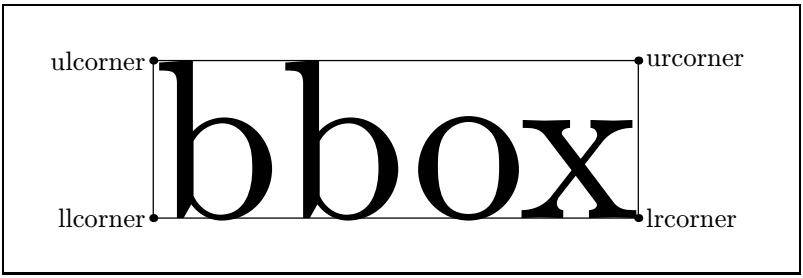
Pro vkládání náročnějšího textu můžeme využít příkazy TEXu. Použijeme-li ve vstupním souboru METAPOSTu

`btex` *příkazy_TEXu* `etex` ,

jsou *příkazy_TEXu* vysázeny TEXem a výsledek je předán zpět METAPOSTu jako datový typ `picture`, což nám umožňuje text otáčet a jinak přizpůsobovat našim požadavkům.

Překladač `mpost` v případě výskytu okolí `btex ... etex` ve vstupním souboru hledá soubor `makempx`. Adresáře, ve kterých `mpost` tento soubor hledá, jsou závislé na operačním systému. Například v operačním systému Linux `mpost` při hledání konzultuje obsah proměnné `PATH`. Pokud je v této proměnné i tečka (`.`), prohledává i aktuální adresář. V ostatních operačních systémech to může být jinak. Standardní postup popsaný v `makempx`, jež je součástí instalace METAPOSTu, je zhruba následující:

- Nejprve je zkontrolováno, zda-li jsme po posledním překladu vstupní soubor `obrázek.mp` měnili – test programem `NEWER`. V případě, že ano, pokračuje se následujícími kroky; v opačném případě nejsou tyto kroky provedeny.
- Okolí `btex ... etex` jsou uložena do souboru, který se v operačním systému Linux jmenuje `mpx$$.tex` (`$$` je číslo procesu), v operačních systémech OS/2 a Windows je tento soubor nazván `mpx_tmp.tex` – program `MPTOTEX`. V Linuxu je navíc možnost přidání nějaké hlavičky (proměnná `$MPTEXPRE`).
- Na vytvořený soubor je spuštěn TEX s parametry, které jsou v `makempx` předdefinovány a my si je zde můžeme změnit. Také je výhodné připsat příkaz, který způsobí výpis případných chybových hlášení přímo při překladu na obrazovku. Takto jsme získali soubor `mpx$$.dvi` (nebo `mpx_tmp.dvi`).
- Ze souboru `mpx$$.dvi` (`mpx_tmp.dvi`) je vytvořen soubor `obrázek.mpx`, který obsahuje kód v METAPOSTu, to jest popis obrázků – program `DVITOMP`. Na konci každého obrázku je příkaz `mpxbreak`. Nyní, vrátíme-li se k prvnímu bodu, můžeme říci, že je testováno, je-li soubor `obrázek.mpx` starší než `obrázek.mp`.
- Nakonec jsou smazány všechny dočasně vytvořené soubory.



Obrázek 6: Bounding box

Popisované kroky tedy lze ovlivňovat modifikací souboru `makempx` (v operačních systémech jiných než Linux s příponou označující spustitelný soubor).

Zjednodušeně si můžeme tento postup představit tak, že `mpost` si vybere všechna okolí `btex` ... `etex` do nějakého souboru, který si zpracuje do podoby pro něj přijatelné. A při svém druhém průchodu už má nachystáno vše potřebné, aby dokázal popisky vykreslit.

Do vstupního souboru METAPOSTu můžeme dále vložit různé definice a pomocné příkazy TEXu uzavřené v okolí `verbatimtex` ... `etex`. Hlavním rozdílem je, že `btex` vytváří obrázek, zatímco `verbatimtex` pouze vkládá příkazy TEXu.

Do okolí `verbatimtex` ... `etex` smíme zapsat např. celou hlavičku kódu LATEXu až po `\begin{document}` (včetně). Pokud v `makempx` specifikujeme použití LATEXu místo přednastavených parametrů TEXu, budeme text v okolí `btex` ... `etex` zapisovat v kódu LATEXu.

Měření textu

Předdefinované makro

`bbox` *proměnná* ,

kde *proměnná* je typu `picture`, `path` nebo `pen`, reprezentuje bounding box obrázku (respektive cesty nebo pera). Bounding boxem obrázku rozumíme nejmenší obdélník obsahující tento obrázek. Je-li `p` proměnná typu `picture`, je příkaz

```
draw bbox p;
```

ekvivalentní příkazu

```
draw llcorner p-(bboxmargin,bboxmargin)
  -- lrcorner p+(bboxmargin,-bboxmargin)
  -- urcorner p+(bboxmargin,bboxmargin)
  -- ulcorner p+(-bboxmargin,bboxmargin)
  -- cycle;
```

Vnitřní proměnná `bboxmargin` má přednastavenu hodnotu 2 bp.

1.5. Výplně

Vyplňovat smíme v METAPOSTu pouze uzavřené křivky, a to křivky uzavřené příkazem `..cycle` nebo `--cycle`. Syntaxe příkazu pro vybarvení je:

`fill` *cesta* `withcolor` *barva* .

Pokud neudáme žádnou barvu, METAPOST vybarví plochu černě.

Chceme-li plochu naopak „odbarvit“, použijeme příkaz `unfill`, který je definován jako `fill` *cesta* `withcolor background` (proměnná `background` je v souboru `plain.mp` předdefinována jako bílá barva).

Kombinací vyplňování a kreslení je příkaz

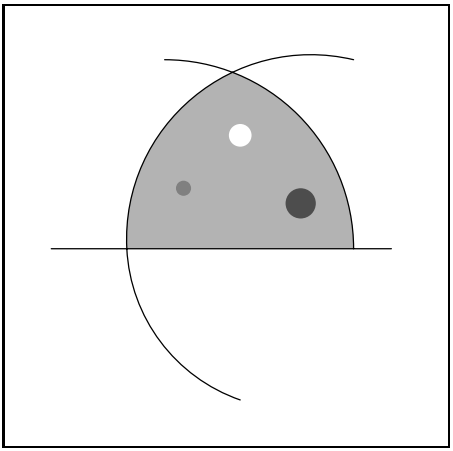
`filldraw` *cesta* `withcolor` *barva* ,

respektive příkaz `unfilldraw` *cesta* pro odbarvování.

Příkaz

`buildcycle` (p_1, p_2, ... , p_k)

vybere průsečíky křivek p_k, p_1; p_1, p_2; ... ; p_{k-1}, p_k a vytvoří uzavřenou cestu reprezentující hranici oblasti ležící „mezi“ všemi zadanými křivkami. Průsečík křivek p_i a p_{i+1} je vyhledáván tak, aby byl co nejpozdějším na p_i a co nejdřívějším na p_{i+1}.



Obrázek 7: Ukázka vyplňování pomocí příkazu `buildcycle`

Zdrojový kód k obrázku 7:

```
u=cm;          path p[];

p1 = quartercircle scaled 5u;
p2 = (-1.5u,0) -- (3u,0);
p3 = (u,-2u) .. (-.5u,0) .. (2.5u,2.5u);
```

```
p4 = buildcycle (p1,p2,p3);

p5 = fullcircle scaled .3u shifted (u,1.5u);
p6 = fullcircle scaled .2u shifted (.25u,.8u);
p7 = fullcircle scaled .4u shifted (1.8u,.6u);

fill p4 withcolor .7white;
draw p1; draw p2; draw p3;
unfill p5;
background:=.5white;%%% rovnocennými příkazy jsou:
unfill p6;          %%% fill p6 withcolor .5 white;
background:=.3white;
unfill p7;          %%% fill p7 withcolor .3 white;
```

1.6. Makra

Některé příkazy zmíněné v tomto textu nejsou přímo „vestavěny“ v METAPOSTu („vestavěné“ příkazy nazýváme primitivní). METAPOST automaticky využívá soubor `plain.mp`, ve kterém jsou uloženy definice mnoha příkazů ulehčujících nám kreslení. Cílem tohoto odstavce je objasnění zápisu podobných maker.

Makro

`def` *posloupnost_tokenů* = *nahrazující_text* `enddef`;

umožňuje použití *posloupnosti_tokenů* místo *nahrazujícího_textu*.

Makro s parametry

```
def kruz (expr s, r) =
   draw fullcircle scaled r shifted s;
enddef;
```

se liší pouze tím, že ukazuje, kde se mají v těle definice použít argumenty s a r, přičemž zde to mohou být libovolné výrazy.

Příkaz `def` lze nahradit příkazem `vardef`. Toto makro má podobnou syntaxi jako makra definovaná pomocí `def`, jen v úvodu na místě `def` *posloupnost_tokenů* stojí `vardef` *všeobecná_proměnná*, kde *všeobecná_proměnná* je jméno proměnné, jehož číselná část je nahrazena obecným symbolem pole `[]`. Jméno následující po `vardef` je tímto nadefinováno stejně jako při deklaraci proměnné. Hlavní rozdíl mezi makrem definovaným `def` a makrem definovaným `vardef` je ten, že makro definované `vardef` má automaticky vloženou skupinu `begingroup` ... `endgroup` na začátku a na konci *nahrazujícího_textu* (viz níže odstavec Seskupování). Podrobnější vysvětlení práce s makry definovanými `vardef` a dalšími typy definic můžeme najít v manuálu k METAPOSTu ([2, str. 49]).

V makrech METAPOSTu můžeme využít lokálních proměnných, cyklů i podmíněných příkazů.

Seskupování

Skupinou nazýváme posloupnost příkazů, oddělených středníky, ohraničenou příkazy `begingroup` ... `endgroup`.

Toho, aby byla proměnná, například proměnná m, lokální, dosáhneme příkazem `save` m. Všechny proměnné, jejichž název začíná m (například tedy m, $m.m$, $m.7$), se tímto stanou proměnnými typu `numeric` a jejich dřívější hodnoty jsou „zapomenuty“ až do uzavření skupiny. Použití `save` p mimo skupinu způsobí úplné zničení předcházejích hodnot proměnné. Takto se tedy budou měnit hodnoty proměnné p:

```
p=7;                    % - hodnota p je 7
begingroup;             % - začátek skupiny
show p;                 % - hodnota p >>7
save p;                 % - uschování hodnoty p
show p;                 % - hodnota p >>p
endgroup;               % - konec skupiny
show p;                 % - hodnota p >>7
save p;                 % - uschování hodnoty p;
                        %   úplné zničení hodnoty p
show p;                 % - hodnota p >>p
```

Důležité pro nás je, že příkaz `save` umožňuje použití stejně pojmenovaných proměnných uvnitř skupiny i mimo ni a opakované volání jednoho makra.

Složitější makra

Makro vykreslující kružnici o daném poloměru r a středu s můžeme zapsat například takto:

```
def k~(expr r, s) =
   draw fullcircle scaled (2*r) shifted s;
enddef;
```

Chceme-li nakreslit kružnici o poloměru 3 a středu $[3, 4]$, použijeme nadefinované makro pomocí příkazu `k(3u,(3u,4u))`.

V makrech lze využít podmíněný příkaz se syntaxí

`if` *podmínka*: *posloupnost příkazů* `else`: *posloupnost příkazů* `fi` .

Vkládáme-li podmíněný příkaz do jiného podmíněného příkazu, použijeme zkrácený zápis

`if` *1. podmínka*: ... `elseif` *2. podmínka*: ... `else`: ... `fi` .

Zvláště ve složitějších makrech oceníme skutečnost, že *posloupnost příkazů* nemusí být ukončeným příkazem. Například:

```
def kruz (expr r, s) =
   draw fullcircle scaled
       if r > 5cm: r else: (2*r) fi
```

```
   shifted s;
enddef;
```

Parametry

V předcházejících makrech jsme používali pouze parametry typu `expr`, což mohly být výrazy libovolného datového typu. Kromě `expr` jsou k dispozici parametry typu `suffix` a `text`, které deklarují libovolné jméno nebo libovolnou posloupnost tokenů. Jako ukázku makra s parametrem typu `text` vytvoříme makro `ramecek`.

```
def ramecek (text t) =
   draw t;
   draw bbox t;
enddef;
```

vykreslující t a rámeček kolem t.

Přidáme parametr typu `expr` pro barvu rámečku:

```
def ramecek (text t) (expr barva) =
   draw t;
   draw bbox t withcolor barva;
enddef;
```

Makro zavoláme například příkazem
`ramecek(fullcircle scaled 5u)(green)`.

Makro `incr` využívá parametr typu `suffix` :

```
def incr (suffix $) =
   $:=$+1;
enddef;
```

Toto makro zvýší hodnotu numerické proměnné o jedničku. Všimněme si, že parametr typu `suffix` se vyskytuje na levé i pravé straně přiřazovacího příkazu (`:=`), což s jiným typem parametru nelze.

Makro, ve kterém figurují parametry typu `suffix` a `expr` společně, například
`def moje (suffix a) (expr b) = label(str a,b); enddef;`
voláme `moje(ahoj,(3cm,3cm))` nebo `moje(ahoj)((3cm,3cm))`. Příkaz `str` převede parametr typu `suffix` na řetězec.

Cykly

Při kreslení obrázků či vytváření maker určitě využijeme cyklus `for`. Můžeme rozlišit čtyři základní druhy cyklu `for`. První z nich zapisujeme

`for` *posloupnost_tokenů* = *výraz*$_1$, *výraz*$_2$, ... , *výraz*$_n$:
posloupnost_příkazů
`endfor` .

Cyklus provede *posloupnost_příkazů* postupně s hodnotami *posloupnosti_tokenů* rovnajícími se *výrazu*$_k$, kde $k = 1, 2, \ldots, n$. *Výraz*$_k$, $k = 1, 2, \ldots, n$ nemusí mít v danou chvíli přiřazenu konkrétní hodnotu.

Obecná syntaxe druhého a nejobecnějšího cyklu typu `for` je:
`for` *posloupnost_tokenů* = *výraz*$_1$ `step` *krok* `until` *výraz*$_2$:
posloupnost_příkazů
`endfor` .

Cyklus nejprve vyhodnotí, zda jsou *výrazu*$_1$, *výrazu*$_2$ a *kroku* přiřazeny nějaké konkrétní numerické hodnoty. Dále postupuje takto:
Je-li *krok* > 0 a *výraz*$_1$ $>$ *výraz*$_2$, nebo *krok* < 0 a *výraz*$_1$ $<$ *výraz*$_2$, *posloupnost_příkazů* není provedena. V případě, že ani jedna z předcházejících podmínek splněna není, je provedena *posloupnost_příkazů* pro hodnotu *výrazu*$_1$. *Výraz*$_1$ je nahrazen hodnotou (*výraz*$_1$+*krok*). Tento proces se opakuje s novou hodnotou *výrazu*$_1$.

Místo `step` 1 `until` je možné použít předdefinovaného příkazu `upto`. Příkaz `step` -1 `until` může být nahrazen příkazem `downto`.

Pro hodnoty *kroků* je doporučeno používat celá čísla nebo přesné násobky hodnoty zlomku $\frac{1}{65536}$. Jinak nemusí *posloupnost_tokenů* vůbec dosáhnout požadované konečné hodnoty. Například hodnoty i v průběhu cyklu

```
for i=0 step 0.1 until 1: show i; endfor;
```

jsou tyto:

```
> 0
> 0.1
> 0.20001
> 0.30002
> 0.40002
> 0.50003
> 0.60004
> 0.70004
> 0.80005
> 0.90005
```

Abychom se vyvarovali podobných případů, použijeme *krok*, který je celým číslem, a vhodně upravíme *posloupnost_příkazů* vynásobením či vydělením odpovídající proměnné k získání požadovaných hodnot. Například pro

```
for i=0 upto 10: show i/10; endfor;
```

dostaneme již výstup odpovídající naší představě:

```
> 0
> 0.1
> 0.2
> 0.3
> 0.4
> 0.5
> 0.6
> 0.7
> 0.8
```

```
> 0.9
> 1
```

Podobně jako u podmíněných příkazů smí být *posloupnost_příkazů* nejen jeden příkaz nebo více příkazů oddělených středníky, ale i příkaz neukončený, například:

```
draw for a=(0,0),(u,7u),(3u,5u): a~-- endfor cycle;
```

Tento příkaz je ekvivalentní příkazu:

```
draw (0,0)--(u,7u)--(3u,5u)--cycle;
```

Index i v cyklu `for` odpovídá parametru typu `expr`, může nabývat jakékoli hodnoty, ale není to proměnná, a tedy nemůžeme její hodnotu změnit přiřazovacím příkazem. To nám umožňuje cyklus `forsuffixes`, třetí druh cyklu `for`, jehož index se chová jako parametr typu `suffix`. Syntaxi a podrobnosti použití cyklu `forsuffixes` lze najít v manuálu k METAPOSTu ([2, str. 53]).

Cyklus

`forever`: *posloupnost_příkazů* `endfor`

provádí nekonečnou smyčku a je posledním z cyklů `for`.

Pro ukončení cyklu v okamžiku, kdy booleovská proměnná dosáhne pravdivé hodnoty, se používá příkaz

`exitif` *booleovská_proměnná* .

Potřebujeme-li ukončit cyklus po dosažení hodnoty nepravdivé, využijeme příkazu

`exitunless` *booleovská_proměnná* .

Propojením nekonečné smyčky a `exitunless` vznikne obdoba cyklu while známého z jiných programovacích jazyků:

`forever`: `exitunless` *booleovská_proměnná*; *posloupnost_příkazů*
`endfor` .

Užitečné makro

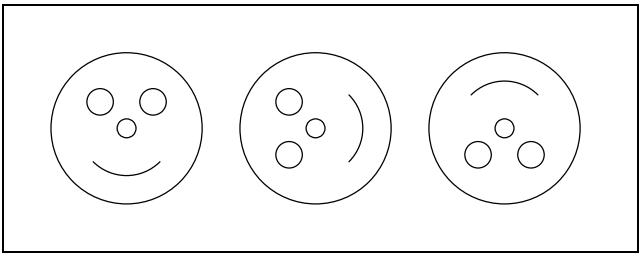
Pěkným a užitečným makrem je makro `image`, které najdeme v již zmíněném souboru `plain.mp`:

```
vardef image(text t) =
  save currentpicture;
  picture currentpicture;
  currentpicture := nullpicture;
  t;
  currentpicture
enddef;
```

Novým prvkem, který toto makro přináší, je předposlední řádek. Je-li před `enddef` proměnná, do níž je v makru něco přiřazováno, při volání makra použijeme:

proměnná = `název_makra` (*parametry makra*) ,

přičemž *proměnná* musí být stejného typu jako proměnná v makru na zmíněném řádku. Potom máme v *proměnné* uložen výsledek makra a můžeme ho vykreslit, transformovat a podobně.



Obrázek 8: Obrázek odpovídající následujícímu zdrojovému textu

```
u:=.5cm;

beginfig(1);

picture obrazek[];
path a[];
transform t[];
a1 = fullcircle scaled 4u;
a2 = fullcircle scaled 0.5u;
a3 = fullcircle scaled 0.7u shifted (0.7u,0.7u);
a4 = fullcircle scaled 0.7u shifted (-0.7u,0.7u);
a5 = subpath (5,7) of fullcircle scaled 2.5u;

obrazek1 = image(
                for i=1 upto 5:
                    draw a[i];
                endfor;
);

t1 = identity rotated 90 shifted (5u,0);
t2 = identity rotated 180 shifted (10u,0);
obrazek2 = obrazek1 transformed t1;
obrazek3 = obrazek1 transformed t2;
for i=1 upto 3:
    draw obrazek[i];
endfor;
```

```
endfig;
end
```

1.7. Prologues

Prohlížíme-li přímé výstupy METAPOSTu, je vhodné v úvodu každého vstupního souboru přiřadit speciální proměnné `prologues` kladnou hodnotu. V souboru `obrazek.mp` na straně 43 bylo použito `prologues:=1`. Tento příkaz způsobí, že v hlavičce souboru `obrazek.1` (a samozřejmě i v hlavičce souboru `obrazek.3`) se objeví

```
%!PS-Adobe-3.0 EPSF-3.0
```

Význam tohoto řádku je zhruba následující: vznikne soubor ve formátu EPS (Encapsulated PostScript), tzv. zapouzdřený PostScript. Formát EPS byl vytvořen kvůli možnosti vkládání těchto obrázků do jiných dokumentů.

Vkládáme-li do souboru text (obyčejný text, bez nějakých zvláštních znaků, například bez znaků matematických), to jest používáme-li nějaký font, je proměnná `prologues` opět důležitá. Při jejím správném nastavení postscriptový soubor obsahuje:

```
%%DocumentFonts: CMR10
 /cmr10 /CMR10 def
 /fshow {exch findfont exch scalefont setfont show}bind def
```

Toto by mělo způsobit vyhledání námi požadovaného fontu. Ovšem při prohlížení pomocí programů, které používají k vyrastrování obrázků `Ghostscript` (například Ghostview), je font `cmr10` nahrazen fontem `Courier` díky standardnímu obsahu souboru `Fontmap` (pokud jej na příslušném místě neupravíme).

Problémy nastanou u složitějšího textu, neboť mapa fontů se může lišit a výsledek nemusí odpovídat naší představě. V takovém případě je lepší obrázek vložit do TEXovského dokumentu a soubor `.dvi` vzniklý z tohoto dokumentu zpracovat programem `dvips` (přitom proměnnou `prologues` nenastavujeme). Řešením této situace je také využití programu `dvips` s volbou `j0` (spouštíme `dvips -j0`) při vytváření postscriptového souboru (nezáleží na tom, zda je proměnná `prologues` nastavena, či nikoli).

Přednastavená hodnota `prologues` je rovna 0.

Pro nás znamená příkaz `prologues:=` c, $c \leqq 0$ to, že obrázek není oříznut (při c>0 je oříznut na nejmenší možný obdélník), což nám při vkládání do textu nevadí, a při pokusu o prohlédnutí obrázku (myšleno výsledku překladu METAPOSTu) s popiskem dostáváme chybové hlášení o nenalezení fontu.

1.8. METAFONT a METAPOST

Hlavním rozdílem mezi METAFONTem a METAPOSTem je jejich rozdílný výstup. Výsledkem práce METAFONTu je bitová mapa a metrika, zatímco výstupem METAPOSTu je program v jazyce PostScript.

V manuálu k METAPOSTu ([2, str. 79]) jsou vypsány příkazy a proměnné nacházející se pouze v METAPOSTu a také ty, které najdeme jen v METAFONTu. Pro METAPOST jsou jedinečné příkazy týkající se barev, vkládání textu, oblouková délky křivky; METAFONT zase poskytuje proměnné a příkazy pro práci s písmeny (ligatury, kerningové páry), pixely a podobně.

Literatura

[1] Adobe Systems Incorporated *PostScript Language Reference Manual*. Massachusetts: Addison-Wesley, 3. vydání, 1999. ISBN 0-201-37922-8 http://adobe.com:80/products/postscript/pdfs/PLRM.pdf

[2] Hobby, J. D. *A User's Manual for MetaPost*. Součást dokumentace programu METAPOST. http://cm.bell-labs.com/cm/cs/cstr/162.ps.gz

[3] Knuth, D. E. *The METAFONTbook*. Massachusetts: Addison-Wesley, 1986. ISBN 0-201-13445-4

[4] Kuben, J. *Diferenciální počet funkcí jedné proměnné*. Brno: VA, 1999

[5] Kuben, J. *Zpravodaj Československého sdružení uživatelů TEXu 2/94*. Brno. http://bulletin.cstug.cz/pdf/bul942.pdf

[6] Olšák, P. *TEXbook naruby*. Brno: Konvoj, 2. vydání, 2001. ISBN 80-7302-007-6 ftp://math.feld.cvut.cz/pub/olsak/tbn/tbn.pdf

[7] Polák, J. *Přehled středoškolské matematiky*. Praha: Prometheus, 2000. ISBN 80-7196-196-5

[8] Rybička, J. *LATEX pro začátečníky*. Brno: Konvoj, 2. vydání, 1999. ISBN 80-85615-74-6

Zajímavé odkazy týkající se tématu:

- http://cm.bell-labs.com/who/hobby/MetaPost.html
- http://comp.uark.edu/~luecking/tex/mfpic.html
- http://ftp.cstug.cz/pub/tex/CTAN/graphics/mfpic/ (bohužel zatím je zde pouze starší verze)
- http://ftp.cstug.cz/pub/tex/CTAN/systems/knuth/mf/mfbook.tex

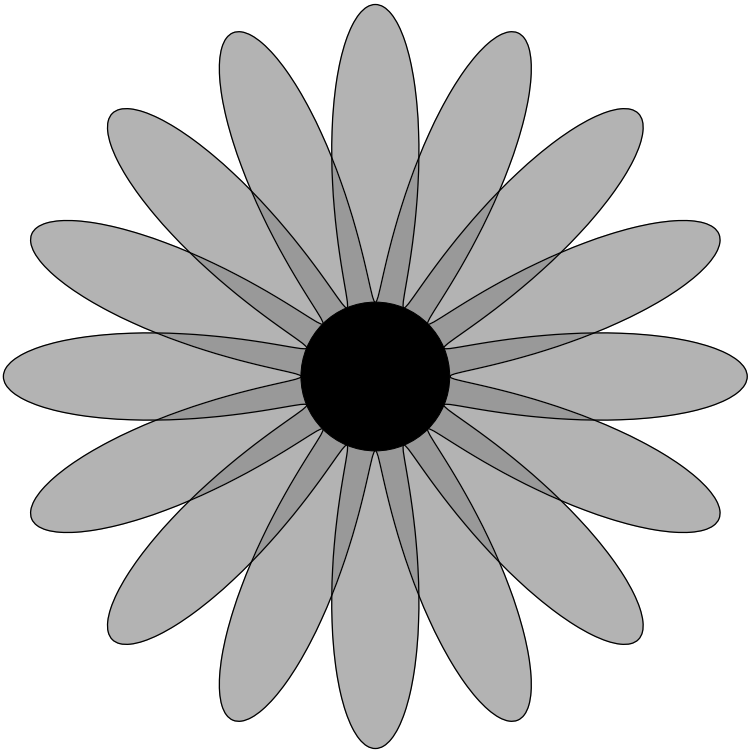
Rejstřík

Summary: METAPOST and mfpic—the first part

This article is dedicated to introduction of the language used by METAPOST system by John D. Hobby. All commands of the language are shown in context and are illustrated by simple examples.

METAPOST a mfpic – druhá část

Miroslava Krátká



Obsah

1. Balíky maker pro METAPOST

První část práce ukazuje nemalé možnosti METAPOSTu při kreslení obrázků. Chceme-li ovšem nakreslit náročnější obrázek, nestačí jen prolistovat manuálem; musíme se naučit pracovat s novým programovacím jazykem. To mnohé, zejména ty méně zvídavé, odradí. A právě pro ně se objevují balíky maker pro METAPOST.

Takovými balíky jsou `mfpic`, kterým se v dalším textu budeme podrobněji zabývat (`http://comp.uark.edu/~luecking/tex/mfpic.html`), a `feynmf`, jehož autorem je Thorsten Ohl (`ftp://ftp.cstug.cz/pub/CTAN/macros/latex/contrib/supported/feynmf`). V roce 1989 byl vytvořen soubor maker `feynman.mf` pro kreslení Feynmanových diagramů. O pět let později, tedy v roce 1994, se autor inspiroval balíkem `mfpic` (zdrojový kód se zapisoval do zdrojového souboru LaTeXu). Vzniklý `feynmf` využíval pro kreslení obrázků buď METAFONT, nebo METAPOST.

Zcela odlišná je grafická nadstavba pro METAPOST nazvaná `metagraf`. Jde o aplikaci napsanou v jazyce Java (je třeba Java2). Kreslí se pomocí menu a myši (`http://w3.mecanica.upm.es/metapost/metagraf.php`).

2. Historie `mfpic`

Během roku 1992 začal Tom Leathrum pracovat na tvorbě `mfpic`. Tento balík maker využívá pro kreslení METAFONT, přičemž zdrojový kód se zapisuje do vstupního souboru TeXu a užití příkazů samotného METAFONTu je pro uživatele skryto. V roce 1996 vývoj převzal Daniel H. Luecking. Od verze 0.3.0 z roku 1998 již můžeme využívat i METAPOST. Aktuální verzí k dubnu 2001 je verze 0.4.05. První zmínka o `mfpic` v češtině se objevila v roce 1994 v druhém čísle Zpravodaje CSTUGu zmíněného roku ([5, str. 87]). Autor se věnoval verzi `mfpic` 0.2, kterou upravil, a verzi 0.2.5.

3. Zdrojový soubor

Makra `mfpic` lze použít ve formátech plain, LaTeX i $\mathcal{AMS}$-TeX. Než začneme tvořit naše obrázky, potřebujeme mít nainstalované následující soubory a balíky (prvním předpokladem je samozřejmě TeX a METAPOST, respektive METAFONT):

- `grafbase.mp` a `dvipsnam.mp` v adresáři prohledávaném METAPOSTem (například aktuální adresář), respektive `grafbase.mf` v adresáři prohledávaném METAFONTem,
- `mfpic.tex` a `mfpic.sty` v adresáři, ve kterém hledá TeX (například aktuální adresář),

- `epsf.tex` pro plain TEX, $\mathcal{AMS}$-TEX a LATEX2.09,
- `supp-pdf.tex` a `supp-mis.tex` pro pdfTEX,
- `graphics` nebo `graphicx` pro LATEX 2_ε a pro pdfLATEX s výstupem do pdf; tento případ vyžaduje tři soubory, a to `pdftex.def`, `supp-pdf.tex`, `supp-mis.tex`.

Jestliže žádný z uvedených souborů a balíků nenačteme (nebo tak učiníme až po načtení balíku `mfpic`), `mfpic` to provede sám. Například pro LATEX načítá balík `graphics`. Toto vše je nezbytné pro vkládání námi vytvořených obrázků, jež je prováděno takto:

- `\epsfbox` {*jméno_souboru*} v plain TEXu, $\mathcal{AMS}$-TEXu a LATEXu 2.09,
- `\convertMPtoPDF` {*jméno_souboru*}{1}{1} v pdfTEXu,
- `\includegraphics` {*jméno_souboru*} v LATEX 2_ε a pdfLATEXu.

V dalším výkladu budeme pojmem LATEX myslet LATEX 2_ε. Zaměříme se na `mfpic` s využitím METAPOSTu.

Jak víme z předchozích odstavců, METAPOST vytváří soubory s příponou *.číslo*, a proto je v pdfLATEXu dále třeba přidat příkaz

`\DeclareGraphicsRule` {$*$} {`mps`} {$*$} {} .

Každý soubor využívající `mfpic` s METAPOSTem má v LATEXu následující strukturu:

```
\documentclass{article}
\usepackage[metapost]{mfpic} %%% pokud neuvedeme nepovinný para-
                             %%% metr, bude použit pro tvorbu
                             %%% obrázků Metafont
%\usepackage{mfpic}  %%% tyto dva řádky jsou ekvivalentní s řád-
%\usemetapost        %%% kem předcházejícím

\begin{document}

\opengraphsfile{obrazek}

%libovolný zdrojový kód

\begin{mfpic}[a][b]{c}{d}{e}{f}
%příkazy popisující obrázek - výsledkem je obrazek.1
\end{mfpic}

%okolí mfpic lze zapisovat také takto:

\mfpic[g][h]{i}{j}{k}{l}
%příkazy popisující obrázek - výsledkem je obrazek.2
\endmfpic
```

```
\closegraphsfile

\opengraphsfile{picture}

\mfpic[m][n]{o}{p}{q}{r}
%příkazy popisující obrázek - výsledkem je picture.1
\endmfpic

\closegraphsfile

\end{document}
```

Obdobný zdrojový soubor v plain TEXu:

```
\input mfpic.tex
\usemetapost

\opengraphsfile{obrazek}

\mfpic[a][b]{c}{d}{e}{f}
%příkazy popisující obrázek - výsledkem je obrazek.1
\endmfpic

\closegraphsfile

\bye
```

Příkazem

\opengraphsfile{*název_souboru*}

otevíráme soubor, do kterého se zapisují příkazy METAPOSTu, příkaz

\closegraphsfile

tento soubor zavírá (tyto příkazy by měly být používány pouze v udaném tvaru, to jest ne jako okolí \begin{*okolí*} ... \end{*okolí*} v LATEXu). Příkazy

\mfpic [*měřítko*$_{\text{osa\_x}}$] [*měřítko*$_{\text{osa\_y}}$] {$x_{\min}$} {$x_{\max}$} {$y_{\min}$} {$y_{\max}$}
...
\endmfpic

\begin{mfpic}[*měřítko*$_{\text{osa\_x}}$][*měřítko*$_{\text{osa\_y}}$]{$x_{\min}$}{$x_{\max}$}{$y_{\min}$}{$y_{\max}$}
...
\end{mfpic} (pouze v LATEXu)

uzavírají popis jednoho obrázku. Měřítko musí být zadáno alespoň na jedné ose; je-li zadáno pouze jedno, je měřítko shodné pro osu x i y. Čísla $x_{\min}$, $x_{\max}$ určují rozsah osy x, čísla $y_{\min}$, $y_{\max}$ určují rozsah osy y.

Nyní jsou dvě možnosti způsobu zpracování vstupního souboru:

Po překladu vstupního souboru LaTeXem (respektive plain TeXem) vzniknou soubory `obrazek.mp` a `picture.mp`. Tyto soubory přeložíme programem `mpost` (viz první část práce) a poté znovu spustíme LaTeX (respektive TeX). Výsledkem je soubor s příponou `.dvi`. Nejsou-li při jeho prohlížení viditelné obrázky (například `xdvi` na novějších verzích operačního systému Linux obrázky zobrazuje), vytvoříme příslušným programem (`dvips`) postscriptový soubor a prohlédneme si jej například pomocí Ghostview.

Použijeme-li pdfLaTeX (respektive pdfTeX), vznikne soubor s příponou `.pdf`, který si prohlédneme například prohlížečem Acrobat Reader.

4. Volby `mfpic`

Již v předchozím odstavci jsme se seznámili s jednou volbou balíku `mfpic`, a to `metapost`. Jak bylo vidět z ukázky, tyto volby se zapisují buďto jako nepovinné parametry u LaTeXovského příkazu `\usepackage`, nebo ekvivalentními příkazy.

Můžeme použít následující volby:

- `metapost`, `\usemetapost` – popisy obrázků vytvářených pomocí `mfpic` jsou zapisovány do souboru s příponou `.mp` a tento soubor je zpracován METAPOSTem; příkaz musí být zapsán před otevřením souboru `.mp`, to jest před příkazem `\opengraphsfile`,
- `metafont`, `\usemetafont` – popisy obrázků jsou zadávány do souboru s příponou `.mf`, tento soubor je zpracován METAFONTem; příkaz musí být zapsán před otevřením souboru `.mf`, to jest před příkazem `\opengraphsfile`; tato volba je přednastavena (chceme-li tedy používat METAFONT, není nutno volbu udávat),
- `mplabels`, `\usemplabels`, `\nomplabels` – tato volba ovlivňuje zpracování popisek u příkazu `\tlabel`, popisky jsou zpracovány METAPOSTem; příkaz musí být uveden až za příkazem `\usemetapost` ; používáme pouze s následující volbou,
- `truebbox`, `\usetruebbox`, `\notruebbox` – zajistí, aby METAPOST určil bounding box obrázku včetně textu; tuto volbu využíváme zároveň s volbou `mplabels`,
- `clip`, `\clipmfpic`, `\noclipmfpic` – odstraní části obrázku přesahující obdélník udaný rozměry u `\mfpic`; lze použít jak u METAFONTu, tak u METAPOSTu,
- `centeredcaptions`, `\usecenteredcaptions`, `\nocenteredcaptions` – zkrácené řádky textu pod obrázkem vytvořeného příkazem `\tcaption` jsou umístěny do středu, neboli vycentrovány podle šířky obrázku,
- `debug`, `\mfpicdebugtrue`, `\mfpicdebugfalse` – sdělí nám více informací o průběhu zpracování vstupního souboru; tyto informace jsou zapisovány do souboru s příponou `.log` a v některých případech na obrazovku.

Pro naši práci jsou ideální následující dvě kombinace:

```
\usepackage [metapost, mplabels, truebbox] {mfpic} ,
\usepackage [metapost, mplabels, truebbox, clip] {mfpic} .
```

Volba `mplabels` zajistí, aby popisky zpracovával METAPOST; volba `truebbox` zajistí, aby METAPOST určil přesný bounding box obrázku a tento předal TEXu (a tedy nevzniknou problémy například při obtékání obrázku); volba `clip` způsobí ořezání výsledného obrázku na rozměr námi zadaný (tudíž zamezíme zasahování okolního textu do obrázku). Všechny další kombinace různých voleb `mfpic` mohou dát zcela nesprávné výsledky.

5. Parametry `mfpic`

V makrech `mfpic` se samozřejmě objevuje mnoho parametrů, jejichž změnami můžeme dosáhnout různých, pro nás příjemných, efektů. U většiny maker, o kterých se zmiňuji v dalších odstavcích, je řečeno, jaké parametry ovlivňují jejich výsledky.

Některé parametry jsou uloženy jako dimenze TEXu; jiné jsou uloženy METAPOSTem, tedy jejich případné změny uvnitř okolí `\mfpic` ... `\endmfpic` mají pouze lokální význam.

Významné pro nás mohou být následující parametry (veškerá zde zmíněná makra jsou podrobněji popisována v dalším textu).

Parametry TEXu:

- `\mfpicunit` – představuje základní jednotku délky; přednastavena je na hodnotu 1 pt; všechny x-ové a y-ové souřadnice jsou jí v makrech `mfpic` násobeny (proměnná typu dimenze),
- `\pointsize` – uchovává průměr kružnice používané v makru `\point`, dále průměr symbolů v makrech `\plotsymbol` a `\plot`; přednastavena je hodnota 2 pt (proměnná typu dimenze),
- `\pointfilltrue`, `\pointfillfalse` – logická proměnná určující, bude-li kružnice kreslená příkazem `\point` vyplněna, či nikoli,
- `\headlen` – dimenze udávající délku šipky při použití makra `\arrow`; přednastavena je délka 3 pt,
- `\axisheadlen` – dimenze, která uchovává délku šipky při využití makra `\axes`, `\xaxis`, `\yaxis`; přednastavena je délka 5 pt,
- `\dashlen` – dimenze určující délku čárek v makru `\dashed`; přednastavena je délka 4 pt,
- `\dashspace` – dimenze udávající velikost mezer používaných v makru `\dashed`; přednastavena je mezera 4 pt,
- `\dashlineset` – makro, které nastaví standardní hodnoty pro `\dashlen` a `\dashspace`, to znamená obě hodnoty na 4 pt,
- `\dotlineset` – makro nastavující hodnotu pro `\dashlen` na 1 pt a hodnotu `\dashspace` na 2 pt,

- \symbolspace – dimenze určující velikost mezery mezi symboly v makru \plot; přednastavena je mezera 5 pt,
- \hashlen – dimenze uchovávající délku čárek používaných v makrech \xmarks a \ymarks; přednastaveny jsou délky 4 pt,
- \dotsize – dimenze určující velikost kruhů v makru, \dotted; přednastaven je průměr 0,5 pt,
- \dotspace – dimenze udávající mezery mezi středy kruhů vykreslujících se v makru \dotted; přednastavena je mezera o velikosti 3 pt,
- \polkadotspace – dimenze nastavující velikost mezer mezi středy kruhů používaných v makru \polkadot; přednastavena je mezera 10 pt,
- \hatchspace – dimenze uchovávající vzdálenost rovnoběžek používaných v makru \hatch; přednastavena je vzdálenost 3 pt,
- \mfpicheight – dimenze, v níž je uložena výška obrázku vytvořeného v okolí \mfpic ... \endmfpic,
- \mfpicwidth – dimenze, v níž je uložena šířka obrázku vytvořeného okolím \mfpic ... \endmfpic.

Poslední dvě dimenze (výška a šířka obrázku) jsou pro nás praktické v případě, že bychom chtěli „šidit“ při umisťování obrázku. Standardně jsou vypočítávány z údajů zadaných v příkazu \mfpic v úvodu každého obrázku.

Chceme-li některou z dimenzí TEXu změnit, provedeme to obvyklým způsobem, to znamená například \mfpicunit=3pt (v LATEXu můžeme také příkazem \setlength{\mfpicunit}{3pt}).

Parametry METAPOSTu (neboli TEXovská makra nastavující interní parametry METAPOSTu, jejichž přesné názvy pro nás nejsou podstatné):

- \pen {*tloušťka*}, \drawpen {*tloušťka*} – nastaví tloušťku pera pro kreslení; přednastavena je hodnota 0,5 pt,
- \hatchwd {*tloušťka*} – určuje tloušťku čar používaných při šrafování; přednastavena je hodnota 0,5 pt,
- \polkadotwd {*průměr*} – přiřazuje průměr kruhům využívaným v makru \polkadot; přednastavena je hodnota 5 pt,
- \headshape {*poměr*} {*napětí*} {*vybarvení*} – toto makro určuje tvar šipky používané v makrech \arrow, \axes, \xaxis, \yaxis; *poměr* je poměr šířky šipky k její délce, *napětí* představuje napětí Bézierovy křivky, kterou je šipka vykreslena, a *vybarvení* značí, či je šipka otevřená (hodnota false) nebo uzavřená a vybarvená; přednastaveny jsou hodnoty 1, 1, false.

6. Barvy mfpic

V METAPOSTových makrech pro mfpic, to jest v souboru grafbase.mp, se setkáváme se čtyřmi barvami – drawcolor, fillcolor, hatchcolor, headcolor.

Barvou `drawcolor` jsou vykreslovány křivky, barva `fillcolor` je využívána pro výplně, `hatchcolor` pro šrafování a `headcolor` pro kreslení šipek. Všechny tyto barvy jsou přednastaveny na černou barvu.

Nejjednodušší způsob, jak některou z výše uvedených barev změnit, je použití odpovídajícího příkazu:

\drawcolor {*barva*} ,
\fillcolor {*barva*} ,
\hatchcolor {*barva*} ,
\headcolor {*barva*} .

Parametrem *barva* může být:

- předdefinovaná barva – `black`, `white`, `red`, `green`, `blue`, `cyan`, `magenta`, `yellow` (tzn. černá, bílá, červená, zelená, modrá, tyrkysová, fialová, žlutá),
- barva vyjádřená modelem `rgb` – `rgb` (a, b, c), kde a, b, c jsou čísla z intervalu $\langle 0, 1\rangle$,
- barva vyjádřená modelem `cmyk` – `cmyk` (a, b, c, d), kde a, b, c, d jsou čísla z intervalu $\langle 0, 1\rangle$, převádí barvu udanou modelem `cmyk` na model `rgb`,
- barva vyjádřená modelem `RGB` – `RGB` (a, b, c), kde a, b, c jsou z intervalu $\langle 0, 255\rangle$, převádí barvu v modelu `RGB` na model `rgb`,
- barva vyjádřená modelem `gray` – `gray` (a), kde a je z intervalu $\langle 0, 1\rangle$, odpovídá barvě $a *$ `white`,
- barva námi předdefinovaná (model `named`) – můžeme využít barvy definované v souboru `dvipsnam.mp` nebo barvy jiné, námi nadefinované pomocí příkazu
  \mfpdefinecolor {*jméno_barvy*} {model} {barva} ,
  kde *jméno_barvy* je libovolný námi zvolený název pro barvu, *model* je jeden z modelů `rgb`, `RGB`, `cmyk`, `gray`, `named` a *barva* je vyjádření požadované barvy v použitém modelu. Použijeme-li model `named`, pouze přejmenujeme danou *barvu*.

Druhou možností změny barvy jsou příkazy:

\drawcolor [*model*] {*barva*} ,
\fillcolor [*model*] {*barva*} ,
\hatchcolor [*model*] {*barva*} ,
\headcolor [*model*] {*barva*} ,

kde *model* může být `rgb`, `cmyk`, `RGB`, `named`, nebo `gray` a *barva* je „hodnota“ barvy v příslušném modelu.

7. Jednoduché útvary

Příkaz

\pointdef {*název*} (x, y)

umožňuje definovat název pro bod a jeho souřadnice, kde *název* je námi zvolený název (neobsahující zpětné lomítko) pro bod o souřadnicích (x, y). Nadefinujeme-li například `\pointdef{M}(3,7)`, `mfpic` bude příkaz `\M` chápat jako zadání souřadnic (3,7) a `\Mx`, popř. `\My`, bude expandovat na 3, popř. 7.

Makro

`\point` [*průměr*] $\{(x_1, y_1), (x_2, y_2), \dots\}$

vykreslí kruhy se středy $(x_1, y_1), (x_2, y_2), \dots$ a průměrem odpovídajícím hodnotě proměnné `\pointsize` (přednastavené na velikost 2 pt). Nepovinný parametr *průměr* nám umožňuje přednastavenou hodnotu průměru kruhů jednoduše upravovat. Chceme-li body označovat pouze kružnicemi, využijeme příkazu `\pointfillfalse`; standardní nastavení, `\pointfilltrue`, vyplňuje kruhy barvou `fillcolor`.

Příkaz

`\grid` {*mezera_vertikálních_rovnoběžek*,
mezera_horizontálních_rovnoběžek}

vytvoří systém kruhů, které umístí na průsečíky horizontálních a vertikálních rovnoběžek. Vzdálenost těchto rovnoběžek udáme dvěma povinnými parametry, a to *mezera_vertikálních_rovnoběžek* a *mezera_horizontálních_rovnoběžek*; tyto parametry zadáváme bezrozměrné, tedy pouze čísla bez jednotky. Kruhy odpovídají interní proměnné `onedot` typu `picture`, což je vnitřek kruhu o průměru 0,5 pt.

Nejsme-li zastánci označování bodů kruhy či kružnicemi, zalíbí se nám makro

`\plotsymbol` [*průměr*] {*symbol*} $\{(x_1, y_1), (x_2, y_2), \dots\}$

vykreslující *symboly* se středy o souřadnicích $(x_1, y_1), (x_2, y_2), \dots$ K dispozici jsou tyto *symboly*:

- `Circle` – kružnice,
- `Diamond` – kosočtverec,
- `Square` – čtverec,
- `Triangle` – trojúhelník,
- `SolidCircle` – kruh,
- `SolidDiamond` – vyplněný kosočtverec,
- `SolidSquare` – vyplněný čtverec,
- `SolidTriangle` – vyplněný trojúhelník,
- `Cross` – křížek,
- `Plus` – plus,
- `Star` – hvězda.

Velikost symbolů koresponduje s hodnotou `\pointsize` (=2 pt) a lze ji upravit nepovinným parametrem *průměr*. Vyplněné symboly mají barvu odpovídající `fillcolor`.

Lomenou čáru s vrcholy v bodech $(x_1, y_1), (x_2, y_2), \dots$ nakreslíme příkazy

`\polyline` $\{(x_1, y_1), (x_2, y_2), \dots\}$,

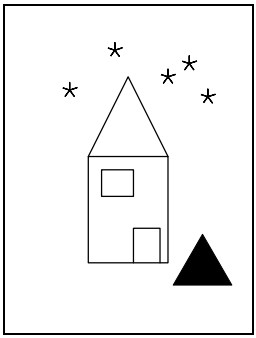
`\lines` $\{(x_1, y_1), (x_2, y_2), \dots\}$.

Makro

\polygon $\{(x_1,y_1),(x_2,y_2),\dots\}$
vykreslí uzavřený polygon s vrcholy v bodech $(x_1,y_1),(x_2,y_2),\dots$

Příkaz

\rect $\{(x_1,y_1),(x_2,y_2)\}$
vytvoří obdélník, kde $(x_1,y_1),(x_2,y_2)$ jsou protilehlé vrcholy obdélníku.



```
\mfpic[10]{0}{7}{0}{10}
\rect{(2,1.5),(5,5.5)}
\polyline{(2,5.5),(3.5,8.5),(5,5.5)}
\lines{(3.7,1.5),(3.7,2.8),
                  (4.7,2.8),(4.7,1.5)}
\polygon{(2.5,4),(2.5,5),
                     (3.7,5),(3.7,4)}
\plotsymbol[17pt]{SolidTriangle}
                              {(6.3,1.3)}
\plotsymbol[5pt]{Star}{(1.3,8),
 (3,9.5),(5,8.5),(6.5,7.75),(5.8,9)}
\endmfpic
```

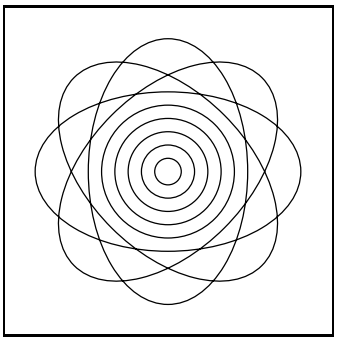
Kružnici se středem (x,y) a poloměrem r nakreslíme pomocí příkazu

\circle $\{(x,y),r\}$.

Elipsu popisuje makro

\ellipse $[\theta]$ $\{(x,y),r_{\mathrm{x},r_{\mathrm{y}}}\}$,

kde r_{x} je délka hlavní poloosy, r_{y} délka vedlejší poloosy, (x,y) je střed elipsy a nepovinný parametr θ umožňuje otočení elipsy o θ stupňů proti směru hodinových ručiček kolem jejího středu.



```
\mfpic[10]{-5}{5}{-5}{5}
\circle{(0,0),2.5}
\circle{(0,0),2}
\circle{(0,0),1.5}
\circle{(0,0),1}
\circle{(0,0),0.5}
\ellipse{(0,0),5,3}
\ellipse[45]{(0,0),5,3}
\ellipse[90]{(0,0),5,3}
\ellipse[135]{(0,0),5,3}
\endmfpic
```

Bézierovu křivku procházející body $(x_1,y_1),(x_2,y_2),\dots$ vykreslíme příkazem

\curve $[napětí]$ $\{(x_1,y_1),(x_2,y_2),\dots\}$.

Nepovinný argument *napětí* upravuje napětí křivky, o němž jsme se zmiňovali již v první části této práce. Jeho předdefinovaná hodnota je 1.

Uzavřenou Bézierovu křivku procházející body $(x_1, y_1), (x_2, y_2), \ldots$ získáme příkazem

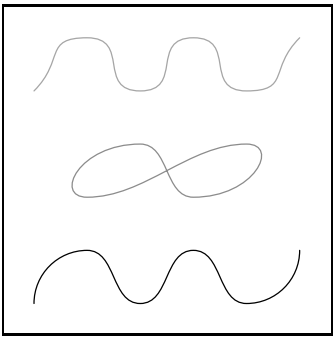
\cyclic [*napětí*] $\{(x_1, y_1), (x_2, y_2), \ldots\}$.

Nepovinný argument *napětí* využijeme stejně jako u předchozího příkazu v případě, že chceme upravit vzhled křivky.

Makro

\fcncurve [*napětí*] $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$

vytvoří křivku procházející danými body. Výsledná křivka je více symetrická než při použití \curve, příkaz je proto vhodný například pro kreslení grafů funkcí. Nepovinný argument opět ovlivňuje napětí křivky a je přednastaven na hodnotu 1,2.



```
\mfpic[20]{0}{5}{0}{5}
\curve{(0,0),(1,1),(2,0),
       (3,1),(4,0),(5,1)}
\draw[.55white]\cyclic{
       (1,2),(2,3),(3,2),
                   (4,3)}
\draw[.65white]\fcncurve{
       (0,4),(1,5),(2,4),
       (3,5),(4,4),(5,5)}
\endmfpic
```

Chceme-li nakreslit kruhový oblouk, máme k dispozici několik možností pro jeho zadání:

\arc [s] $\{(x_1, y_1), (x_2, y_2), \theta\}$

vykresluje oblouk z bodu (x_1, y_1) do bodu (x_2, y_2) tak, aby oblouk pokrýval úhel θ, který je měřen proti směru hodinových ručiček.

Oblouk procházející třemi body $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ je zadán příkazem

\arc [t] $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$.

Makro

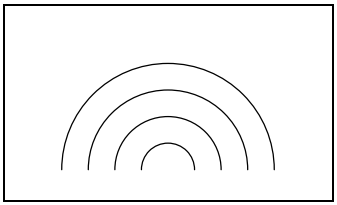
\arc [p] $\{(x_{\mathrm{s}, y_{\mathrm{s}}), \theta_1, \theta_2, r\}}$

nakreslí část kružnice se středem $(x_{\mathrm{s}, y_{\mathrm{s}})}$ a poloměrem r začínající úhlem θ_1 a končící úhlem θ_2, přičemž oba úhly jsou měřeny proti směru hodinových ručiček od kladné části souřadné osy x.

Posledním možným zadáním kruhového oblouku je příkaz

\arc [c] $\{(x_{\mathrm{s}, y_{\mathrm{s}}), (x_1, y_1), \theta\}}$,

který vykreslí oblouk se středem $(x_{\mathrm{s}, y_{\mathrm{s}}}$ začínající v bodu (x_1, y_1) a pokrývající úhel θ. Všechna předchozí zadání oblouku (s parametry [s], [t], [p]) jsou interně převáděna a vykreslena tímto makrem.



```
\mfpic[10]{-5}{5}{0}{5}
\arc[s]{(1,0),(-1,0),180}
\arc[t]{(-2,0),(0,2),(2,0)}
\arc[p]{(0,0),0,180,3}
\arc[c]{(0,0),(4,0),180}
\endmfpic
```

7.1. Souřadné osy

Pro znázornění souřadných os můžeme použít jedno z následujících maker:

\axes [*délka_šipky*] ,
\xaxis [*délka_šipky*] ,
\yaxis [*délka_šipky*] .

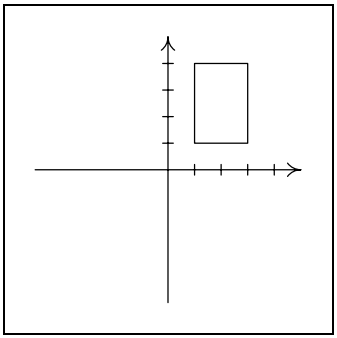
První z maker vyznačuje obě souřadné osy, zbývající dvě makra pouze souřadnou osu x, respektive y. Nevyužijeme-li nepovinný parametr, budou na osách vykresleny šipky o délce přiřazené proměnné \axisheadlen (přednastavena je hodnota 5 pt), tvaru určeného makrem \arrow a barvě odpovídající headcolor. Značky na osách získáme příkazem

\xmarks {*seznam_bodů_na_ose_x*}

pro osu x a příkazem

\ymarks {*seznam_bodů_na_ose_y*}

pro osu y. Délka čárečky odpovídá hodnotě \hashlen, nastavena je na 4 pt.



```
\mfpic[10]{-5}{5}{-5}{5}
\axes
\xmarks{1,2,3,4}
\ymarks{1,2,3,4}
\rect{(1,1),(3,4)}
\endmfpic
```

8. Polární souřadnice

Příkaz

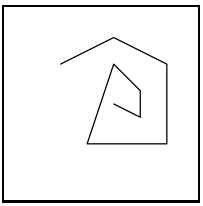
\plr {$(r_1, \theta_1), (r_2, \theta_2), \dots$}

nahradí seznam bodů zadaných polárními souřadnicemi odpovídajícími souřadnicemi pravoúhlými.

Makro

\turtle $\{(x,y),(u_1,v_1),(u_2,v_2),\dots\}$

nakreslí lomenou čáru, která vychází z bodu (x,y) a dále postupuje vždy o daný vektor.

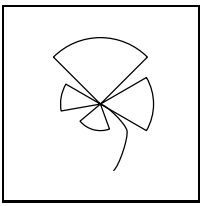


```
\mfpic[10]{0}{5}{0}{5}
\turtle{(1,4),(2,1),(2,-1),(0,-3),
   (-3,0),(1,3),(1,-1),(0,-1),(-1,0.5)}
\endmfpic
```

Kruhovou výseč o středu (x,y) a poloměru r zadáváme dvěma ohraničujícími úhly, a to pomocí příkazu

\sector $\{(x,y),r,\theta_1,\theta_2\}$.

Úhly θ_1 a θ_2 jsou měřeny proti směru hodinových ručiček od rovnoběžky se souřadnou osou x.



```
\mfpic[10]{0}{5}{0}{5}
\sector{(2.5,2.5),2,30,-30}
\sector{(2.5,2.5),2.5,45,135}
\sector{(2.5,2.5),1.5,150,190}
\sector{(2.5,2.5),1,220,290}
\curve[2]{(2.5,2.5),(3.5,1.5),(3,0)}
\endmfpic
```

9. Uložení objektu

Používáme-li ve zdrojovém textu obrázku několikrát jeden *objekt*, který je dvojicí nebo cestou, je výhodné si jej uložit příkazem

\store {*jméno*} {*objekt*}

a na potřebném místě vložit pomocí makra

\mfobj {*jméno*} .

Pomocí \mfobj lze vložit libovolnou proměnnou typu `path` (nejen uloženou pomocí \store), která je využívána v METAPOSTovém kódu našeho obrázku.

Praktické využití těchto příkazů bude ukázáno u vhodných obrázků následujících odstavců.

10. Prefixová makra

Některá z maker balíku `mfpic` se chovají jako tzv. *prefixová* makra, což v praxi znamená, že se tato makra aplikují na příkazy stojící ihned za nimi.

10.1. Kreslení

Makro

\draw [*barva*] ...

vykreslí námi požadovaný objekt plnou čarou.

Příkaz

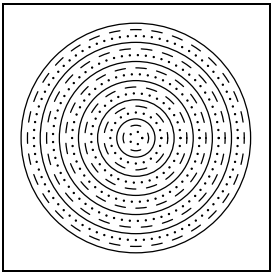
\dashed [*délka*, *mezera*] ...

použije pro vykreslení objektu čáru přerušovanou. Přednastavená délka jedné čárky je dána hodnotou proměnné \dashlen (je rovna 4 pt); mezery mezi jednotlivými čárkami jsou určeny rozměrem \dashspace (je roven rovněž 4 pt). Čárky na začátku a konci křivky jsou dlouhé polovinu z hodnoty \dashlen. Aby byl zachován námi požadovaný poměr čárek a mezer, mohou být námi zadané délky čárek a rozměry mezer v nepovinných parametrech mírně upraveny, tedy mohou být prodlouženy či zkráceny.

Křivku vykreslíme tečkovanou čarou, použijeme-li makro

\dotted [*velikost*, *mezera*] ... .

Předdefinovanou velikost teček udává hodnota proměnné \dotsize (je rovna 0,5 pt), předdefinovaný rozměr mezer je dán hodnotou \dotspace (je roven 3 pt). Na začátku a konci křivky je vykreslena tečka. Velikost teček a mezer může být ovlivněna stejně jako u \dashed.



```
\mfpic[8]{-5.5}{5.5}{-5.5}{5.5}
\dotted\circle{(0,0),0.3}
\dashed\circle{(0,0),0.6}
\draw\circle{(0,0),0.9}
\dotted\circle{(0,0),1.2}
\dashed\circle{(0,0),1.5}
\draw\circle{(0,0),1.8}
\dotted\circle{(0,0),2.1}
\dashed\circle{(0,0),2.4}
\draw\circle{(0,0),2.7}
\dotted\circle{(0,0),3}
\dashed\circle{(0,0),3.3}
\draw\circle{(0,0),3.6}
\dotted\circle{(0,0),3.9}
\dashed\circle{(0,0),4.2}
\draw\circle{(0,0),4.5}
\dotted\circle{(0,0),4.8}
\dashed\circle{(0,0),5.1}
\draw\circle{(0,0),5.4}
\endmfpic
```

Zdá-li se nám tečkovaná nebo čárkovaná čára příliš obyčejná, máme možnost vykreslení křivky například pomocí symbolů trojúhelníku, kosočtverce, křížku či hvězdičky

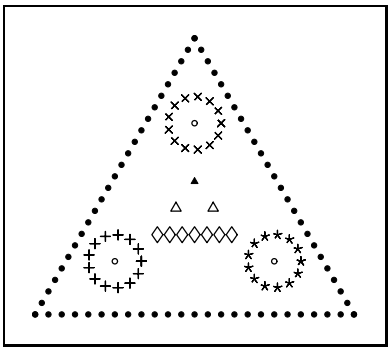
\plot [*velikost*, *mezera*] {*symbol*} ... .

Na místě povinného parametru *symbol* smí být jakýkoli symbol používaný u makra \plotsymbol (viz strana 75). Přednastavená *velikost* odpovídá \pointsize (=2 pt) a *mezera* odpovídá \symbolspace (=5 pt).

Makro

\plotnodes [*velikost*] {*symbol*} ...

také vykresluje *symboly* stejné jako v makru \plotsymbol (viz strana 75), ale nakreslí je pouze v bodech, jimiž jsme zadali křivku, na kterou toto makro aplikujeme. Parametr *velikost* je nastaven na hodnotu rovnu \pointsize (=2 pt).



```
\mfpic[10]{0}{12}{0}{10.4}
\plot{SolidCircle}
       \lines{(0,0),(12,0)}
\plot{SolidCircle}
  \lines{(0,0),(6,sqrt108)}
\plot{SolidCircle}
 \lines{(12,0),(6,sqrt108)}
\plot[3pt,5pt]{Plus}
          \circle{(3,2),1}
\plot[3pt,5pt]{Star}
          \circle{(9,2),1}
\plot[3pt,5pt]{Cross}
  \circle{(6,(2+sqrt27)),1}
\plotnodes{Circle}\polygon{
(3,2),(9,2),(6,(2+sqrt27))}
\plot[4pt,5pt]{Diamond}
    \lines{(4.6,3),(7.4,3)}
\plotnodes[3pt]{Triangle}
    \lines{(5.3,4),(6.7,4)}
\plotsymbol[2pt]
     {SolidTriangle}{(6,5)}
\endmfpic
```

Pro nadefinování vlastní čárkované čáry použijeme příkaz
\dashpattern {*jméno*} {*čárka*$_1$, *mezera*$_1$, *čárka*$_2$, *mezera*$_2$, ... } ,
kde *jménu* je přiřazen vzorek tvořený postupně *čárkou*$_1$, *mezerou*$_1$, *čárkou*$_2$, *mezerou*$_2$ atd. *Čárka* délky 0 pt odpovídá tečce. Popis vzoru musí končit *mezerou*.

Námi nadefinovanou čáru využijeme pro vykreslení křivky pomocí příkazu
\gendashed {*jméno*} ... .

```
\mfpic[10]{0}{5}{0}{5}
\dashpattern{moje}{0pt,1pt,2pt,3pt,4pt,
                   5pt,6pt,7pt,8pt,9pt}
\gendashed{moje}\lines{(0,0),(5,0)}
\pen{2pt}
\dashpattern{takemoje}{0pt,3pt,4pt,7pt}
\gendashed{takemoje}\lines{(0,2),(5,2)}
\pen{4pt}
\dashpattern{jestemoje}{0pt,5pt,7pt,
                                  10pt}
\gendashed{jestemoje}\lines{(0,4.5),
                                (5,4.5)}
\endmfpic
```

10.2. Výplně

Následující *prefixová* makra mohou být aplikována pouze na uzavřené křivky.

Makro

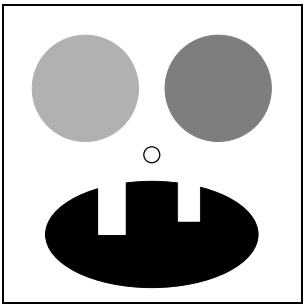
\gfill [*barva*] ...

vyplní uzavřenou křivku barvou uvedenou v nepovinném parametru *barva*. Není-li *barva* uvedena, je použita přednastavená barva odpovídající `fillcolor`. Inverzním příkazem ke \gfill je příkaz \gclear.

Příkaz

\shade [*šeď*] ...

vyplní uzavřenou křivku. Nepovinným parametrem *šeď* můžeme ovlivňovat stupeň šedi (přednastavena je hodnota 0.75`white`).



```
\mfpic[10]{-5}{5}{-5}{5}
\gfill\ellipse{(0,-3),4,2}
\gclear\rect{(-2,-3),(-1,-1)}
\gfill[white]
     \rect{(1,-2.5),(1.8,-1)}
\shade[0.9]\circle{(-2.5,2.5),2}
\shade[0.7]\circle{(2.5,2.5),2}
\circle{(0,0),0.3}
\endmfpic
```

Makro

\thatch [*mezera*,*úhel*] [*barva*] ...

vyšrafuje uzavřenou křivku. Mezery, standardně nastavené na hodnotu \hatchspace (=3 pt), mohou být změněny hodnotou nepovinného parametru *mezera*. Při nulové či záporné hodnotě je křivka vyplněna stejně jako příkazem \gfill. Tloušťka čáry, kterou jsou šrafy vykreslovány, je dána makrem \hatchwd (standardně =0,5 pt). Parametrem *úhel* ovlivníme sklon šrafů (přednastaven je úhel 0°) a parametrem *barva* jejich barvu (ta odpovídá hodnotě `hatchcolor`). Parametry *mezera* a *úhel* zadáváme pouze současně a také při využití parametru *barva* musí být zadány i *mezera* a *úhel*.

Pro práci se šrafováním jsou připravena makra:

\lhatch [*mezera*] [*barva*] ...

šrafuje z levého horního rohu k pravému dolnímu (to znamená pod úhlem −45°);

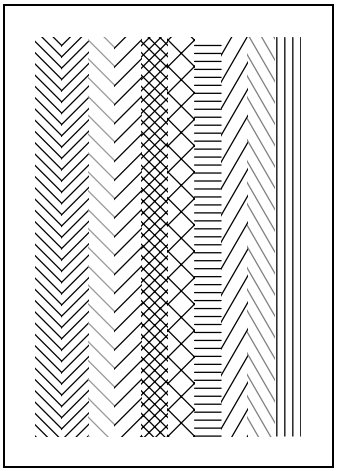
\rhatch [*mezera*] [*barva*] ...

šrafuje z pravého horního k levému dolnímu rohu (to znamená pod úhlem 45°);

\xhatch [*mezera*] [*barva*] ...

\hatch [*mezera*] [*barva*] ...

jsou kombinací `lhatch` a `rhatch`.

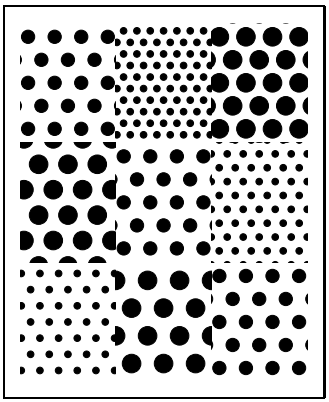


```
\mfpic[10]{0}{10}{0}{15}
\lhatch\rect{(0,0),(1,15)}
\rhatch\rect{(1,0),(2,15)}
\lhatch[5][.6white]
       \rect{(2,0),(3,15)}
\rhatch[5]\rect{(3,0),(4,15)}
\xhatch\rect{(4,0),(5,15)}
\hatch[7]\rect{(5,0),(6,15)}
\thatch\rect{(6,0),(7,15)}
\thatch[5,60 deg]
       \rect{(7,0),(8,15)}
\thatch[4,120 deg][.5white]
       \rect{(8,0),(9,15)}
\thatch[3,90 deg]
      \rect{(9,0),(10,15)}
\endmfpic
```

Makro

\polkadot [*mezera*] ...

vyplní uzavřený útvar kruhy o velikosti udané hodnotou \polkadotwd (=5 pt) vyplněnými barvou fillcolor s mezerami odpovídající hodnotě \polkadotspace (=10 pt).



```
\mfpic[9]{0}{12}{0}{15}
\polkadot\rect{(0,10),(4,15)}
\polkadot\rect{(4,5),(8,10)}
\polkadot\rect{(8,0),(12,5)}
\polkadotwd{2.5pt}
\polkadot[5]
   \rect{(4,10),(8,15)}
\polkadot[6]
   \rect{(8,5),(12,10)}
\polkadot[7]\rect{(0,0),(4,5)}
\polkadotwd{7pt}
\polkadot\rect{(8,10),(12,15)}
\polkadot[11]
   \rect{(0,5),(4,10)}
\polkadot[12]
   \rect{(4,0),(8,5)}
\endmfpic
```

Další možností vyplňování uzavřené křivky je její „pokrytí“ námi nadefinovaným vzorem, jakési „kachličkování“. Vzor vytvoříme pomocí okolí

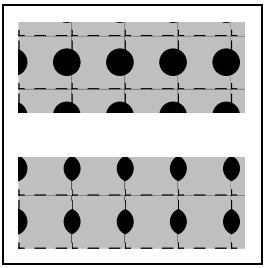
`\tile` {*jméno_vzoru*, *jednotka*, *šířka*, *výška*, *ořezání*}
...
`\endtile` ,
`\begin{tile}` ... `\end{tile}` (v LaTeXu).

Vzor je tvořen v obdélníku, respektive čtverci, s protilehlými vrcholy $(0, 0)$ a (*šířka*, *výška*) a definují ho veškeré příkazy v okolí `\tile` ... `\endtile`, respektive `begin{tile}` ... `end{tile}`. Povinný parametr *jednotka* určuje jednotku, ve které je daný vzor kreslen; parametr *ořezání* má hodnoty `true` (kachlička je oříznuta na námi zadaný obdélník nebo čtverec) a `false` (kachličky nejsou ořezány; při pokládání jsou použity takové, jaké jsme je vytvořili, nezávisle na možné přesahy přes dané rozměry).

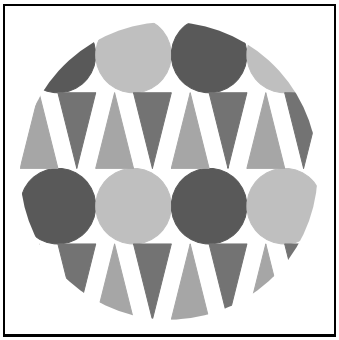
Příkaz

`\tess` {*jméno_vzoru*} ...

vyplní uzavřenou křivku vzorem *jméno_vzoru*. Vždy musíme použít nějaký námi vytvořený vzor, neboť v `mfpic` není žádný předdefinovaný. Při pokusu o pokrytí otevřené křivky nedostaneme chybové hlášení, ale křivka bude pouze vykreslena. Kachličkování je prováděno tak, aby levý dolní roh některé z pokrývajících kachliček ležel v počátku, a dělá se postupně zdola nahoru a zleva doprava (viz efekt v případě `false` na následujícím obrázku).



```
\mfpic[17]{0}{5}{0}{5}
\begin{tile}{kachl_i,20pt,1,1,true}
\dashed\shade\rect{(0,0),(1,1)}
\gfill\circle{(-0.1,0.5),0.25}
\gfill\circle{(1.1,0.5),0.25}
\end{tile}
\tile{kachl_ii,20pt,1,1,false}
\dashed\shade\rect{(0,0),(1,1)}
\gfill\circle{(-0.1,0.5),0.25}
\gfill\circle{(1.1,0.5),0.25}
\endtile
\tess{kachl_i}\rect{(0,0),(5,2)}
\tess{kachl_ii}\rect{(0,3),(5,5)}
\endmfpic
```



```
\mfpic[8]{-7}{7}{-7}{7}
\tile{kachlicka,cm,2,2,true}
\gfill[.35white]
        \circle{(.5,1.5),.5}
\gfill[.75white]
       \circle{(1.5,1.5),.5}
\gfill[.65white]\polygon{
       (0,0),(.25,1),(.5,0)}
\gfill[.45white]\polygon{
       (.5,1),(.75,0),(1,1)}
\gfill[.65white]\polygon{
     (1,0),(1.25,1),(1.5,0)}
\gfill[.45white]\polygon{
     (1.5,1),(1.75,0),(2,1)}
\endtile
\tess{kachlicka}
             \circle{(0,0),7}
\endmfpic
```

10.3. Šipky

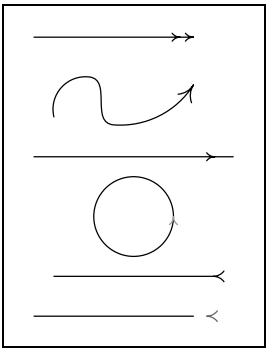
Makro

\arrow [l *délka_šipky*] [r *otočení*] [b *posunutí_zpět*] [c *barva*] .

Jednotlivé nepovinné parametry mají následující význam:

- `b` – posun šipky směrem k počátku křivky, a to po tečně křivky v jejím posledním bodu; přednastavená hodnota pro tento parametr je 0 pt,
- `c` – barva šipky; předdefinovaná barva odpovídá barvě `headcolor`,
- `l` – délka šipky; předdefinovaná délka koresponduje s hodnotou \`headlen`, což je 3 pt,
- `r` – otočení šipky proti směru hodinových ručiček; přednastaven je úhel 0°.

Parametry mohou být zapsány v jakémkoli pořadí. Při použití otočení současně s posunem je třeba myslet na to, že je nejdříve provedeno otočení a poté posun, ovšem ten už probíhá na otočené tečně. Při zápisu nepovinných parametrů tohoto makra může (a nemusí) být mezi `l` a *délkou_šipky*, respektive mezi `r` a *otočením* a podobně, mezera.



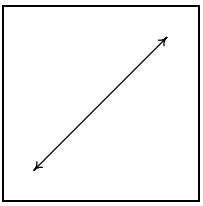
```
\mfpic[15]{0}{5}{0}{7}
\arrow\arrow[b 5pt]\lines{
                          (0,7),(4,7)}
\arrow[l 6pt]\curve{(0.5,5),(1.4,6),
                    (2,4.8),(4,5.8)}
\arrow[b 7pt]\lines{(0,4),(5,4)}
\arrow[c .6white]\circle{
                          (2.5,2.5),1}
\arrow[r180][l4pt]
             \lines{(0.5,1),(4.5,1)}
\arrow[r180][l4pt][b5pt][c.4white]
                 \lines{(0,0),(4,0)}
\endmfpic
```

10.4. Změna orientace křivky

Orientaci křivky můžeme jednoduše změnit příkazem

`\reverse` ... .

Tohoto makra využijeme například při vykreslování šipek v počátečním i koncovém bodu křivky:



```
\mfpic[10]{0}{5}{0}{5}
\arrow\reverse\arrow\lines{(0,0),(5,5)}
\endmfpic
```

První šipka zleva ve zdrojovém zápisu je kreslena v počátečním bodu dané křivky.

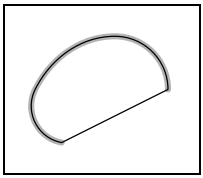
10.5. Uzavírání křivek

V `mfpic` jsou nadefinované tyto uzavřené křivky: `\rect`, `\circle`, `\ellipse`, `\sector`, `\cyclic`, `\polygon`, `\plrregion` a `\btwnfcn`. Libovolné jiné křivky, které se nám na pohled mohou jevit uzavřenými, `mfpic` chápe jako otevřené.

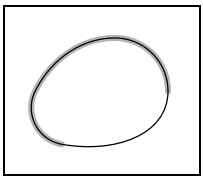
Pokud jsme vytvořili otevřenou křivku, můžeme využít některý z následujících příkazů pro její uzavření. Všechny uvedené příkazy způsobí uzavření i z pohledu `mfpic`.

- `\lclosed` – uzavře křivku úsečkou z počátečního do koncového bodu křivky
- `\bclosed` – uzavře křivku Bézierovou křivkou vedenou z prvního k poslednímu bodu dané křivky

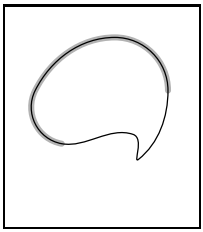
- \cbclosed – uzavře křivku doplněním kubického B-splinu mezi prvním a posledním bodem zadané křivky
- \sclosed – uzavře křivku tak, aby se jevila co nejhladší, přičemž v tomto smyslu může být změněn tvar křivky, kterou uzavíráme
- \uclosed – uzavře křivku tak, aby se jevila co nejhladší, ale tvar zadané křivky nezmění



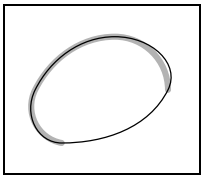
```
\mfpic[20]{0.5}{3}{1}{3}
\pen{2.3pt}
\draw[.7white]\curve{(1,1),(.5,2),
                          (2,3),(3,2)}
\pen{.5pt}
\lclosed\curve{(1,1),(.5,2),(2,3),
                                (3,2)}
\endmfpic
```



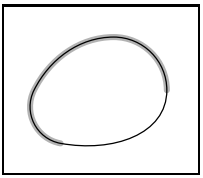
```
\mfpic[20]{0.5}{3}{1}{3}
\pen{2.3pt}
\draw[.7white]\curve{(1,1),(.5,2),
                          (2,3),(3,2)}
\pen{.5pt}
\bclosed\curve{(1,1),(.5,2),(2,3),
                                (3,2)}
\endmfpic
```



```
\mfpic[20]{0.5}{3}{0}{3}
\pen{2.3pt}
\draw[.7white]\curve{(1,1),(.5,2),
                          (2,3),(3,2)}
\pen{.5pt}
\cbclosed\curve{(1,1),(.5,2),(2,3),
                                (3,2)}
\endmfpic
```



```
\mfpic[20]{0.5}{3}{1}{3}
\pen{2.3pt}
\draw[.7white]\curve{(1,1),(.5,2),
                          (2,3),(3,2)}
\pen{.5pt}
\sclosed\curve{(1,1),(.5,2),(2,3),
                                (3,2)}
\endmfpic
```



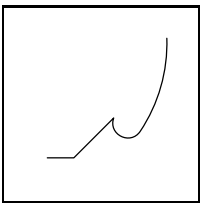
```
\mfpic[20]{0.5}{3}{1}{3}
\pen{2.3pt}
\draw[.7white]\curve{(1,1),(.5,2),
                          (2,3),(3,2)}
\pen{.5pt}
\uclosed\curve{(1,1),(.5,2),(2,3),
                                (3,2)}
\endmfpic
```

10.6. Napojení křivek

Okolí

\connect ... \endconnect ,
\begin{connect} ... \end{connect} (pro LaTeX)

sestrojí úsečku z koncového bodu výše zapsané otevřené křivky k počátečnímu bodu další otevřené křivky. Výsledkem je otevřená křivka.



```
\mfpic[10]{0}{5}{0}{5}
\begin{connect}
\lines{(0.5,0.5),(1.5,0.5)}
\curve{(3,2),(4,1.5),(5,5)}
\end{connect}
\endmfpic
```

10.7. Afinní transformace

V mfpic můžeme použít následující afinní transformace:

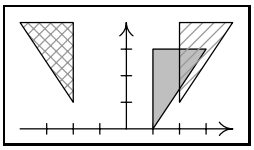
- \rotate $\{\theta\}$ – otočení o θ stupňů kolem počátku proti směru hodinových ručiček,
- \rotatearound $\{(x,y)\}$ $\{\theta\}$ – otočení o θ stupňů kolem bodu (x,y) proti směru hodinových ručiček,
- \turn $[(x,y)]$ $\{\theta\}$ – otočení o θ stupňů kolem bodu (x,y) proti směru hodinových ručiček; není-li uveden nepovinný parametr, otáčení je provedeno kolem počátku,
- \mirror $\{(x_1,y_1)\}$ $\{(x_2,y_2)\}$ – osová souměrnost zadaná přímkou procházející body (x_1,y_1) a (x_2,y_2),
- \reflectabout $\{(x_1,y_1)\}$ $\{x_2,y_2\}$ – osová souměrnost daná přímkou procházející body (x_1,y_1) a (x_2,y_2),
- \shift $\{(u,v)\}$ – posunutí o vektor (u,v),
- \scale $\{k\}$ – stejnolehlost se středem v počátku a koeficientem k,

- `\xscale` $\{k\}$ – změna měřítka na ose x s koeficientem k,
- `\yscale` $\{k\}$ – změna měřítka na ose y s koeficientem k,
- `\zscale` $\{(u,v)\}$ – stejnolehlost s koeficientem o velikosti vektoru (u,v) a otočení proti směru hodinových ručiček o úhel, který svírá vektor (u,v) s kladnou částí osy x (bodu (x,y) odpovídá bod $(xu-yv, xv+yu)$),
- `\xslant` $\{k\}$ – zkosení ve směru osy x s koeficientem k (obrazem bodu (x,y) je bod $(x+ky,y)$),
- `\yslant` $\{k\}$ – zkosení ve směru osy y s koeficientem k (bodu (x,y) odpovídá bod $(x,kx+y)$),
- `\zslant` $\{(u,v)\}$ – transformace, která bod (x,y) zobrazí na bod $(xu+yv, xv+yu)$,
- `\boost` $\{\chi\}$ – má stejný efekt jako `zslant` $\{(\cosh\chi, \sinh\chi)\}$,
- `\xyswap` – osová souměrnost daná osou $y=x$.

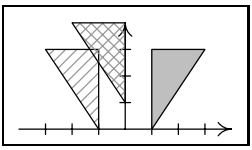
10.8. Používání afinních transformací

Použijeme-li libovolnou afinní transformaci, budou se jí podrobovat všechny objekty v aktuálním okolí `\mfpic` ... `\endmfpic` zapsané níže od této transformace. Při aplikaci další transformace dojde ke složení s předcházející.

Dvěma nepříliš složitými obrázky můžeme ukázat, že grupa afinních transformací s operací skládání není komutativní. (U obou obrázků využíváme posunutí o vektor $(1,1)$ a osovou souměrnost podle osy y, přičemž v prvním obrázku musí být zadaná osa y posunuta o odpovídající vektor, neboť `mfpic` ji před provedením osové souměrnosti posune o vektor předcházejícího posunutí; u druhého obrázku zase upravíme příslušným způsobem posunutí tak, abychom neprováděli posunutí o vektor $(1,1)$ zobrazený v předchozí osové souměrnosti.)



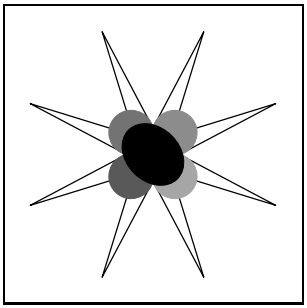
```
\mfpic[10]{-4}{4}{0}{4}
\axes
\xmarks{-3,-2,-1,1,2,3}
\ymarks{1,2,3}
\store{trojuhelnik}
       {\polygon{(1,0),(1,3),(3,3)}}
\draw\shade\mfobj{trojuhelnik}
\shift{(1,1)}\draw
            \rhatch[\the\hatchspace]
         [.6white]\mfobj{trojuhelnik}
\reflectabout{(-1,-1)}{(-1,3)}\draw
                   \hatch[3][.6white]
                  \mfobj{trojuhelnik}
\endmfpic
```



```
\mfpic[10]{-4}{4}{0}{4}
\axes
\xmarks{-3,-2,-1,1,2,3}
\ymarks{1,2,3}
\store{trojuhelnik}
  {\polygon{(1,0),(1,3),(3,3)}}
\draw\shade\mfobj{trojuhelnik}
\reflectabout{(0,0)}{(0,3)}\draw
   \rhatch[\the\hatchspace][.6white]
                    \mfobj{trojuhelnik}
\shift{(-1,1)}\draw
                     \hatch[3][.6white]
                    \mfobj{trojuhelnik}
\endmfpic
```

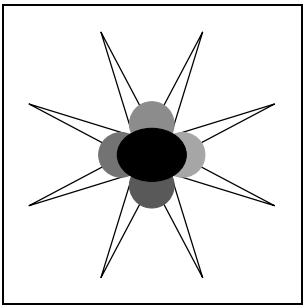
Pro případ, kdy nám skládání transformací není moc vhod, lze použít okolí
\coords ... \endcoords ,
\begin{coords} ... \end{coords} (pro LaTeX).
Při otevření okolí dojde k „uschování“ právě aktuální transformace a v okamžiku uzavření je transformace navrácena.

Následující tři obrázky přibližují chování tohoto okolí. První z nich provede postupné otáčení trojúhelníku o 45°, otevřením okolí `coords` je uloženo otočení o $7\cdot45°$ a poté je vykreslen (nezapomeňme, že je nejprve otočen o $7\cdot45°$) a otočen další útvar.



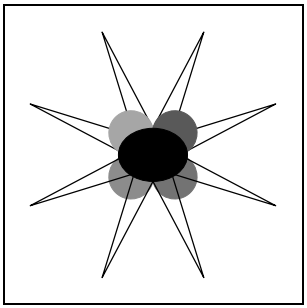
```
\mfpic[10]{-5}{5}{-5}{5}
\store{listik}{\polyline{(1,0),
          5*dir(22.5).dir(45)}}
\store{listecek}{\lclosed\curve{
        dir(-45),(2,0),dir(45)}}
\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\begin{coords}
\gfill[.65white]\mfobj{listecek}
\rotate{90}\gfill[.55white]
                \mfobj{listecek}
\rotate{90}\gfill[.45white]
                \mfobj{listecek}
\rotate{90}\gfill[.35white]
                \mfobj{listecek}
\end{coords}
\gfill\ellipse{(0,0),1.3,1}
\endmfpic
```

Ve druhém obrázku je na počátku uložena identita, neboť žádná jiná transformace nebyla použita, potom je aplikováno otáčení vždy o 45°, to jest celkem o $7 \cdot 45°$. Uzavřením prvního okolí `coords` je zapomenuto otočení o $7 \cdot 45°$ a je navrácena identita, což způsobí, že útvar vytvořený v dalším okolí `coords` se nijak netransformuje.



```
\mfpic[10]{-5}{5}{-5}{5}
\store{listik}{\polyline{(1,0),
          5*dir(22.5),dir(45)}}
\store{listecek}{\lclosed\curve{
        dir(-45),(2,0),dir(45)}}
\mfobj{listik}
\begin{coords}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\end{coords}
\begin{coords}
\gfill[.65white]\mfobj{listecek}
\rotate{90}\gfill[.55white]
                \mfobj{listecek}
\rotate{90}\gfill[.45white]
                \mfobj{listecek}
\rotate{90}\gfill[.35white]
                \mfobj{listecek}
\end{coords}
\gfill\ellipse{(0,0),1.3,1}
\endmfpic
```

Ve třetím obrázku jsou dvě okolí `coords` vnořena do sebe, což je praktické v situacích, kdy využíváme některé z transformací a na určité objekty potřebujeme aplikovat ještě nějaké transformace další. V naší ukázce je tedy trojúhelník nejprve otočen třikrát, ve vnořeném okolí `coords` je první vykreslovaný útvar otočen o $3\cdot45°$, druhý útvar je otočen o $3\cdot45°+90°$ atd.; při uzavírání vnořeného okolí `coords` je vrácena hodnota otočení na $3\cdot45°$.



```
\mfpic[10]{-5}{5}{-5}{5}
\store{listik}{\polyline{(1,0),
          5*dir(22.5),dir(45)}}
\store{listecek}{\lclosed\curve{
        dir(-45),(2,0),dir(45)}}
\mfobj{listik}
\begin{coords}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\begin{coords}
\gfill[.65white]\mfobj{listecek}
\rotate{90}\gfill[.55white]
                \mfobj{listecek}
\rotate{90}\gfill[.45white]
                \mfobj{listecek}
\rotate{90}\gfill[.35white]
                \mfobj{listecek}
\end{coords}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\rotate{45}\mfobj{listik}
\end{coords}
\gfill\ellipse{(0,0),1.3,1}
\endmfpic
```

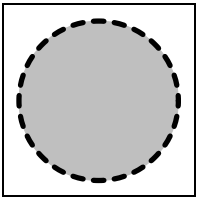
Zdrojové texty předchozích příkladů bychom mohli učiniti trochu „elegantnějšími“ využitím cyklu TEXu, kterým nahradíme sedmkrát prováděné otáčení:

```
\newcount\pocet
\def\kresli#1{\pocet=#1
    \loop\ifnum\pocet>0
         \rotate{45}\mfobj{listik}
         \advance \pocet by -1
    \repeat}
\kresli{7}
```

10.9. Pořadí prefixových maker

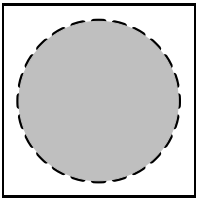
Při práci s tímto druhem maker je důležité si uvědomit, že příkaz vlevo je vždy aplikován na výsledek příkazu vpravo. Jinak řečeno, námi napsaný postup tvorby obrázku je prováděn od konce (zprava).

Podívejme se, jak se mění výsledek kreslení, použijeme-li různá pořadí příkazů \gfill, \dashed:



```
\mfpic[30]{-1}{1}{-1}{1}
\pen{2pt}
\dashed\gfill[0.75white]\circle{(0,0),1}
\endmfpic
```

Nejdříve je nadefinována kružnice se středem $(0, 0)$ a poloměrem 1, poté je vyplněna. Hranice vytvořeného kruhu je vyznačen čárkovanou čarou, přičemž střed pera je veden po hraniční křivce (kružnici), což způsobí, že polovina tloušťky kružnice překrývá výplň kruhu.



```
\mfpic[30]{-1}{1}{-1}{1}
\pen{2pt}
\gfill[0.75white]\dashed\circle{(0,0),1}
\endmfpic
```

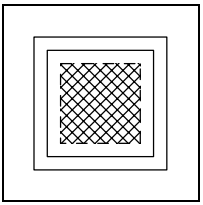
Této posloupnosti příkazů odpovídá překrytí poloviny tloušťky pera, vyznačujícího kružnici, výplní kruhu.

11. Rendrování

Redrováním rozumíme proces vlastního překreslení geometrického popisu obrázku. V METAPOSTu je to konkrétně tvorba postscriptového popisu křivek, výplní a podobně. Příkazem

\setrender {*příkazy_TEXu*}

změníme přednastavený způsob rendrování – pomocí \draw.



```
\mfpic[10]{0}{5}{0}{5}
\rect{(0,0),(5,5)}   % \draw není nutné
\draw\rect{(.5,.5),(4.5,4.5)}
\setrender{\dashed\xhatch}
\rect{(1,1),(4,4)}
\endmfpic
```

12. Funkce

Makro

\fdef {*název_funkce*} {*proměnná*$_1$, *proměnná*$_2$, ... } {*předpis*}

nadefinuje funkci *název_funkce* (*název_funkce* smí být složen pouze z písmen a podtržítka) o proměnných *proměnná*$_1$, *proměnná*$_2$, ... a předpisem *předpis*, který je předán METAPOSTu ke zpracování. Můžeme tudíž používat všechny operace, které umožňuje METAPOST, tedy $+$, $-$, $*$, $/$, $**$; závorky pro upřesňování pořadí operací (,) ; znaménka rovnosti, respektive nerovnosti $=$, $<$, $>$, $<=$, $>=$ a další funkce uvedené v následující tabulce:

`round`	zaokrouhlení podle obvyklých pravidel,
`floor`	zaokrouhlení směrem dolů,
`ceiling`	zaokrouhlení směrem nahoru,
`min`	minimum (dva a více argumentů),
`max`	maximum (dva a více argumentů),
`abs`	absolutní hodnota,
`sqrt`	druhá odmocnina,
`sind`	funkce sinus (argument zadáváme ve stupních),
`cosd`	funkce kosinus (argument zadáváme ve stupních),
`mlog`	logaritmus se základem $e^{1/256}$ (`mlog`$= 256 * \ln x$),
`mexp`	inversní funkce k funkci předchozí.

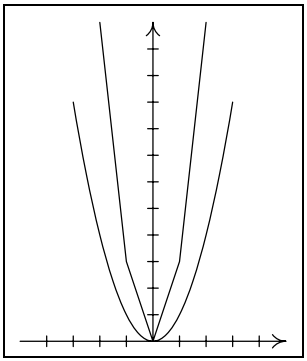
Dále můžeme využít funkce předdefinované v `grafbase.mp`, což jsou tyto:

`tand`	funkce tangens (argument zadáváme ve stupních),
`cotd`	funkce kotangens (argument zadáváme ve stupních),
`secd`	funkce sekans (argument zadáváme ve stupních),
`cscd`	funkce kosekans (argument zadáváme ve stupních),
`asin`	funkce arkussinus (výsledek dostáváme ve stupních),
`acos`	funkce arkuskosinus (výsledek dostáváme ve stupních),
`atan`	funkce arkustangens (výsledek dostáváme ve stupních),
`sin`	funkce sinus (argument zadáváme v radiánech),
`cos`	funkce kosinus (argument zadáváme v radiánech),
`tan`	funkce tangens (argument zadáváme v radiánech),
`cot`	funkce kotangens (argument zadáváme v radiánech),
`sec`	funkce sekans (argument zadáváme v radiánech),
`csc`	funkce kosekans (argument zadáváme v radiánech),
`invsin`	funkce arkussinus (výsledek dostáváme v radiánech),
`invcos`	funkce arkuskosinus (výsledek dostáváme v radiánech),
`invtan`	funkce arkustangens (výsledek dostáváme v radiánech),
`exp`	exponenciální funkce e^x,
`sinh`	funkce hyperbolický sinus,
`cosh`	funkce hyperbolický kosinus,

`tanh`	funkce hyperbolický tangens,
`ln`	přirozený logaritmus,
`asinh`	funkce argument hyperbolického sinu,
`acosh`	funkce argument hyperbolického kosinu,
`atanh`	funkce argument hyperbolického tangens.

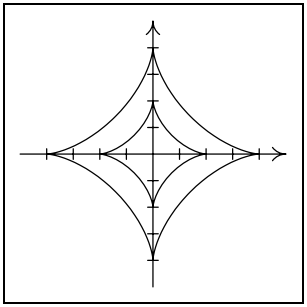
Následující čtyři makra mají společný nepovinný parametr *typ_křivky*, který definuje, zda je funkce vykreslena jako Bézierova křivka (odpovídá hodnota parametru `s`) nebo jako lomená čára (odpovídá hodnota `p`). Dále je pro nás podstatné, že smíme využívat pouze proměnných x a t, jak je udáno u jednotlivých maker:

- `\function` [*typ_křivky*] {$x_{\min}$, $x_{\max}$, *krok*} {$f(x)$} – vykresluje funkci $f(x)$ na intervalu $\langle x_{\min}, x_{\max}\rangle$ (přičemž METAPOST postupuje od $x_{\min}$ postupně po kroku *krok* až k $x_{\max}$); nepovinný parametr je přednastaven na `s`,
- `\parafcn` [*typ_křivky*] {$t_{\min}$, $t_{\max}$, *krok*} {$f(t)$} – vytvoří parametricky zadanou křivku $f(t) = (x(t), y(t))$ s parametrem t z intervalu $\langle t_{\min}, t_{\max}\rangle$ (a krokem *krok*); nepovinný argument je přednastaven na `s`,
- `\plrfcn` [*typ_křivky*] {$t_{\min}$, $t_{\max}$, *krok*} {$f(t)$} – určuje křivku $f(t)$ danou polárními souřadnicemi (t, r), kde $r = f(t)$; úhel t nabývá hodnot z intervalu $\langle t_{\min}, t_{\max}\rangle$ a je měřen ve stupních; nepovinný parametr je nastaven jako `s`,
- `\btwnfcn` [*typ_křivky*] {$x_{\min}$, $x_{\max}$, *krok*} {$f(x)$} {$g(x)$} – vyznačí oblast ohraničenou funkcemi $f(x)$, $g(x)$ na intervalu $\langle x_{\min}, y_{\max}\rangle$ a dvěma vertikálními přímkami v $x_{\min}$, $x_{\max}$; parametr *typ_křivky* je nastaven jako `p`,
- `\plrregion` [*typ_křivky*] {$t_{\min}$, $t_{\max}$, *krok*} {$f(t)$} – stanoví oblast ohraničenou křivkou *f(t)* zadanou polárními souřadnicemi (t, r), kde $r = f(t)$; úhel t nabývá hodnot v intervalu $\langle t_{\min}, t_{\max}\rangle$ a je měřen ve stupních; oblast je uzavřena úsečkou spojující počátek souřadné soustavy a bod odpovídající $t_{\min}$ a spojnicí bodu odpovídajícího $t_{\max}$ s počátkem; nepovinný parametr je přednastaven na `p`.



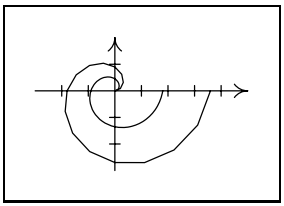
```
\mfpic[10]{5}{5}{0}{12}
\axes
\xmarks{-4,-3,-2,-1,1,2,3,4}
\ymarks{1,2,3,4,5,6,7,8,9,10,11}
\function{-3,3,.5}{x*x}
\function[p]{-2,2,1}{3*x*x}
\endmfpic
```

Asteroida:

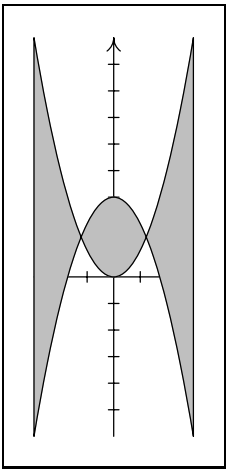


```
\mfpic[10]{-5}{5}{-5}{5}
\axes
\xmarks{-4,-3,-2,-1,1,2,3,4}
\ymarks{-4,-3,-2,-1,1,2,3,4}
\parafcn{0,1440,22.5}{
((3*cosd (t/4))+cosd ((3*t)/4),
(3*sind (t/4))-sind ((3*t)/4))}
\parafcn[p]{0,1440,22.5}{
    (.5*((3*cosd (t/4))+
         cosd ((3*t)/4)),
      .5*((3*sind(t/4))-
        sind ((3*t)/4)))}
\endmfpic
```

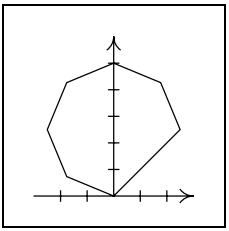
Archimédova spirála:



```
\mfpic[10]{-3}{5}{-3}{2}
\axes
\xmarks{-2,-1,1,2,3,4}
\ymarks{-2,-1,1}
\plrfcn{0,360,22.5}{.005*t}
\plrfcn[p]{0,360,22.5}{.01*t}
\endmfpic
```

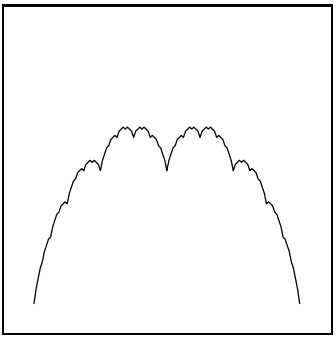


```
\mfpic[10]{-3}{3}{-6}{9}
\axes
\xmarks{-2,-1,1,2}
\ymarks{-5,-4,-3,-2,-1,1,2,3,4,5,6,7,8}
\shade\btwnfcn[s]{-3,3,.5}{x*x}{-x*x+3}
\btwnfcn[s]{-3,3,.5}{x*x}{-x*x+3}
\endmfpic
```



```
\mfpic[10]{-3}{3}{0}{6}
\axes
\xmarks{-2,-1,1,2}
\ymarks{1,2,3,4,5}
\plrregion{45,180,22.5}{5*sind(t)}
\endmfpic
```

Funkce na následujícím obrázku je sedmým částečným součtem řady, která konverguje ke spojité funkci nemající nikde derivaci. V `mfpic` je znázornění takové funkce velmi jednoduché.



```
\mfpic[100]{0}{1}{0}{1}
\fdef{f}{x}{min(x-(floor x),
             (ceiling x)-x)}
\function[p]{0,1,1/128}{f(x)+
                +(.5*f(2*x))+
               +(.25*f(4*x))+
              +(.125*f(8*x))+
            +(.0625*f(16*x))+
           +(.03125*f(32*x))+
         +(.015625*f(64*x))}
\endmfpic
```

13. Kreslení dat z externího souboru

Balík maker `mfpic` nám umožňuje vykreslovat útvary sestrojené z dat externích souborů.

Externí soubory použité v tomto textu jsou nazvány `data`, `data1` a podobně. Při spouštění TEXu z instalace emTEX byla tato jména nepostačující (pokud neměla příponu) a musela být doplněna o tečku na konci jména (`data.`).

Příkaz

\datafile {*jméno_souboru*}

vytvoří lomenou čáru procházející body zapsanými v souboru *jméno_souboru*. Předpokládáme, že každý neprázdný řádek obsahuje alespoň dvě čísla. První dvě čísla na řádku reprezentují souřadnice x a y jednoho bodu, ostatní čísla jsou ignorována. Prázdné řádky na začátku souboru jsou ignorovány, všechny další jsou chápány jako konec křivky; komentář je uvozen znakem procenta (%); viz příkaz \mfpdatacomment. Takto vytvořený objekt smíme uzavřít, vybarvit a podobně.

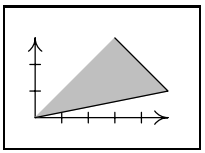
Příkaz

\smoothdata [*napětí*]

způsobí, že výsledkem makra \datafile je Bézierova křivka místo lomené čáry, a příkaz

\unsmoothdata

nastaví zpět použití lomené čáry.



```
\mfpic[10]{0}{5}{0}{3}
\axes
\xmarks{1,2,3,4}
\ymarks{1,2}
\shade\lclosed\datafile{data}
\datafile{data}
\endmfpic
```

Soubor `data` je pro tento příklad zadán:

```
   prázdná řádka
   prázdná řádka
   prázdná řádka
0   0    7          2
5 1         5
3 3

0 0
3 4
0 5
```

Chceme-li uvozovat komentáře jiným způsobem, než-li znakem procenta, nadefinujeme si k tomuto účelu námi zvolený znak příkazem

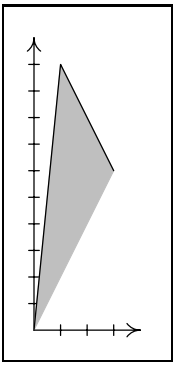
\mfpdatacomment *znak*

Zároveň bude při čtení daného souboru zrušeno zvláštní postavení znaku procenta.

Příkaz

\using {*vstupní_vzor*} {*výstupní_vzor*}

změní standardní způsob zpracování datového souboru. Použijeme-li například vzorek \using{#1 #2 #3}{#3,#1}, bude to znamenat následující: čísla na řádku až po první mezeru jsou přiřazena do parametru #1 *vstupního_vzoru*, další čísla až po druhou mezeru do parametru #2 a zbývající čísla na řádku do parametru #3 *vstupního_vzoru*. Pro vykreslení křivky jsou použity souřadnice dle *výstupního_vzoru* odpovídající jednotlivým parametrům *vstupního_vzoru*. Námi nadefinovaný způsob zpracování datových souborů platí až do konce okolí mfpic, tedy až do příkazu \endmfpic, a vztahuje se na zpracování souborů při použití příkazů \datafile i \plotdata (viz následující příkaz).



```
\mfpic[10]{0}{4}{0}{11}
\axes
\xmarks{1,2,3}
\ymarks{1,2,3,4,5,6,7,8,9,10}
\using{#1 #2 #3}{#2,#1*2}
\shade\lclosed\datafile{data}
\datafile{data}
\endmfpic
```

Pro nakreslení několika lomených čar (nebo Bézierových křivek po využití příkazu \smoothdata) zadaných v jednom datovém souboru do jednoho obrázku je připraveno makro

\plotdata {*jméno_souboru*}

Jednotlivé čáry jsou od sebe odděleny jednou volným řádkem, více prázdných řádků je chápáno jako konec souboru. Čáry jsou vykreslovány postupně, a to šesti různými typy čar. Příkazem \coloredlines změníme barvu čar (střídá se osm různých barev počínající od černé). Příkazy \pointedlines a \datapointsonly využívají makra \plot a \plotnodes a způsobují vykreslení čar nebo pouze zadaných bodů pomocí devíti různých symbolů známých z makra \plotsymbol (viz strana 75).

Nastavení původního způsobu (to jest různými typy čar) kreslení se provádí příkazem \dashedlines. Chceme-li ovlivnit první typ čáry (respektive barvu

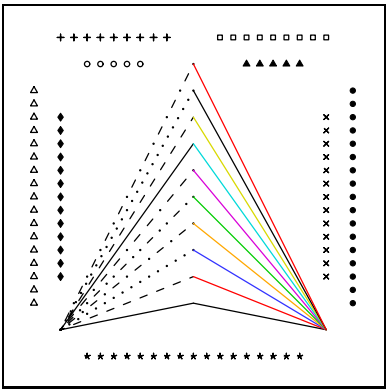
nebo druh symbolu), použijeme \mfplinestyle{*číslo*}, kde *číslo* je nezáporné. Typy čar (respektive barvy, druhy symbolu) jsou číslovány od 0; je-li v okolí mfpic více příkazů \plotdata, číslování vždy navazuje na číslování předcházející.

U tohoto příkazu se nesmí využívat žádná prefixová makra.

Pro datové soubory data1, data2, data3:

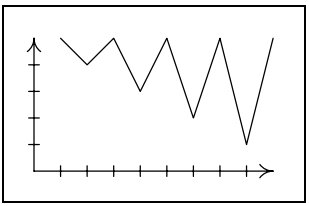
```
data1          data2          data3
 0  0          10  0           1 -1
 5  1           5  1           9 -1

 0  0          10  0          -1  1
 5  2           5  2          -1  9

 0  0          10  0          11  1
 5  3           5  3          11  9

 0  0          10  0           0 11
 5  4           5  4           4 11

 0  0          10  0           6 11
 5  5           5  5          10 11

 0  0          10  0           0  2
 5  6           5  6           0  8

 0  0          10  0          10  2
 5  7           5  7          10  8

 0  0          10  0           1 10
 5  8           5  8           3 10

 0  0          10  0           7 10
 5  9           5  9           9 10

 0  0          10  0
 5 10           5 10
```

dostaneme následující obrázek.



```
\mfpic[10]{-1}{11}{-1}{11}
\plotdata{data1}
\coloredlines
\plotdata{data2}
\pointedlines
\plotdata{data3}
\endmfpic
```

V dalším obrázku využijeme makro \sequence, které reprezentuje posloupnost přirozených čísel. V našem případě nabývají x-ové souřadnice bodů hodnot 1, 2, ... až počet řádků v souboru.



```
\mfpic[10]{0}{9}{0}{5}
\axes
\xmarks{1,2,3,4,5,6,7,8}
\ymarks{1,2,3,4}
\using{#1}{\sequence,#1}
\datafile{data4}
\endmfpic
```

Soubor data4:

```
5
4
5
3
5
2
5
1
5
```

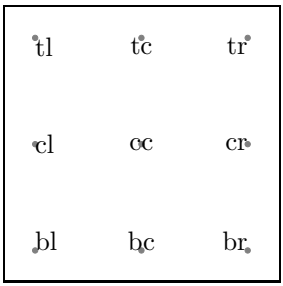
14. Popisy obrázků

Příkazem

\tlabel [*umístění otočení*] (x, y) {*popiska*}

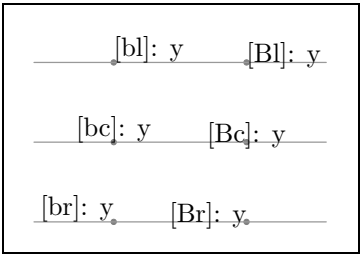
umístíme k bodu (x, y) text *popiska*. Parametr *umístění* je složen ze dvou písmen; první písmeno udává vertikální polohu (čtyři varianty: `t`, `c`, `b`, `B` odpovídající postupně anglickým výrazům top, center, bottom, Baseline), druhé polohu horizontální (tři varianty: `l`, `c`, `r` korespondující s anglickými left, center, right). Mezi písmeny nesmí být mezera. Použitím jednotlivých kombinací získáme popisky umístěné vzhledem k bodu (x, y) podle následujících obrázků; přednastavená hodnota tohoto argumentu je `Bl`.

Všimněme si, že zadáváme polohu bodu vůči popisce (u METAPOSTu tomu bylo naopak).




```
\mfpic[10]{-4}{4}{-4}{4}
\fillcolor{0.5white}
\point{(-4,4),(0,4),(4,4),
       (-4,0),(0,0),(4,0),
    (-4,-4),(0,-4),(4,-4)}
\tlabel[lt](-4,4){lt}
\tlabel[tc](0,4){tc}
\tlabel[tr](4,4){tr}
\tlabel[cl](-4,0){cl}
\tlabel[cc](0,0){cc}
\tlabel[cr](4,0){cr}
\tlabel[bl](-4,-4){bl}
\tlabel[bc](0,-4){bc}
\tlabel[br](4,-4){br}
\endmfpic
```

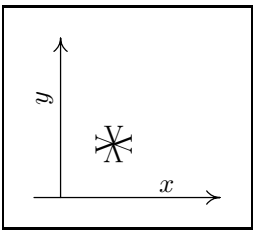
Použijeme-li argument `b` je text sázen tak, aby bounding box písmene byl postaven na účaří, zatímco argument `B` ztotožní y-ovou souřadnici referenčního bodu bounding boxu písmene s y-ovou souřadnicí bodu, který zadáváme. Rozdíl mezi parametrem `b` a `B` je vidět na ukázce písmene y:




```
\mfpic[10]{-3}{8}{-6}{1}
\fillcolor{.5white}
\point{(0,0),(5,0),(0,-3),
     (5,-3),(0,-6),(5,-6)}
\draw[.6white]
      \lines{(-3,0),(8,0)}
\draw[.6white]
    \lines{(-3,-3),(8,-3)}
\draw[.6white]
    \lines{(-3,-6),(8,-6)}
\tlabel[bl](0,0){[bl]: y}
\tlabel[Bl](5,0){[Bl]: y}
\tlabel[bc](0,-3){[bc]: y}
\tlabel[Bc](5,-3){[Bc]: y}
\tlabel[br](0,-6){[br]: y}
\tlabel[Br](5,-6){[Br]: y}
\endmfpic
```

Parametrem *otočení* text otáčíme kolem bodu, který popisujeme; přednastaveno je nula stupňů. Mezi parametry *umístění* a *otočení* může a nemusí být mezera.

O zpracování popisek rozhoduje volba `mplabels` (viz strana 71). Je-li využita, umisťuje a otáčí popisky METAPOST, v opačném případě je otáčení vyloučeno (nepovinný argument *otočení* je ignorován) a text sází TEX.



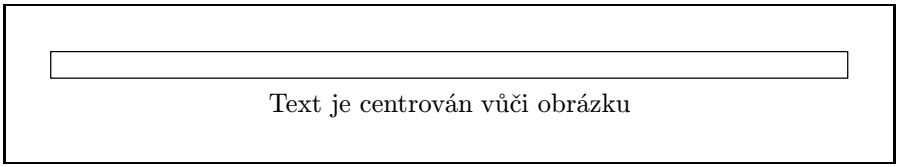

```
\mfpic[20]{-.5}{3}{0}{3}
\axes
\tlabel[bc](2,0.1){$x$}
\tlabel[cr 90](-0.3,2){$y$}
\tlabel[bc](1,1){V}
\tlabel[bc 90](1,1){V}
\tlabel[bc180](1,1){V}
\tlabel[bc 270](1,1){V}
\endmfpic
```

Příkaz

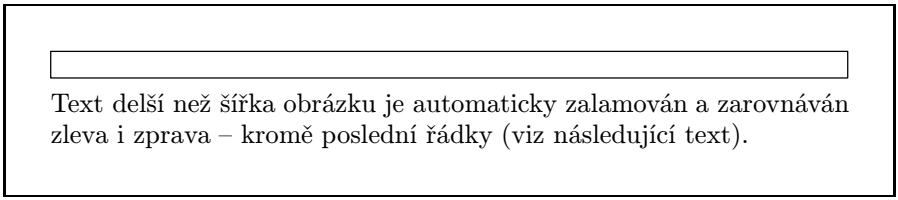
\tcaption [*maximum*, *řádka*] {*text*}

umístí *text* pod obrázek.

Chování makra v různých situacích ukazují nejlépe následující obrázky.

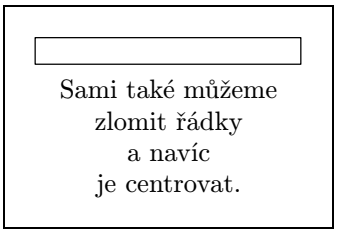



```
\mfpic[10]{0}{30}{0}{1}
\rect{(0,0),(30,1)}
\tcaption{Text je centrován vůči obrázku}
\endmfpic
```




```
\mfpic[10]{0}{30}{0}{1}
\rect{(0,0),(30,1)}
\tcaption{Text delší než šířka obrázku je automaticky zalamován
          a~zarovnáván zleva i~zprava -- kromě poslední řádky
          (viz následující text).}
\endmfpic
```

Překročí-li délka řádky *textu* hodnotu danou součinem *maxima* a šířky obrázku, jsou řádky lámány tak, aby jejich šířka odpovídala hodnotě součinu *řádky* a šířky obrázku. Přednastavená hodnota *maxima* je 1,2; *řádky* 1,0.




```
\mfpic[10]{0}{10}{0}{1}
\rect{(0,0),(10,1)}
\usecenteredcaptions
\tcaption{Sami také můžeme\\
                zlomit řádky\\
                     a~navíc\\
                zlomit řádky.}
\endmfpic
```

15. Pole křivek

Okolí

\patharr {*jméno_pole*} ... \endpatharr ,

\begin{patharr} {*jméno_pole*} ... \end{patharr} (v LaTeXu),

v němž je zapsáno několik křivek, vytvoří z těchto křivek pole nazvané *jméno_pole*. Na jednotlivé položky tohoto pole se odkazujeme příkazy \mfobj{*jméno_pole1*}, \mfobj{*jméno_pole2*}, ...

```
\mfpic[10]{0}{6}{0}{3}
\patharr{pole}
\lines{(1,0),(1,3)}
\rect{(2,0),(3,3)}
\polygon{(4,0),(5,3),(6,0)}
\endpatharr
\mfobj{pole1}
\shade\mfobj{pole2}
\rhatch\mfobj{pole3}
\endmfpic
```

16. Definice příkazů

Při otevření nového okolí \mfpic ... \endmfpic jsou vždy znovu nadefinovány všechny příkazy kreslení. Pokud bychom chtěli některý z těchto příkazů předefinovat, musíme to tedy provádět pouze uvnitř okolí \mfpic ... \endmfpic. Ovšem například změna velikosti pera mezi jednotlivými okolími mfpic se v obrázcích projeví.

17. Složitější obrázky

Chceme-li tvořit složitější obrázky, budeme zřejmě potřebovat znát kromě příkazů mfpic některé příkazy samotného METAPOSTu a navíc nám mohou posloužit i makra souboru grafbase.mp.

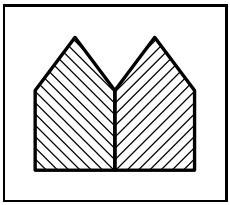
Příkazem

\mfsrc {*příkazy*_METAPOSTu}

zapisujeme *příkazy*_METAPOSTu přímo do výstupního souboru .mp.

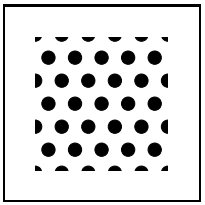
Nadefinujeme-li si vlastní transformaci (přímo v METAPOSTu, musí být v uživatelských souřadnicích), aplikujeme ji pomocí příkazu

\applyT {transformed *transformace*} .



```
\mfpic[10]{-3}{3}{0}{5}
\mfsrc{transform tr;
(1,0) transformed tr = (-1,0);
(2,0) transformed tr = (-2,0);
(1.5,1) transformed tr = (-1.5,1);}
\pen{1.3pt}
\draw\rhatch\polygon{(0,0),(3,0),
                (3,3),(1.5,5),(0,3)}
\draw\lhatch\applyT{transformed tr}
       \polygon{(0,0),(3,0),(3,3),
                     (1.5,5),(0,3)}
\endmfpic
```

Využitím makra `image`, o kterém jsem se zmínila v první části, obrázek přímo nevykreslíme, jen ho uložíme do proměnné typu `picture`:



```
\def\uschovej#1#2{%
\mfsrc{picture #1; #1 = image(}%
#2%
\mfsrc{);}}%
\mfpic[10]{0}{5}{0}{5}
\uschovej{mujobr}{
    \polkadot\rect{(0,0),(5,5)}}
\mfsrc{draw mujobr;}
\endmfpic
```

Pokud bychom požadovali vykreslit jen část obrázku „oříznutou“ uzavřenou křivkou, lze využít makro ze souboru `grafbase.mp`

`clipto` (*proměnná_typu_picture*) *křivka* ,

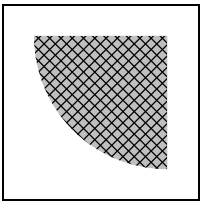
kde *proměnná_typu_picture* je obrázek, který budeme „ořezávat“ křivkou *křivka*. Velmi podobný je příkaz

`clipped` (*proměnná_typu_picture*) *křivka* ,

jen s tím rozdílem, že výstupem je oříznutý obrázek, jenž musí být přiřazen proměnné typu `picture`. *Křivku* je možné vytvořit příkazem \`store` (viz strana 79); je jen vhodné něco vědět o souřadných systémech, se kterými pracujeme.

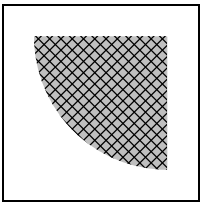
Balík `mfpic` pracuje interně v souřadném systému – device coordinates – odlišném od uživatelského souřadného systému – graph coordinates. Převod z uživatelského systému do interního udává afinní transformace nazvaná `zconv` (inverzní transformací je `invzconv`). Lineární zobrazení indukované touto afinní transformací se nazývá `vconv` (inverzní potom `invvconv`). Například
\`lines{(1,2),(3,-2)}`
je ekvivalentní \`mfsrc{draw zconv((1,2)) -- zconv((3,-2));}`.

Pro nás je tedy důležité, že křivka uložená pomocí makra \store je v uživatelských souřadnicích, avšak makra clipto a clipped požadují argument v interních souřadnicích.



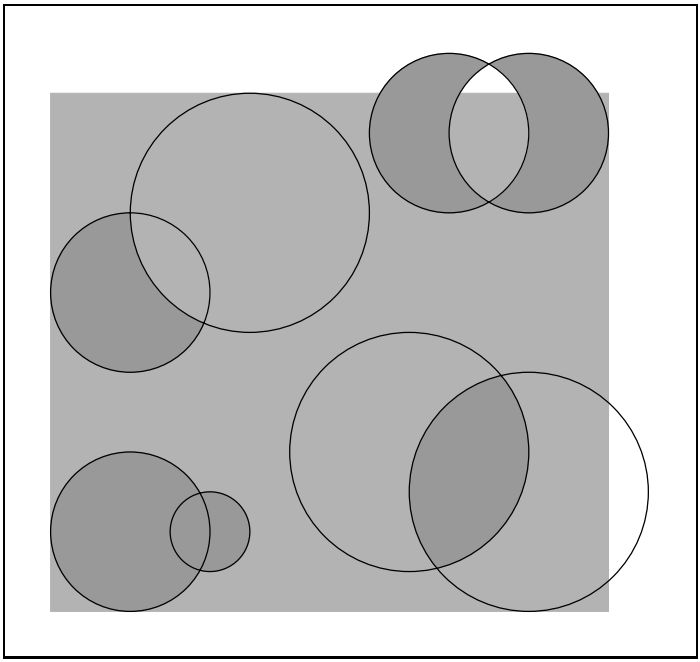
```
\def\uschovej#1#2{%
\mfsrc{picture #1; #1 = image(}%
#2%
\mfsrc{);}}%
\mfpic[10]{0}{5}{0}{5}
\uschovej{dalsiobr}{
\xhatch\shade\rect{(0,0),(5,5)}}
\store{krivka}{\circle{(5,5),5}}
\mfsrc{clipto (dalsiobr)
       zconv(krivka);
       draw dalsiobr;}
\endmfpic
```

Shodný výsledek dává i následující zdrojový text.



```
\def\uschovej#1#2{%
\mfsrc{picture #1; #1 = image(}%
#2%
\mfsrc{);}}%
\mfpic[10]{0}{5}{0}{5}
\uschovej{dalsiobr}{
\xhatch\shade\rect{(0,0),(5,5)}}
\mfsrc{picture o;
         o~= clipped (dalsiobr)
          fullcircle scaled 100
               shifted (50,50);
                       draw o;}
\endmfpic
```

Pro kreslení průniku a rozdílu dvou množin také použijeme zmíněná makra. Proměnná active_plane je proměnná METAPOSTu typu picture definovaná v grafbase.mp a ukládá se do ní postupně vše námi nakreslené. Má podobnou úlohu jako v METAPOSTu proměnná currentpicture. Dále si v definici příkazu \rozdil všimněte, že aby šedé pozadí nebylo překryto (barvy v PostScriptu jsou tzv. krycí, tedy pozdější barva překryje předchozí), musela se část pozadí (uloženého v active_plane) „uschovat“ do proměnné pom_iii a na závěr se zase nakreslila. Jinak by zde vznikl „otvor“ v barvě background (ta se používá v příkazu undraw pro mazání).



```
\def\uschovej#1#2{%
 \mfsrc{picture #1; #1 = image(}%
 #2%
 \mfsrc{);}}%
\def\sjednoceni#1#2{%
\gfill#1\gfill#2}%
\def\prunik#1#2{%
 \uschovej{mnozina_i}{\gfill #1}
 \store{krivka_i}{#2}
 \mfsrc{clipto (mnozina_i) zconv(krivka_i);
 draw mnozina_i;}}%
\def\rozdil#1#2{%
 \uschovej{mnozina_i}{\gfill #1}
 \store{krivka_i}{#1}
 \store{krivka_ii}{#2}
 \mfsrc{picture mnozina_ii;
 mnozina_ii=clipped (mnozina_i) zconv(krivka_ii);
 mnozina_ii:=image(draw mnozina_i; undraw mnozina_ii;);}
 \mfsrc{picture mnozina_iii; mnozina_iii=clipped (active_plane)
 zconv(krivka_i); clipto (mnozina_iii) zconv(krivka_ii);}
 \mfsrc{draw mnozina_ii; draw mnozina_iii;}}%
```

```
\def\symrozdil#1#2{%
 \rozdil{#1}{#2}
 \rozdil{#2}{#1}}%
\mfpic[30]{0}{7.5}{0}{7}
\gfill[.7white]\rect{(0,0),(7,6.5)}
\fillcolor{.6white}
\sjednoceni{\circle{(1,1),1}}{\circle{(2,1),.5}}
\circle{(1,1),1}
\circle{(2,1),.5}
\prunik{\circle{(4.5,2),1.5}}{\circle{(6,1.5),1.5}}
\circle{(4.5,2),1.5}
\circle{(6,1.5),1.5}
\rozdil{\circle{(1,4),1}}{\circle{(2.5,5),1.5}}
\circle{(1,4),1}
\circle{(2.5,5),1.5}
\symrozdil{\circle{(6,6),1}}{\circle{(5,6),1}}
\circle{(6,6),1}
\circle{(5,6),1}
\endmfpic
```

Kopretina na titulní straně této práce také využívá průniky množin. Autorem tohoto obrázku je doc. J. Kuben a zdrojový text vypadá takto (komentářem je barevná varianta):

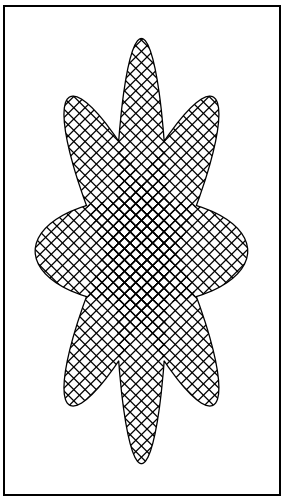
```
\mfpic[7]{-20}{20}{-20}{20}
\newcommand{\obraz}[2]{%
\mfsrc{picture #1; #1=image(}%
#2%
\mfsrc{);}}
\newcommand{\rozdil}[2]{%
\obraz{pom_i}{\gfill #1}%
\store{krivka_i}{#1}    \store{krivka_ii}{#2}%
\mfsrc{picture pom_ii; pom_ii=clipped (pom_i) zconv(krivka_ii);
pom_ii:=image(draw pom_i; undraw pom_ii;);}%
\mfsrc{picture pom_iii;
pom_iii=clipped (active_plane) zconv(krivka_i);
clipto (pom_iii) zconv(krivka_ii);}%
\mfsrc{draw pom_ii; draw pom_iii;}}
\newcommand{\symrozdil}[2]{\rozdil{#1}{#2}\rozdil{#2}{#1}}
\gfill[.6white]\circle{(0,0),12}  %\gfill[green]\circle{(0,0),12}
\gfill\circle{(0,0),4}            %\gfill[red]\circle{(0,0),4}
\store{ki}{\bclosed\plrfcn{0,360,1.25}{12+8*cosd(8t)}}
\store{kii}{\bclosed\plrfcn{0,360,1.25}{12-8*cosd(8t)}}
```

```
\fillcolor{.7white}                %\fillcolor{yellow}
\symrozdil{\mfobj{ki}}{\mfobj{kii}}
%\drawcolor[named]{OrangeRed}
\mfobj{ki}\mfobj{kii}
\endmfpic
```

Křivky v interních souřadnicích jsou argumenty dalšího příkazu – `clipsto`:

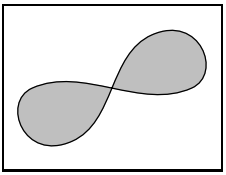
`clipsto` (*proměnná_typu_picture*) (*pole_uzavřených_křivek*) ,

který ořeže *proměnnou_typu_picture* na sjednocení vnitřků všech křivek, které jsou obsaženy v *poli_uzavřených_křivek*.

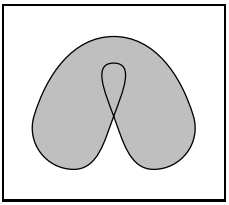


```
\def\uschovej#1#2{%
 \mfsrc{picture #1; #1 = image(}%
 #2%
 \mfsrc{);}}%
\mfpic[10][20]{-4}{4}{-4}{4}
\uschovej{ctverec}{
    \xhatch\rect{(-4,-4),(4,4)}}
\patharr{pole}
\ellipse{(0,0),4,1}
\ellipse[45]{(0,0),4,1}
\ellipse[90]{(0,0),4,1}
\ellipse[135]{(0,0),4,1}
\endpatharr
\mfsrc{for i=1 upto pole:
pole[i]:=zconv(pole[i]); endfor;
         clipsto (ctverec,pole);
                  draw ctverec;}
\mfsrc{path pompole[];
for i=1 upto pole:
pompole[i]:=
(subpath (-.25*length(pole[i]),
.25*length(pole[i])) of pole[i]);
endfor;
for i=1 upto pole:
pompole[i+4]:=
(subpath (.25*length(pole[i]),
.75*length(pole[i])) of pole[i]);
endfor;
draw (buildcycle(pompole1 for i=2
upto 2pole: ,pompole[i] endfor));}
\endmfpic
```

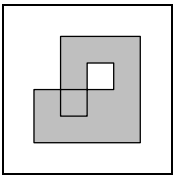
Na závěr se zmíníme o vybarvování uzavřených křivek. PostScript má dva způsoby, jak vyplňuje uzavřené křivky. METAPOST používá ten, který vybarví daný bod, pokud počet oběhů křivky kolem tohoto bodu (tzv. index bodu vzhledem ke křivce) je nenulový. Orientace po směru nebo proti směru hodinových ručiček (tedy zda je index kladný nebo záporný) nehraje roli. Situaci ilustrují následující obrázky.



```
\mfpic[10]{-3.5}{3.5}{-2.5}{2.5}
\draw\gfill[.75white]
\cyclic{(-3,0),(-1.5,-2),(1.5,2),(3,0)}
\endmfpic
```



```
\mfpic[10]{-3}{3}{-2}{3}
\draw\gfill[.75white]
\cyclic{(-3,0),(-1.5,-2),(0,0),
(0,2),(0,0),(1.5,-2),(3,0),(0,3)}
\endmfpic
```



```
\mfpic[10]{0}{4}{0}{4}
\draw\gfill[.75white]
\polygon{(0,0),(4,0),(4,4),(1,4),
(1,1),(2,1),(2,3),(3,3),(3,2),(0,2)}
\endmfpic
```

18. Použití `mfpic` při tvorbě obrázků

V další části mé práce bych ráda na několika obrázcích ze středoškolské tematiky prakticky demonstrovala využití balíku `mfpic`.

Obrázky jsou umístěny záměrně tak, aby je bylo možno porovnávat se zdrojovým kódem.

Ve většině obrázků je použito makro

\mfsrc {*příkazy* METAPOSTu} ,

pomocí něhož vpisujeme do výstupního souboru `.mp` libovolné příkazy METAPOSTu.

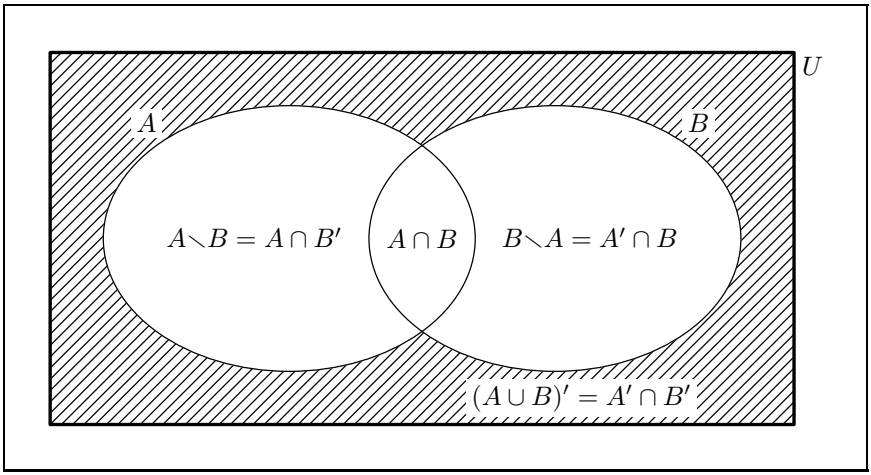
U dvou obrázků je ukázáno použití příkazu

`\mfpverbtex` {*příkazy* TEXu} ,
který zapíše *příkazy* TEXu ohraničené příkazy `verbatimtex` ... `etex` do výstupního souboru `.mp` (viz první část práce).

18.1. Grafické znázornění množin

Obrázek zachycuje Vennův diagram znázorňující obecnou polohu dvou množin.

Je zde naznačeno využití příkazu `\mfpverbtex`. Pokud by byla skutečně použita jeho zakomentovaná forma, znak rozdílu množin by byl vysázen LATEXem, což je v tomto případě vhodné. Samozřejmě je ještě třeba upravit soubor `makempx` (viz první část práce). Znak rozdílu množin lze také vysázet příkazem `\setminus` (nemusíme nic upravovat, protože plain TEX ho zná), ale výsledek není ideální. Třetí možností je nadefinování vlasního symbolu. Makro `\obd` vytvoří pod popiskou bílý obdélník.



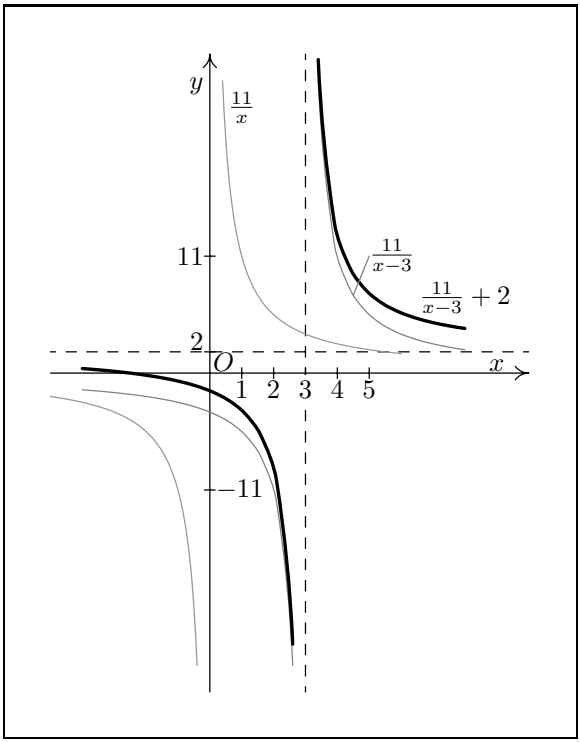
```
%\mfpverbtex{\documentclass{article}
%             \usepackage{amsmath,amsfonts,amssymb}
%             \begin{document}}
\mfpverbtex{%
\def\smallsetminus{{\font\amsb=msbm10
   \mathbin{\hbox{\amsb\char"72}}}}}%
\def\obd#1{%
\mfsrc{picture obr; obr = image(}%
\tlabel#1%
\mfsrc{); unfill bbox (obr); draw obr; }%
}%
\mfpic[10]{-14}{15}{-7}{7}
\pen{1.3pt}
\rect{(-14,-7),(14,7)}
\pen{.5pt}
\rhatch\rect{(-14,-7),(14,7)}
\gfill[white]\ellipse{(5,0),7,5}
\draw\gfill[white]\ellipse{(-5,0),7,5}
\ellipse{(5,0),7,5}
\tlabel[cl](14.3,6.5){$U$}
\obd{[br](-10,4){$A$}}
\obd{[bl](10,4){$B$}}
\obd{[cc](6,-6){$(A\cup B)'=A' \cap B'$}}
\tlabel[cc](0,0){$A\cap B$}
%\tlabel[cl](3,0){$B\setminus A=A' \cap B$}
\tlabel[cl](3,0){$B\smallsetminus A=A' \cap B$}
%\tlabel[cr](-3,0){$A\setminus B=A \cap B'$}
\tlabel[cr](-3,0){$A\smallsetminus B=A \cap B'$}
\endmfpic
```

18.2. Grafy funkcí

Graf funkce $g\colon y = \frac{2x+5}{x-3}$.

U tohoto obrázku si znovu ukážeme praktické použití příkazu `\mfpverbtex`; pokud by byl skutečně využit, mohli bychom požadované popisky vysázet příkazem `\frac`.

Makro `\obd` nám zajistí, aby se popiska bodu na ose x nepřekrývala s přerušovanou čarou naznačující posun osy y při konstrukci funkce.

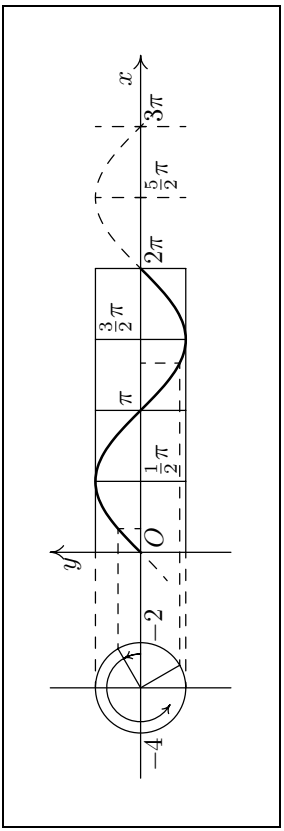



```
%\mfpverbtex{\documentclass{article}
%             \begin{document}}
 %%% makro pro umístění textu na vybarvenou plochu
\def\obd#1{%
\mfsrc{picture obr; obr = image(}%
\tlabel#1%
\mfsrc{); unfill bbox (obr); draw obr; }%
}%
```

```
\mfpic[12][4]{-5}{10}{-30}{30}
\axes
\xmarks{1,2,3,4,5}
\ymarks{-11,2,11}
\tlabel[cr](-.2,27){$y$}
\tlabel[bc](9,.2){$x$}
\tlabel[bl](.1,.2){$0$}
\dashed\lines{(-5,2),(10,2)}
\dashed\lines{(3,-30),(3,30)}
\draw[.6white]\function{-5,-.4,.1}{11/x}
\draw[.6white]\function{.4,6,.1}{11/x}
\draw[.5white]\function{-4,2.6,.5}{11/(x-3)}
\draw[.5white]\function{3.4,8,.5}{11/(x-3)}
\pen{1.3pt}
\function{-4,2.6,.5}{(11/(x-3))+2}
\function{3.4,8,.5}{(11/(x-3))+2}
\pen{.5pt}
\tlabel[cr](-.2,11){11}
\tlabel[cl](.2,-11){$-11$}
\tlabel[br](-.2,2.2){2}
\tlabel[bc](1,-2.3){1}
\tlabel[bc](2,-2.3){2}
\obd{[bc](3,-2.3){3}}
\tlabel[bc](4,-2.3){4}
\tlabel[bc](5,-2.3){5}
\draw[.5white]\lines{(4.5,11/1.5),(5,11)}
%\tlabel[cc](1,25){$\frac{11}{x}$}
%\tlabel[cl](5,11){$\frac{11}{x-3}$}
%\tlabel[cc](8,7){$\frac{11}{x-3}+2$}
\tlabel[cc](1,25){$11\over x$}
\tlabel[cl](5,11){$11\over {x-3}$}
\tlabel[cc](8,7){${11\over {x-3}}+2$}
\endmfpic
```

Graf funkce sinus.

Chceme-li obrázek otočit o devadesát stupňů, lze to provést až po nakreslení celého obrázku, nebo můžeme na úvod zapsat příkaz \rotate{90} a potom ještě otáčet všechny popisky (tento způsob je zakomentovaný a otočení popisek naznačeno u první z nich).

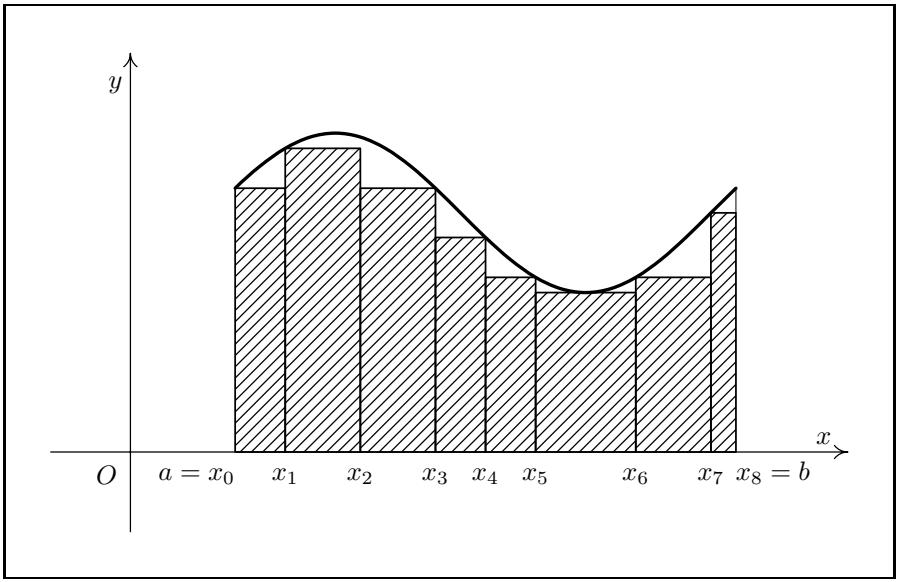



```
\mfpic[17]{-5}{11}{-2}{2}
%\rotate{90}
\axes
\xmarks{(5/2)*pi,3*pi}
%\tlabel[cr 90](-.1,1.5){$y$}
\tlabel[cr](-.1,1.5){$y$}
\tlabel[bc](10.5,.2){$x$}
```

```
\lines{(-3,-2),(-3,2)}
\lines{(0,1),(2*pi,1)}
\lines{(0,-1),(2*pi,-1)}
\lines{(.5*pi,-1),(.5*pi,1)}
\lines{(pi,-1),(pi,1)}
\lines{((3/2)*pi,-1),((3/2)*pi,1)}
\lines{(2*pi,-1),(2*pi,1)}
\dashed\lines{((5/2)*pi,-1),((5/2)*pi,1)}
\dashed\lines{(3*pi,-1),(3*pi,1)}
\fdef{f}{x}{sin x}
\dashed\function{-pi/5,3.1*pi,.1*pi}{f(x)}
\pen{1pt}
\function{0,2*pi,.1*pi}{f(x)}
\pen{.5pt}
\circle{(-3,0),1}
\arrow[r-5]\arc[p]{(-3,0),0,30,.75}
\arrow[r -5]\arc[p]{(-3,0),0,240,.75}
\lines{(-3,0),(-3+cosd 30,sind 30)}
\lines{(-3,0),(-3-cosd 240,sind 240)}
\dashed\lines{(-3,1),(0,1)}
\dashed\lines{(-3,-1),(0,-1)}
\dashed\lines{(-3+cosd 30,sind 30),(pi/6,f(pi/6))}
\dashed\lines{(pi/6,0),(pi/6,f(pi/6))}
\dashed\lines{(-3-cosd 240,sind 240),((4/3)*pi,f((4/3)*pi))}
\dashed\lines{((4/3)*pi,0),((4/3)*pi,f((4/3)*pi))}
\tlabel[tr](-4.1,-.1){$-4$}
\tlabel[tl](-2,-.1){$-2$}
\tlabel[tl](.1,-.1){$0$}
\tlabel[tl]((.5*pi)+.1,-.1){${1\over 2} \pi$}
\tlabel[bl](1*pi+.1,0.2){$\pi$}
\tlabel[bl]((3/2)*pi+.1,.2){${3\over 2} \pi$}
\tlabel[tl](2*pi+.1,-.1){$2\pi$}
\tlabel[tl]((5/2)*pi+.1,-.1){${5\over 2} \pi$}
\tlabel[tl](3*pi+.1,-.1){$3\pi$}
\mfsrc{picture a; a~:= currentpicture;
          currentpicture:=nullpicture;
                    draw a~rotated 90;}%
\endmfpic
```

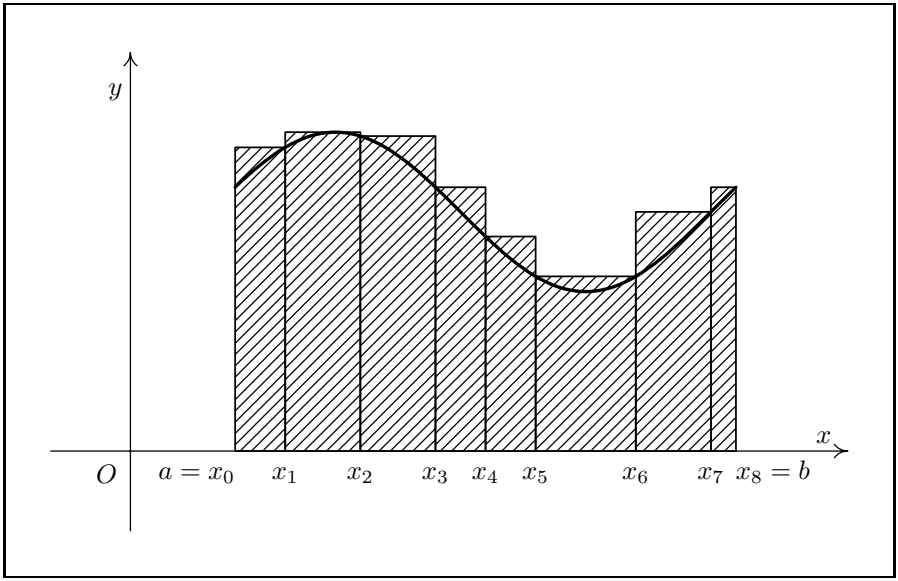
18.3. Určitý integrál a jeho geometrické aplikace

Znázornění dolního integrálního součtu příslušného určitému dělení intervalu $\langle a, b\rangle$.



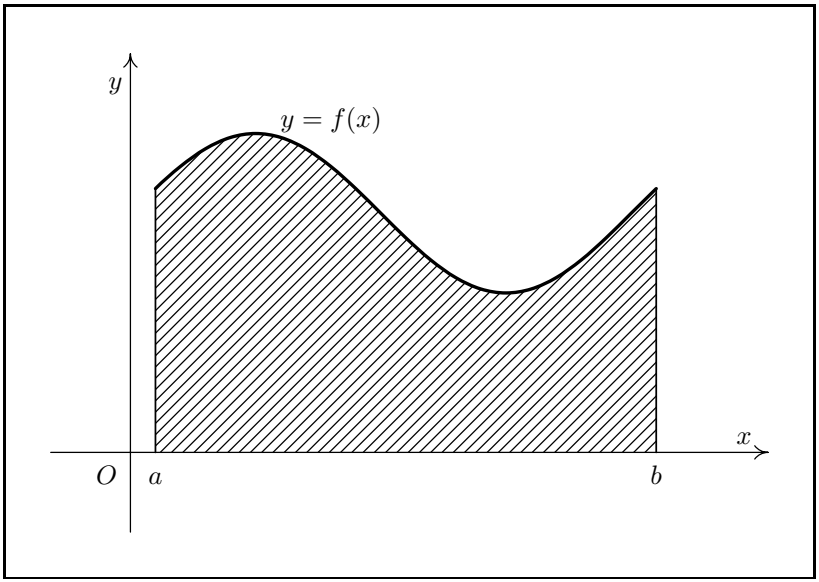
```
\mfpicunit=30pt
\mfpic[1]{-2}{8}{-1}{5}
\xaxis
\headlen=5pt                  %%% musíme změnit přednastavenou
\arrow\lines{(-1,-1),(-1,5)} %%% délku, neboť je jiná než
\tlabel[cr](-1.1,4.6){$y$}   %%% u makra \xaxis; nebo lze
\tlabel[bc](7.7,0.1){$x$}    %%% \arrow[l 5pt]\lines{...}
\tlabel[bc](-1.3,-0.4){$0$}
\fdef{f}{x}{(sin x)+3}
\pen{1.3pt}
\function{.1*pi,2.1*pi,.1*pi}{f(x)}
\pen{.7pt}
\draw\rhatch\rect{(.1*pi,f(.1*pi)),(.3*pi,0)}
\draw\rhatch\rect{(.3*pi,f(.3*pi)),(.6*pi,0)}
\draw\rhatch\rect{(.6*pi,f(.9*pi)),(.9*pi,0)}
\draw\rhatch\rect{(.9*pi,f(1.1*pi)),(1.1*pi,0)}
\draw\rhatch\rect{(1.1*pi,f(1.3*pi)),(1.3*pi,0)}
\draw\rhatch\rect{(1.3*pi,f(1.5*pi)),(1.7*pi,0)}
\draw\rhatch\rect{(1.7*pi,f(1.7*pi)),(2*pi,0)}
\draw\rhatch\rect{(2*pi,f(2*pi)),(2.1*pi,0)}
\pen{.3pt}
\lines{(2.1*pi,f(2*pi)),(2.1*pi,f(2.1*pi))}
\tlabel[br](.1*pi,-0.4){$a=x_0$}
\tlabel[bc](.3*pi,-0.4){$x_1$}
\tlabel[bc](.6*pi,-0.4){$x_2$}
\tlabel[bc](.9*pi,-0.4){$x_3$}
\tlabel[bc](1.1*pi,-0.4){$x_4$}
\tlabel[bc](1.3*pi,-0.4){$x_5$}
\tlabel[bc](1.7*pi,-0.4){$x_6$}
\tlabel[bc](2*pi,-0.4){$x_7$}
\tlabel[bl](2.1*pi,-0.4){$x_8=b$}
\endmfpic
```

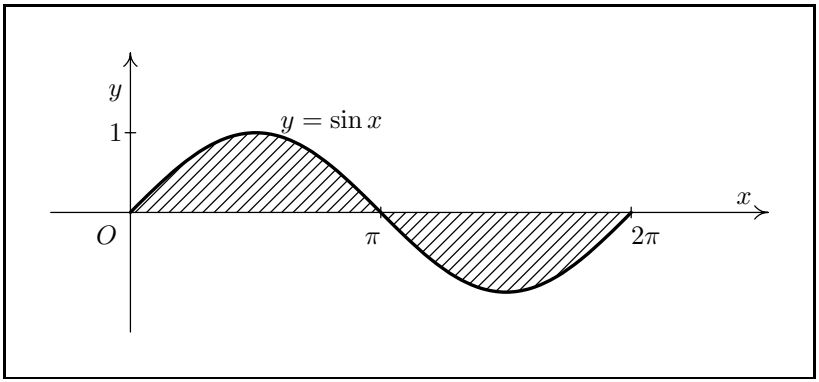
Znázornění horního integrálního součtu příslušného určitému dělení intervalu $\langle a, b\rangle$.



```
\mfpic[1]{-2}{8}{-1}{5}
\mfpicunit=30pt
\xaxis
\headlen=5pt                  %%% pro dosažení stejné délky
\arrow\lines{(-1,-1),(-1,5)} %%% šipky jako v makru \xaxis;
\tlabel[cr](-1.1,4.6){$y$}   %%% nebo lze použít:
\tlabel[bc](7.7,0.1){$x$}    %%% \arrow[l 5pt]\lines{...}
\tlabel[bc](-1.3,-0.4){$0$}
\fdef{f}{x}{(sin x)+3}
\pen{1.3pt}
\function{.1*pi,2.1*pi,.1*pi}{f(x)}
\pen{.7pt}
\draw\rhatch\rect{(.1*pi,f(.3*pi)),(.3*pi,0)}
\draw\rhatch\rect{(.3*pi,f(.5*pi)),(.6*pi,0)}
\draw\rhatch\rect{(.6*pi,f(.6*pi)),(.9*pi,0)}
\draw\rhatch\rect{(.9*pi,f(.9*pi)),(1.1*pi,0)}
\draw\rhatch\rect{(1.1*pi,f(1.1*pi)),(1.3*pi,0)}
\draw\rhatch\rect{(1.3*pi,f(1.7*pi)),(1.7*pi,0)}
\draw\rhatch\rect{(1.7*pi,f(2*pi)),(2*pi,0)}
\draw\rhatch\rect{(2*pi,f(2.1*pi)),(2.1*pi,0)}
\pen{.5pt}
\tlabel[br](.1*pi,-0.4){$a=x_0$}
\tlabel[bc](.3*pi,-0.4){$x_1$}
\tlabel[bc](.6*pi,-0.4){$x_2$}
\tlabel[bc](.9*pi,-0.4){$x_3$}
\tlabel[bc](1.1*pi,-0.4){$x_4$}
\tlabel[bc](1.3*pi,-0.4){$x_5$}
\tlabel[bc](1.7*pi,-0.4){$x_6$}
\tlabel[bc](2*pi,-0.4){$x_7$}
\tlabel[bl](2.1*pi,-0.4){$x_8=b$}
\endmfpic
```

Geometrický význam určitého integrálu $\int_a^b f(x)\,\mathrm{d}x$ funkce f a náčrt obsahu obrazce ohraničeného grafem funkce $f\colon y = \sin x$ v intervalu $\langle 0, 2\pi\rangle$ a osou x.

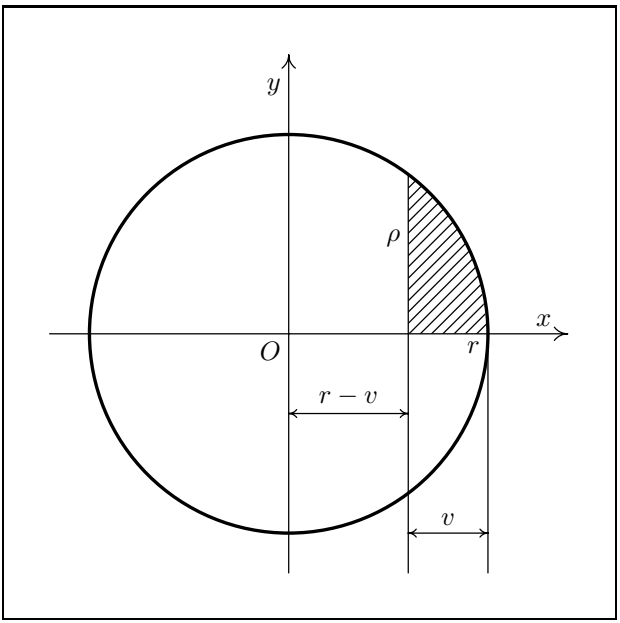




```
\mfpic[1]{-1}{8}{-1}{5}
\mfpicunit=30pt
\axes
\tlabel[cr](-0.1,4.6){$y$}
\tlabel[bc](7.7,0.1){$x$}
\tlabel[bc](-0.1,-0.4){$0$}
\fdef{f}{x}{(sin x)+3}
\pen{1.3pt}
\function{.1*pi,2.1*pi,.1*pi}{f(x)}
\pen{.5pt}
\rhatch\btwnfcn{.1*pi,2.1*pi,.1*pi}{0}{f(x)}
\pen{.7pt}
\lines{(.1*pi,0),(.1*pi,f(.1*pi))}
\lines{(2.1*pi,0),(2.1*pi,f(2.1*pi))}
\tlabel[bl](.6*pi,4){$y=f(x)$}
\tlabel[bc](.1*pi,-0.4){$a$}
\tlabel[bc](2.1*pi,-0.4){$b$}
\endmfpic
```

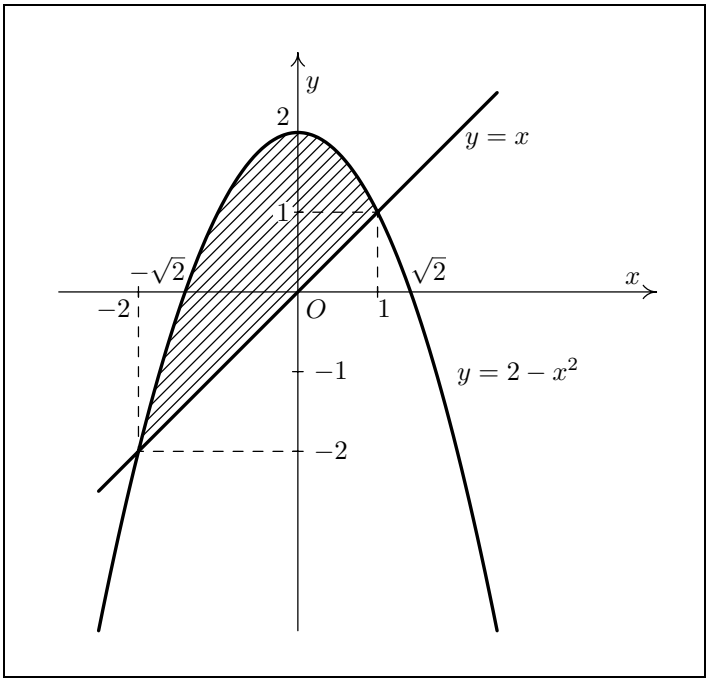
```
\mfpic[1]{-1}{8}{-1.5}{2}
\mfpicunit=30pt
\axes
\ymarks{1}
\xmarks{pi,2*pi}
\tlabel[cr](-0.1,1.5){$y$}
\tlabel[bc](7.7,0.1){$x$}
\tlabel[bc](-.3,-.4){$0$}
\fdef{f}{x}{sin x}
\pen{1.3pt}
\function{0,2*pi,.1*pi}{f(x)}
\pen{.7pt}
\rhatch\btwnfcn{0,2*pi,.1*pi}{0}{f(x)}
\pen{.5pt}
\tlabel[br](pi,-.4){$\pi$}
\tlabel[bl](2*pi,-.4){$2\pi$}
\tlabel[cr](-.1,1){$1$}
\tlabel[bl](.6*pi,1){$y=\sin{x}$}
\endmfpic
```

Náčrt k odvození vzorce pro objem kulové úseče výšky v, která je vyťata z koule o poloměru r.



```
\mfpic[1]{-3}{3.5}{-3}{3.5}
\mfpicunit=30pt
\axes
\tlabel[cr](-.1,3.1){$y$}
\tlabel[bc](3.2,.1){$x$}
\tlabel[tr](-.1,-.1){$O$}
\pen{1.3pt}
\circle{(0,0),2.5}
\pen{.5pt}
\lines{(1.5,sqrt((2.5*2.5)-(1.5*1.5))),(1.5,-3)}
\lines{(2.5,0),(2.5,-3)}
\arrow\reverse\arrow\lines{(0,-1),(1.5,-1)}
\arrow\reverse\arrow\lines{(1.5,-2.5),(2.5,-2.5)}
\pen{.7pt}
\rhatch\btwnfcn{1.5,2.5,.1}{0}{sqrt((2.5*2.5)-(x*x))}
\tlabel[cr](1.4,1.2){$\rho$}
\tlabel[tr](2.4,-.1){$r$}
\tlabel[bc](.75,-.9){$r-v$}
\tlabel[bc](2,-2.4){$v$}
\endmfpic
```

Obsah obrazce ohraničeného grafy funkcí $f\colon y = 2 - x^2$, $g\colon y = x$.

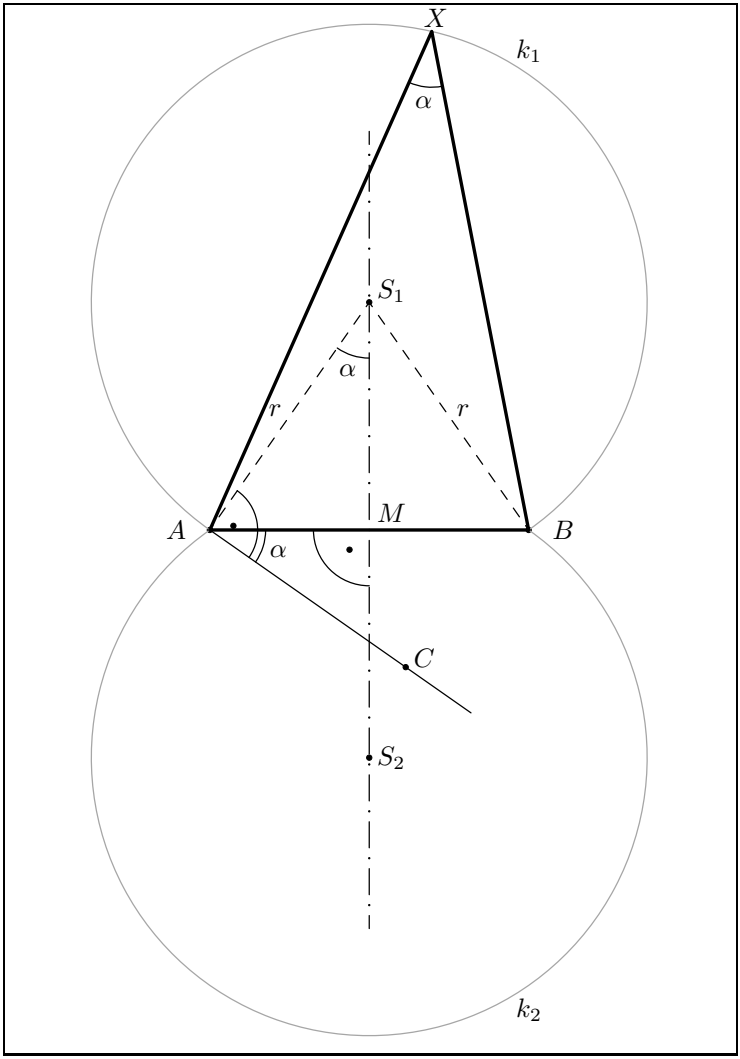


```
\def\obd#1{%                          %%% makro pro odbarvení plochy
\mfsrc{picture obr; obr = image(} %%% pod textem ve tvaru kruhu;
\tlabel#1%                            %%% příkaz center určuje střed
\mfsrc{); unfill fullcircle           %%% bounding boxu
    scaled (2*(abs( center obr -
                  llcorner obr)))
           shifted (center obr);
                      draw obr;}%
\mfpic[1]{-3}{4.5}{-4.25}{3}
\mfpicunit=30pt
\axes
\xmarks{-2,1}
\ymarks{-2,-1,1}
\tlabel[cl](0.1,2.6){$y$}
\tlabel[bc](4.2,.1){$x$}
\tlabel[tl](.1,-.1){$0$}
\fdef{f}{x}{x}
\fdef{g}{x}{2-(x*x)}
\pen{1.3pt}
\function{-2.5,2.5,.1}{f(x)}
\function{-2.5,2.5,.1}{g(x)}
\pen{.7pt}
\rhatch\btwnfcn{-2,1,.1}{g(x)}{f(x)}
\pen{.5pt}
\dashed\lines{(-2,0),(-2,f(-2))}
\dashed\lines{(0,-2),(-2,f(-2))}
\dashed\lines{(1,0),(1,f(1))}
\dashed\lines{(0,1),(1,f(1))}
\tlabel[tl](2.1,2){$y=x$}
\tlabel[cl](2,-1){$y=2-x^2$}
\tlabel[tr](-2.1,-.1){$-2$}
\tlabel[br](-sqrt 2,.1){$-\sqrt{2}$}
\tlabel[tl](1,-.1){1}
\tlabel[bl](sqrt 2,.1){$\sqrt{2}$}
\tlabel[cl](0.2,-2){$-2$}
\tlabel[cl](0.2,-1){$-1$}
\obd{[cr](-.1,1){1}}
\tlabel[cr](-.1,2.2){2}
\endmfpic
```

18.4. Množiny bodů dané vlastnosti

Množina všech bodů X, z nichž vidíme úsečku AB pod úhlem α.

U tohoto obrázku si všimněme zejména vyznačování úhlů a tvorby čerchované čáry.




```
\mfpicunit=30pt
\mfpic[1]{-4}{4}{-6}{6}
\mfsrc{pair A,B,C,D,S,O,X,Y,Z;
A=(-2,0); B=(2,0);      alpha:=35;
```

```
C=3*dir(-alpha)+A;      D=4*dir(-alpha)+A;
S=A+whatever*dir(90-alpha);    S=whatever*up;    O=-S;}
\lines{A,B}     \lines{A,D}     \point{A,B,C,S,O}
\dashed\lines{A,S}   \dashed\lines{B,S}
\dashpattern{cerchovana}{10pt,4pt,0pt,4pt}
\gendashed{cerchovana}\lines{(0,-5),(0,5)}
\point{.3*dir(45-alpha)+A}
\arc[p]{A,-alpha,90-alpha,.6}    \arc[p]{A,-alpha,0,.7}
\point{(.35*dir 225)}
\arc[p]{.5[A,B],180,270,.7} \arc[p]{S,270-alpha,270,.7}
\draw[.65white]\arc[p]{S,alpha-90,270-alpha,abs(S-A)}
\coords     \mirror{A}{B}
\draw[.65white]\arc[p]{S,alpha-90,270-alpha,abs(S-A)}
\endcoords
\mfsrc{X=S+abs(S-A)*dir(77);}
\pen{1.3pt}     \polygon{A,B,X}     \pen{.5pt}
%\mfsrc{path a;   %%% vyznačení úhlu lze nakreslit také takto
%a=fullcircle scaled 42pt shifted zconv (X);
%draw subpath (ypart (zconv ((X--A)) intersectiontimes a),
%ypart(zconv((X--B)) intersectiontimes a)) of a;}%
\mfsrc{x=angle(X-A); y=angle(X-B);}
\arc[p]{X,x+180,y+180,.7}
\tlabel[cr](xpart A~-.3,ypart A){$A$}
\tlabel[cl](xpart B +.3,ypart B){$B$}
\tlabel[bl](xpart C +.1,ypart C){$C$}
\tlabel[bl](xpart S~+.1,ypart S){$S_1$}
\tlabel[cl](xpart O~+.1,ypart O){$S_2$}
\tlabel[bl](0.1,0.1){$M$}
\tlabel[bc](xpart X + 0.05,ypart X + 0.05){$X$}
\tlabel[br](xpart .5[A,S] -.1,ypart .5[A,S]){$r$}
\tlabel[bl](xpart .5[B,S] +.1,ypart .5[A,S]){$r$}
\mfsrc{Y=S+abs(S-A)*dir(60);}
\tlabel[bl](xpart Y +.1,ypart Y){$k_1$}
\tlabel[tl](xpart Y +.1,-ypart Y){$k_2$}
\mfsrc{pair f,g,h;f=.9*dir((x+y)/2+180)+X;
                  g=.9*dir(-alpha/2)+A;
                  h=.9*dir(270-(alpha/2))+S;}
\tlabel[cc](xpart f,ypart f){$\alpha$}
\tlabel[cc](xpart g,ypart g){$\alpha$}
\tlabel[cc](xpart h,ypart h){$\alpha$}
\endmfpic
```

Literatura

[1] Adobe Systems Incorporated *PostScript Language Reference Manual*. Massachusetts: Addison-Wesley, 3. vydání, 1999. ISBN 0-201-37922-8 http://adobe.com:80/products/postscript/pdfs/PLRM.pdf

[2] Hobby, J. D. *A User's Manual for MetaPost*. Součást dokumentace programu METAPOST. http://cm.bell-labs.com/cm/cs/cstr/162.ps.gz

[3] Knuth, D. E. *The METAFONTbook*. Massachusetts: Addison-Wesley, 1986. ISBN 0-201-13445-4

[4] Kuben, J. *Diferenciální počet funkcí jedné proměnné*. Brno: VA, 1999

[5] Kuben, J. *Zpravodaj Československého sdružení uživatelů TEXu 2/94*. Brno. http://bulletin.cstug.cz/pdf/bul942.pdf

[6] Olšák, P. *TEXbook naruby*. Brno: Konvoj, 2. vydání, 2001. ISBN 80-7302-007-6 ftp://math.feld.cvut.cz/pub/olsak/tbn/tbn.pdf

[7] Polák, J. *Přehled středoškolské matematiky*. Praha: Prometheus, 2000. ISBN 80-7196-196-5

[8] rybicka Rybička, J. *LATEX pro začátečníky*. Brno: Konvoj, 2. vydání, 1999. ISBN 80-85615-74-6

Zajímavé odkazy týkající se tématu:

- http://cm.bell-labs.com/who/hobby/MetaPost.html
- http://comp.uark.edu/~luecking/tex/mfpic.html
- http://ftp.cstug.cz/pub/tex/CTAN/graphics/mfpic/ (bohužel zatím je zde pouze starší verze)
- http://ftp.cstug.cz/pub/tex/CTAN/systems/knuth/mf/mfbook.tex

Poznámka

Tento článek je věnován `mfpic` verzi 0.4.05, poslední dostupné verzi v době uzávěrky Zpravodaje. Již brzy bude dostupná verze `mfpic` 0.5.0. Nové vlastnosti budou popsány v dodatku článku v následujícím čísle Zpravodaje.

Rejstřík

Summary: METAPOST and mfpic—the second part

Although METAPOST is not hard to learn, most users prefer the `mfpic` macro package which can be used in plain, LaTeX and pdfTeX. This article presents almost all features of this macro package in examples (full source code for each is included).

Sazba matematiky v českých textech

Karel Horák

Sazba matematiky v českém (a většinou i evropském) prostředí vykazuje některé odlišnosti od standardu AMS, který je (v podstatě) implementován v algoritmech TEXu.

Kromě jiných názvů některých trigonometrických (a hyperbolických) funkcí jako tg, cotg, tgh, cotgh, arctg, arccotg, nebo operátorů daných zkratkou jako třeba nsd (největší společný dělitel, v angličtině gcd) a nsn (nejmenší společný násobek; školská matematika však používá i funkce $n(.,.)$ a $D(.,.)$, zatímco symbolika využívající jen závorek, vyskytující se běžně např. v Rychlíkových monografiích, se dnes prakticky nepoužívá) se v české sazbě standardně objevují i některé značky, které zejména v anglosaské matematické literatuře nenajdeme, nicméně některé z nich jako např. $\leqq$, resp. $\geqq$ najdeme v doplňkových fontech $\mathcal{AMS}$-TEXu (`msam10`).

Nebudu tu podrobně rozvádět, jak se zavádí nový font do matematiky, to je dostatečně popsáno jinde (doporučuji zejména [TBN]), takže jen stručně:

```
\newfam\safam
\font\tensa=msam10
\font\sevensa=msam7
\font\fivesa=msam5
\textfont\safam=\tensa\scriptfont\safam=\sevensa
  \scriptscriptfont\safam=\fivesa
\mathchardef\le="3835
\mathchardef\ge="383D
```

Za předpokladu, že jako první novou rodinu fontů zavedeme právě zmíněné značky v `msam10`, přidělí jí TEX číslo 8, o čemž najdeme hlášku ve výpise o průběhu kompilace: `\safam=\fam8`. Samozřejmě se dá přiřazené číslo zjistit prostředky TEXu, to ale není obsahem tohoto stručného příspěvku.

Zmíněné dvě značky pro tzv. neostré nerovnosti patří asi mezi nejpoužívanější. V uvedené sadě (viz tabulky na konci článku) však najdeme i některé další, např. $\eqslantless$, $\eqslantgtr$. Procházíme-li seznam značek ať už těch, jež se používají ve středoškolské matematice, nebo těch, jež byly k dispozici v klasické sazárně podle Wickovy knihy [W], zjistíme, že sady písem, jež má TEX k dispozici, spolu s jeho programovacími (konstrukčními) možnostmi jsou dostatečně bohaté na sazbu čehokoli. Chybí snad jen značka pro úhel $\sphericalangle$ hojně používaná zejména ve středoškolských

učebnicích, řidčeji pak značka pro nekonvexní úhel ∢. První se dá velmi dobře zkonstruovat jako

```
\def\uhel{{{}\!\!<\nobreak\kern-5pt)\,}}
```

nicméně časem jsem se odhodlal k tomu, že jsem dal dohromady malou sadu pěti znaků `cmnsy10` obsahující kromě těchto dvou značek ještě varianty znaků „mnohem menší (větší)“, jež mají být dle naší normy useknuté, a znak proporcionality otočený na druhou stranu. Všechny využívají jen nepatrně pozměněný METAFONTový kód z Computer Modern. Z podobného důvodu jsem kdysi vytvořil variantu `cmnsy10` s otevřenějšími lomenými závorkami, protože to byl požadavek jednoho kolegy matematika, kterému se původní závorky zdály příliš „krotké“. Víckrát jsem však tuto variantu již nepoužil. Tyto závorky se u nás dosud hojně využívaly k označování uzavřených intervalů, pod vlivem norem platných v Evropské unii se však v poslední době přechází na používání hranatých závorek např. ve tvaru $]0,1]$ místo obvyklého $(0,1\rangle$. To přináší tu nepříjemnost, že v takovýchto případech nedokáže TEX sám poznat, že levá závorka byla otevírací…

Budeme-li pečlivě porovnávat kresbu jednotlivých znaků Computer Modern se znaky v české matematické literatuře, jistě najdeme řadu nepříliš nápadných odlišností (mně např. připadají hodně úzké hranaté závorky v základní velikosti), ale mnohem spíš brzy zjistíme větší nejednotnost v kresbě jednotlivých znaků nejen mezi dříve sázenými publikacemi navzájem, ale i uvnitř jediné publikace (rozličné velikosti rovnítek, poskakující minusy apod.) A to nemluvím o „sazbě“ matematiky v různých textových editorech, jejichž uživatelé nejen že nemají tušení o různých pomlčkách, ale ani o minusu… Ale proti takovým uživatelům samozřejmě není imunní ani tak dokonalý program jako TEX.

Za zmínku snad ještě stojí, že původní typografická norma uvádí jako znak násobení tečku na účaří, zatímco většina uživatelů TEXu běžně používá `\cdot`, jež je stejně jako ostatní značky relací a operací na společné („matematické“) ose. Tato značka se navíc nikdy neplete s tečkou interpunkční. To jsou podstatné argumenty pro to, abychom prosazovali její používání i v naší praxi, a nutno dodat, že jsem se už dávno nesetkal s odmítavou reakcí. Pokud se z nějakého důvodu chceme přidržet původní praxe, je potřeba dbát, aby násobicí tečka měla charakter operace, nikoli interpunkce (`\mathop`, nikoli `\mathpunct`).

Opakování značek relací a operací při zlomu vzorečků na řádce patří asi mezi rozdíly zásadní. Myslím, že toto pravidlo výrazně přispívá k čitelnosti matematického textu. Značky se opakují i při vysazení vzorců do více samostatných řádků, zatímco anglosaská typografie značky na koncích řádků naopak vynechává.

Idea je ostatně popsána i v [TBN] (str. 160). Soubor `opakuj.tex`, jehož podstatnou část uvádím dále, vznikl už v době, kdy jsem začal poprvé pronikat

do tajů TEXu poněkud hlouběji. Jakmile jsem se propracoval k aktivním znakům v matematickém módu, bylo vyhráno, protože to bylo ta správná cesta — všechna „ruční“ řešení byla nevyhovující a navíc nebezpečná svými důsledky při nějakém náhlém přeformátování.

Vlastní makra pak během jednoho velmi dlouhého večera sepsal tehdy ještě TEXový novic Štěpán Kasal. Šířena jsou samozřejmě volně, nebo spíše samovolně, tj. kdykoli někdo o něco podobného projevil zájem, dostal je. Snad se octla i někde v archivu po mém vystoupení na minulém SLT v Jevíčku. Sám jsem je nikam neposílal, protože jestli něco nemám rád, tak je to psaní dokumentace, pokynů apod. jinak nepochybně užitečných svodných informací. Za celých zhruba dvanáct let je používám takřka denně. Musím však přiznat, že jsem nikdy netestoval jejich chování ve spojení s LATEXem, ale mezi zájemci o jejich používání aspoň jeden uživatel tohoto obskurního formátu byl. Občas je používám i ve velmi redukované podobě při sazbě běžného textu, vyskytují-li se tam občas rovnítka, protože i v takových situacích by se rovnítka měla při zlomu řádky opakovat.

```
\exhyphenpenalty 1000   % opakuj
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% původní význam makra "\NějakáRelace" dostaneme
% jako "\original\NějakáRelace"
% znaky  +, -, :, <, =, >  jsou aktivní pouze ve formulích
% (ani v $\mskip-4pt$, či $\let\v=\in$ by neměly zlobit)

\def\original#1{\csname\string#1\endcsname}

\mathchardef\pl@s="202B
\mathchardef\min@s="2200
\mathchardef\dv@j="303A
\mathchardef\l@ss="313C
\mathchardef\rovn@="303D
\mathchardef\gre@ter="313E
\mathchardef\n@t="3236
\mathchardef\nd="3A2D

\outer\def\opakuj#1{\expandafter\let\csname\string#1\endcsname=#1
   \def#1{\nobreak\csname\string#1\endcsname\nobreak\discretionary
          {}{\hbox{$\csname\string#1\endcsname$}}{}}}

\opakuj\le               \let\leq=\le    \let\leqq=\le
\opakuj\ge               \let\geq=\ge    \let\geqq=\ge
\def\nequiv{\mathrel{\mathpalette\c@ncel{\original\equiv}}}
```

```
\opakuj\nequiv
\def\relbar{\mathrel{\smash\min@s}}  %  \smash, because  -  and  +
\def\Relbar{\mathrel\rovn@}          %     have the same height

\def\hookrightarrow{\lhook\joinrel\original\rightarrow}
\def\hookleftarrow{\original\leftarrow\joinrel\rhook}
\def\Longrightarrow{\Relbar\joinrel\original\Rightarrow}
\def\longrightarrow{\relbar\joinrel\original\rightarrow}
\def\longleftarrow{\original\leftarrow\joinrel\relbar}
\def\Longleftarrow{\original\Leftarrow\joinrel\Relbar}
\def\longleftrightarrow
      {\original\leftarrow\joinrel\original\rightarrow}
\def\Longleftrightarrow
      {\original\Leftarrow\joinrel\original\Rightarrow}
\def\rightarrowfill{$\m@th\mathord-\mkern-6mu%
  \cleaders\hbox{$\mkern-2mu\mathord-\mkern-2mu$}\hfill
  \mkern-6mu\mathord\original\rightarrow$}
\def\leftarrowfill{$\m@th\mathord\original\leftarrow\mkern-6mu%
  \cleaders\hbox{$\mkern-2mu\mathord-\mkern-2mu$}\hfill
  \mkern-6mu\mathord-$}

\opakuj\hookrightarrow
\opakuj\hookleftarrow
\opakuj\Longrightarrow
\opakuj\longrightarrow
\opakuj\longleftarrow
\opakuj\Longleftarrow
\opakuj\longleftrightarrow
\opakuj\Longleftrightarrow

\def\mapsto{\mapstochar\original\rightarrow}  \opakuj\mapsto
\def\longmapsto{\mapstochar\original\longrightarrow}
\opakuj\longmapsto
\opakuj\bowtie
\def\models{\mathrel|\joinrel\rovn@}
\opakuj\models

\def\*{\discretionary{\thinspace\the\textfont2\char2}%
                     {\the\textfont2\char2\thinspace}{}}
\def\iff@{\mskip\thickmuskip\original\Longleftrightarrow
          \mskip\thickmuskip}
\def\iff{\iff@\discretionary{\hbox{$\!\!$}}{\hbox{$\!\iff@$}}{}}
```

```
\def\cong{\mathrel{\mathpalette\@vereq
    {\original\sim}}}% congruence
\def\@vereq#1#2{\lower.5\p@\vbox
    {\baselineskip\z@skip\lineskip-.5\p@
    \ialign{$\m@th#1\hfil##\hfil$\crcr#2\crcr\rovn@\crcr}}}
\opakuj\cong                    % congruence sign
\def\notin{\mathrel{\mathpalette\c@ncel{\original\in}}}
\opakuj\notin
\def\rlh@#1{\vcenter{\hbox{\ooalign{\raise2pt
          \hbox{$#1\original\rightharpoonup$}\crcr
        $#1\original\leftharpoondown$}}}}
\opakuj\rightleftharpoons
\def\doteq{\buildrel\textstyle.\over\rovn@}       \opakuj\doteq


\opakuj\wedge      \let\land=\wedge
\opakuj\vee           \let\lor=\vee
\opakuj\cap
\opakuj\cup

   ⋮ \mathcode`\+="8000%

\mathcode`\-="8000%
\mathcode`\:="8000%
\mathcode`\<="8000%
\mathcode`\=="8000%
\mathcode`\>="8000%
{\catcode`\+=\active
    \gdef+{\pl@s\nobreak\discretionary{}{\hbox{$\pl@s$}}{}}}
{\catcode`\-=\active
    \gdef-{\min@s\nobreak\discretionary{}{\hbox{$\min@s$}}{}}}
{\catcode`\:=\active
    \gdef:{\dv@j\nobreak\discretionary{}{\hbox{$\dv@j$}}{}}}
{\catcode`\<=\active
    \gdef<{\l@ss\nobreak\discretionary{}{\hbox{$\l@ss$}}{}}}
{\catcode`\==\active
    \gdef={\rovn@\nobreak\discretionary{}{\hbox{$\rovn@$}}{}}}
{\catcode`\>=\active
    \gdef>{\gre@ter\nobreak\discretionary{}{\hbox{$\gre@ter$}}{}}}
```

```
\def\not#1{\ifcat=#1\decid@char#1\else
    \n@t\original#1\discretionary
    {}{\hbox{$\n@t\original#1$}}{}\fi}
\def\decid@char#1{\if=#1\ne\else
                    \if<#1\nl\else
                        \if>#1\ng\fi\fi\fi}

\def\nl{\n@t\l@ss\nobreak\discretionary{}{\hbox{$\n@t\l@ss$}}{}}
\def\ne{\n@t\rovn@\nobreak\discretionary{}{\hbox{$\n@t\rovn@$}}{}}
\def\ng{\n@t\gre@ter\nobreak\discretionary
    {}{\hbox{$\n@t\gre@ter$}}{}}
```

Hned zpočátku je nastaveno `\exhyphenpenalty=1000`. Je potřeba mít na paměti, že hodnota této penalty větší či rovná 10 000 způsobí, že se žádný vzoreček nebude v opakovacích verzích relací a binárních operací dělit, protože použité `\discretionary` má první parametr prázdný. Vlastní makro `\opakuj` s jedním parametrem umožňuje, jak je i z uvedené ukázky snad vidět, změnit chování jakékoli nově zavedené značky. Pokud některou značku potřebujeme uchovat i v neopakovací verzi, stačí si uchovat její neopakovací verzi pomocí přiřazení `\let`.

Pokud chceme mít klid a jistotu, že se v textu neobjeví opakovaně nějaké unární minus nebo plus, stačí využít složených závorek (budeme tedy preventivně psát `2x={-1}` nebo `({-\infty,+\infty})`) a skutečnosti, že TEX v matematickém módu skupiny vkládá do atomů jako nedělitelný celek. Tento způsob psaní mi připadá přehlednější než používání nové sekvence pro takové případy, jak bylo Štěpánem Kasalem původně navrženo.

Zvyk odlišovat v matematických textech proměnné kurzívními písmeny od neproměnných, jako jsou číselné konstanty (nejčastěji Eulerovo e nebo imaginární jednotka i) a značka d pro diferenciál, panuje snad v celé Evropě. U nás s tím mívají problémy jen fyzikové, kteří občas proměnné indexují náslovně (např. F_{t} pro třecí sílu), což se však ne vždy dá z kontextu jednoznačně poznat. V případě známé číselné konstanty $\pi = 3{,}141\,592\,653\ldots$ (pokud nepatříte mezi ty, kterým plně dostačuje 22/7) však už s řezy Computer Modern nevystačíme. Na rozdíl od řeckých verzálek, jež jsou obsaženy v základním řezu `cmr10` (a matematici jim, nevím proč, vesměs dávají přednost před kurzívním řezem), minusky existují jen skloněné, a to v základním matematickém fontu `cmmi10`. Nicméně v archivu CTAN si můžeme vybrat hned z několika pěkných metafontových verzí. Já jsem kdysi zvolil řečtinu Silvia Levyho, kterou jsem často používal i pro sazbu řeckých citací, ale protože obsahuje velké množství akcentovaných znaků, zredukoval jsem ji pro potřeby matematické sazby na malá písmena (viz tabulky na konci článku). Neskloněné řecké minusky najdeme ještě mezi eule-

rovskými fonty Hermanna Zapfa (`eurm10`), ta se však k běžné sazbě s písmy Computer Modern příliš nehodí (Knuth sám k nim vytvořil variantu Concrete, již použil k sazbě knihy [GKP]).

Vzhledem k tomu, že i z uvedených minusek většinou použijeme kromě zmíněného π už jen μ (při sazbě předpony mikro- u jednotek SI), nemá smysl plýtvat na zavedení neskloněné řečtiny do matematiky další rodinou (neboť jak známo, je počet rodin v TEXu omezen na šestnáct, což některým matematikům stejně nestačí), a tak raději využijeme primitiv `\mathchoice`:

```
%======================== Silvio Levy's nonslanted greek
\font\tengr=gr10
\font\sevengr=gr7
\font\fivegr=gr5
\let\oldpi\pi
\def\pi{{\mathchoice{\hbox{\tengr p}}{\hbox{\tengr p}}
            {\hbox{\sevengr p}}{\hbox{\fivegr p}}}}
\def\mugr{\mathchoice{\hbox{\tengr m}}{\hbox{\tengr m}}
            {\hbox{\sevengr m}}{\hbox{\fivegr m}}}
\let\mathmu\mu
\def\mu{\ifmmode\mathmu\else$\mugr$\fi}
```

V anglických textech se běžně k oddělení řádu tisíců používá čárka, zatímco anglická typografie k oddělení zlomkové části dekadického číselného zápisu používá desetinnou tečku. Na rozdíl od angličtiny se u nás i jinde v Evropě používá místo tečky desetinná čárka, zatímco skupiny číslic vždy po třech řádech oddělujeme tenkým výplňkem.

Protože TEX zná jen desetinnou tečku, předpokládá o čárce v matematickém módu, že slouží k oddělování např. prvků množiny, uspořádané dvojice apod., stačí dát desetinnou čárku do složených závorek, a vypnout tak její charakter „punct“: `$3{,}141\dots$`

Důmyslnější postup pomocí `\mathcode` je popsán např. v [TBN], nicméně je potřeba jisté opatrnosti, protože hodně autorů klidně píše interpunkci do matematického módu, takže se nám pak může stát, že některá důležitá věta skončí čárkou…

Mezi významné odlišnosti sazby matematiky v českém jazyce bych ještě zařadil výběr písma pro sazbu vektorů a matic (případně i zobrazení ve školské matematice). Anglosaská matematika používá pro vektory neskloněný tučný řez (rodina 6 v plainu), u nás se preferuje šikmý polotučný grotesk. V rodině Computer Modern najdeme vcelku snadno vhodný řez, má však pro nás jeden podstatný nedostatek: malé *a* nemá kurzívní kresbu! Proto jsem pro potřebu matematiky vytvořil variantu `cmvee10` (viz tabulky na konci článku) s odpovídajícím znakem i trochu tučnější kresbou a sadu jsem doplnil i o číslice (i když se využije asi

jen nula pro označení nulového vektoru) a malá řecká písmena, jež se též občas fyzikům hodí k označování např. vektoru úhlové rychlosti. A protože se použitý řez Computer Modern přeci jen liší od tradičního grotesku, vytvořil jsem v poslední době i řez `cmvgr` vzniklý kombinací grotesku od URW a několika řeckých písmen získaných úpravou znaků z Computer Modern.

Doplňkové sady značek `msam10` a `msbm10` pro $\mathcal{AMS}$-TEX obsahují, jak známo, také tzv. zdvojenou latinku. Ta se však bohužel dost podstatně liší od té, jež byla v původní verzi `msym10`, se kterou jsme se ještě běžně setkávali na začátku 90. let. Když jsem se pídil po původní variantě, jež kresebně zcela odpovídala klasickým vzorníkům značek a písem a pro sazbu matematiky mi svou jednodušší kresbou bez „ozdobných patek“ připadá jednoznačně vhodnější, ukázalo se, že původní verze byly napsány ještě ve staré verzi METAFONTu, jež se od stávající liší natolik, že se nedají původní programy použít. Protože všude byly k dispozici jen bitmapy v dnes už nízkém rozlišení 300 dpi, začal jsem postupně jednotlivé znaky pro velká písmena v `msbm10` upravovat (nejdříve přirozeně $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ a $\mathbb{C}$), až se mi podařilo zrekonstruovat písmena všechna. Zatím jsem písmena nikde nešířil, ponechal jsem jim tedy i původní název `msym`. Bude-li zájem, mohu je dát volně k dispozici, bude však třeba rozhodnout, zda je možno ponechat původní, dnes již nepoužívaný název. V tabulkách na konci článku můžete porovnat jejich kresbu i s dalším podobným řezem `bbold10` z kolekce fontů Sant Mary. Zejména písmeno $\mathbb{N}$ stojí za porovnání!

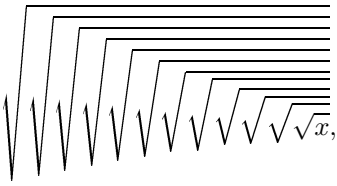
Na závěr ještě několik zajímavostí. Již několik let se dá v archivu CTAN najít rozšířená verze fontu `cmex10` od Yannise Haralambouse (`yhcmex10`), která obsahuje mnohem *širší* výběr některých opravdu *širokých* akcentů (viz tabulky na konci článku), bohužel ale nelze bezhlavě zaměnit `cmex10` za Yannisův `yhcmex10`, protože kvůli omezení formátu TFM na pouhých 16 různých výšek a hloubek je přirozeným důsledkem zvýšeného počtu odmocnítek a velkých závorek tato hranice podstatně překročena, takže METAFONT ohlásí

```
some chardp values had to be adjusted by as much as 1.30002pt.
```

To se bohužel projeví, jakmile potřebujeme vysadit větší složenou závorku:

$$\left\{\begin{array}{l} x = \left\{\begin{array}{l} 0, \\ 1 \end{array}\right. \\ x = \left\{\begin{array}{l} 0, \\ 1 \end{array}\right. \end{array}\right.$$

Zato možnost vysadit dvanáct krásných odmocnin a superdlouhé obloučky



$$\widehat{A},\ \widehat{AB},\ \widehat{ABC},\ \widehat{ABCD},\ \widehat{ABCDE},\ \widehat{ABCDEF},\ \widehat{ABCDEFG}$$

nám případné zklamání bohatě vynahradí.

Eulerovské fonty vytvořené Hermannem Zapfem v rámci jednoho projektu na Stanfordu, jež jsou součástí instalace $\mathcal{AMS}$-TEXu, byly využity při sazbě jedné po všech stránkách krásné knihy *Concrete Mathematics* autorů Grahama, Knutha a Patashnika. Nicméně neznám jiný zajímavý příklad knihy, kde by byly použity výhradně pro sazbu matematiky. Kdysi jsem použil variantu řezu `eurm10` doplněnou o akcentovaná česká písmena pro sazbu Dürerovy Apokalypsy. Doufám, že se mi brzy podaří připravit i nějakou matematickou publikaci tak, aby její čtenáři mohli ocenit půvab a originalitu takového řešení matematické sazby. Makra pod názvem `gkp.tex` jsou ostatně volně k dispozici a věřte, že jejich studium není ztrátou času.

Literatura:

GKP Graham, Knuth, Patashnik: *Concrete Mathematics*
TBN Petr Olšák: *TEXbook naruby*
W Karel Wick: *Pravidla matematické sazby*

Summary: Typesetting mathematics in Czech texts

Paper deals with some traditional differences between Czech (and mostly European) and American mathematical typographical rules. Some solutions using TEX and METAFONT are presented.

eurm10

Γ Δ Θ Λ Ξ Π Σ Υ
Φ Ψ Ω α β γ δ ϵ
ζ η θ ι κ λ μ ν
ξ π ρ σ τ υ ϕ χ
ψ ω ε ϑ ϖ φ

0 1 2 3 4 5 6 7
8 9 . , < / >
∂ A B C D E F G
H I J K L M N O
P Q R S T U V W
X Y Z
ℓ a b c d e f g
h i j k l m n o
p q r s t u v w
x y z ı ȷ ℘

eusm10

– ~ ¬ ℜ ℑ § ∧ ∨ { } | \

eufm10

𝔟 𝔡 𝔣 𝔣 𝔤 𝔨 𝔱 𝔲
‘ ’
! & '
( ) * + , – . /
0 1 2 3 4 5 6 7
8 9 : ; = ?
𝔄 𝔅 𝔈 𝔇 𝔈 𝔉 𝔊
𝔥 𝔍 𝔍 𝔎 𝔏 𝔐 𝔑 𝔒
𝔓 𝔔 𝔑 𝔖 𝔗 𝔘 𝔙 𝔚
𝔛 𝔜 𝔷 [ ] ^
𝔞 𝔟 𝔠 𝔡 𝔢 𝔣 𝔤
𝔥 𝔦 𝔧 𝔨 𝔩 𝔪 𝔫 𝔬
𝔭 𝔮 𝔯 𝔰 𝔱 𝔲 𝔳 𝔴
𝔵 𝔶 𝔷 " 1

gr10

σψ α β σ δ ε φ γ η ι θ κ λ μ ν ο π χ ρ ς τ υ ω ξ ψ ζ

cmnsy10

≪ ≮ ⋘ ⋙ ∞

cmvee10

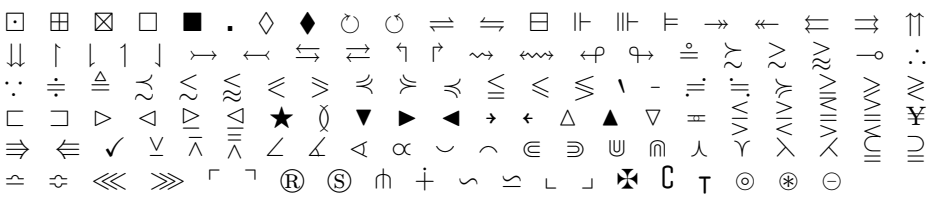
Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω α β γ δ ϵ ζ η θ ι κ λ μ ν ξ
π ρ σ τ υ ϕ χ ψ ω ε ϑ ϖ ϱ ς φ 0 1 2 3 4 5 6 7 8 9 A
B C D E F G H I J K L M N O P Q R S T U V W X Y
Z a b c d e f g h i j k l m n o p q r s t u v w x y z

cmvgr

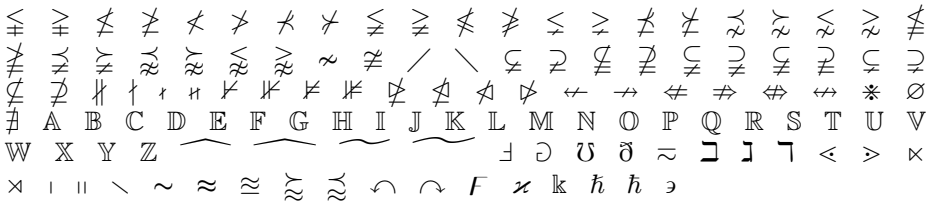
Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω α μ τ ω ϵ ( ) * - / 0 1
2 3 4 5 6 7 8 9 < = > A B C D E F G H I J K L M

N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

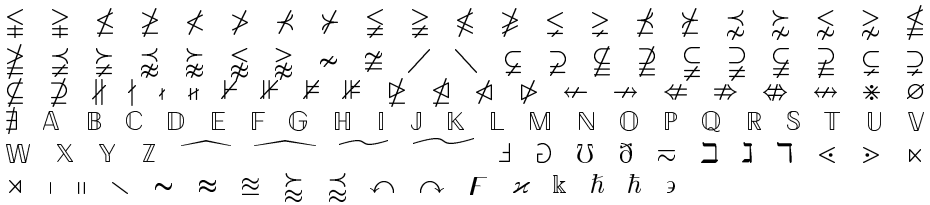
msam10



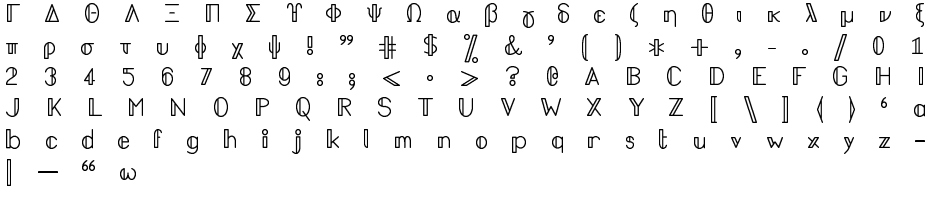
msbm10



msym10



bbold10



rsfs10

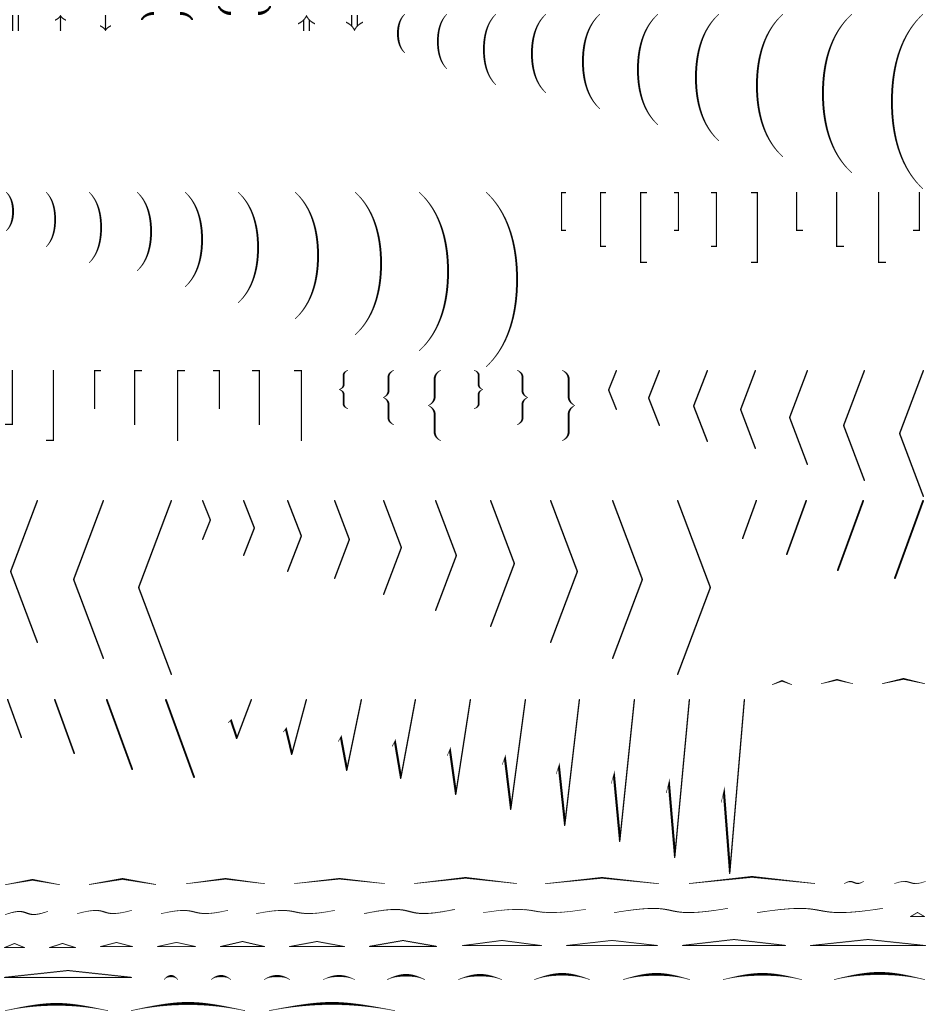
$\mathscr{A}\ \mathscr{B}\ \mathscr{C}\ \mathscr{D}\ \mathscr{E}\ \mathscr{F}\ \mathscr{G}\ \mathscr{H}\ \mathscr{I}\ \mathscr{J}\ \mathscr{K}\ \mathscr{L}\ \mathscr{M}\ \mathscr{N}\ \mathscr{O}\ \mathscr{P}\ \mathscr{Q}\ \mathscr{R}\ \mathscr{S}\ \mathscr{T}$
$\mathscr{U}\ \mathscr{V}\ \mathscr{W}\ \mathscr{X}\ \mathscr{Y}\ \mathscr{Z}\ \mathscr{o}\ t\ \widehat{}$

cmex10

$( )\ [ ]\ \lfloor\ \rfloor\ \lceil\ \rceil\ \{ \}\ \langle\ \rangle\ |\ \|\ /\ \backslash\ ( )\ \big(\big)\ \big[\big]\ \big\lfloor\ \big\rfloor\ \big\lceil$
$\big\rceil\ \big\{ \big\}\ \big\langle\ \big\rangle\ \big/\ \big\backslash\ \Big(\Big)\ \Big[\Big]\ \Big\lfloor\ \Big\rfloor\ \Big\lceil\ \Big\rceil\ \Big\{ \Big\}\ \Big\langle\ \Big\rangle\ \Big/\ \Big\backslash$
$\Big/\ \Big\backslash\ \Big(\Big)\ \Big[\Big]\ \Big\lfloor\ \Big\rfloor\ '\ '\ \big(\big)\ \big(\big)\ \big\{ \big\}\ \big\{ \big\}\ '\ '\ \big\backslash\ \big)\ '$
$'\ \big\langle\ \big\rangle\ \sqcup\ \bigsqcup\ \oint\ \oint\ \odot\ \bigodot\ \oplus\ \bigoplus\ \otimes\ \bigotimes\ \sum\ \prod\ \int\ \bigcup\ \bigcap\ \biguplus$
$\wedge\ \vee\ \sum\ \prod\ \int\ \bigcup\ \bigcap\ \biguplus\ \wedge\ \vee\ \coprod\ \coprod\ \widehat{}\ \overbrace{}\ \widetilde{}\ \overbrace{}\ [$
$]\ \lfloor\ \rfloor\ \lceil\ \rceil\ \{ \}\ \surd\ \surd\ \surd\ \surd\ \backslash\ |\ \lceil\ \|\ \uparrow\ \downarrow\ \overbrace{}\ \overbrace{}\ \underbrace{}\ \underbrace{}\ \Uparrow$
$\Downarrow$

yhcmex10

$( )\ [ ]\ \lfloor\ \rfloor\ \lceil\ \rceil\ \{ \}\ \langle\ \rangle\ |\ \|\ /\ \backslash\ ( )\ \big(\big)\ \big[\big]\ \big\lfloor$
$\big\rfloor\ \big\lceil\ \big\rceil\ \big\{ \big\}\ \big\langle\ \big\rangle\ \big/\ \big\backslash\ \Big(\Big)\ \Big[\Big]\ \Big\lfloor\ \Big\rfloor\ \Big\lceil\ \Big\rceil\ \Big\{ \Big\}\ \Big\langle\ \Big\rangle$
$\Big/\ \Big\backslash\ \Big/\ \Big\backslash\ \Big(\Big)\ \Big[\Big]\ \Big\lfloor\ \Big\rfloor\ '\ '\ \big(\big)\ \big(\big)\ \big\{ \big\}\ \big\{ \big\}\ '\ '$
$\big\backslash\ \big)\ '\ '\ \big\langle\ \big\rangle\ \sqcup\ \bigsqcup\ \oint\ \oint\ \odot\ \bigodot\ \oplus\ \bigoplus\ \otimes\ \bigotimes\ \sum\ \prod$
$\int\ \bigcup\ \bigcap\ \biguplus\ \wedge\ \vee\ \sum\ \prod\ \int\ \bigcup\ \bigcap\ \biguplus\ \wedge\ \vee\ \coprod\ \coprod\ \widehat{}\ \overbrace{}$
$\widetilde{}\ \overbrace{}\ [\ ]\ \lfloor\ \rfloor\ \lceil\ \rceil\ \{ \}\ \surd\ \surd\ \surd\ \surd\ \backslash\ |\ \lceil$



BUGS and UPDATES for the TEX Live CD-ROM (version 6)

Installation and setup

On the TEX Live 6 CD (v. 20010720 and 20010730), the size of the main memory allocated in LaTeX is wrong. To correct it:

- find the location of the main configuration file with command `kpsewhich texmf.cnf`
- edit the `texmf.cnf` file and replace 'main_memory.latex = 11000000' with 'main_memory.latex = 1100000'
- recreate LaTeX format: `fmtutil --byfmt=latex`

Systems

Macintosh

People wishing to mount the TEX Live CD on a Mac OS computer will find that there is a sad lack in that OS in that it supports neither the Joliet nor the Rock Ridge extensions, so you will see only the ISO 9660 file names (8+3 format).

The following link leads to freeware providing a Joliet file system for Mac OS: `http://www.tempel.org/joliet/`

After installing this system extension, you will be able to mount any ISO 9660 CD-ROM with Joliet extensions and see the full file names, access the CD-ROM's web pages with your favorite browser, etc.

Windows 32

The autostart/setup programs on the TeX live CD on Windows 95 only run with Internet Explorer (>= 4.0?) installed. Otherwise the programs complain about WININET.DLL missing. This is missing from the documentation, but there is a redistributable component from Microsoft that will install this dll on systems where it is lacking. Run `setupw32\wintdist.exe`, follow the instructions and then you will be able to autorun the CD.

Fonts

No known problems

Macros

`http://tug.org/texlive/tlprod/Master/texmf/tex/latex/texlive/graphics.cfg` and `http://tug.org/texlive/tlprod/Master/texmf/tex/latex/texlive/color.cfg` files have been updated by Heiko Oberdiek. Grab the files and install them in your `$TEXMF/tex/latex/texlive/` directory.

Extra binary sets
Author: Sebastian Rahtz.
Please send comments by email to `sebastian.rahtz@oucs.ox.ac.uk`

TUGboat 21(1), March 2000

TUGboat 21(1), March 2000

Zpravodaj Československého sdružení uživatelů TEXu
ISSN 1211-6661

Vydalo:	Československé sdružení uživatelů TEXu vlastním nákladem jako interní publikaci
Obálka:	Antonín Strejc
Počet výtisků:	750
Uzávěrka:	11. září 2001
Odpovědný redaktor:	Zdeněk Wagner
Tisk a distribuce:	KONVOJ, spol. s r. o., Berkova 22, 612 00 Brno, tel. 05-740233
Adresa:	CSTUG, c/o FI MU, Botanická 68a, 602 00 Brno
fax:	05–412 125 68
e-mail:	cstug@cstug.cz

Zřízené poštovní aliasy sdružení CSTUG:

`bulletin@cstug.cz`, `zpravodaj@cstug.cz`
korespondence ohledně Zpravodaje sdružení

`board@cstug.cz`
korespondence členům výboru

`cstug@cstug.cz`, `president@cstug.cz`
korespondence předsedovi sdružení

`gacstug@cstug.cz`
grantová agentura CSTUGu

`secretary@cstug.cz`, `orders@cstug.cz`
korespondence administrativní síle sdružení, objednávky CD-ROM

`cstug-members@cstug.cz`
korespondence členům sdružení

`cstug-faq@cstug.cz`
řešené otázky s odpověďmi navrhované k zařazení do dokumentu CSFAQ

`bookorders@cstug.cz`
objednávky tištěné TEXové literatury na dobírku

ftp server sdružení:
`ftp://ftp.cstug.cz/`

www server sdružení:
`http://www.cstug.cz/`

Podávání novinových zásilek povoleno Českou poštou, s. p. OZJM Ředitelství v Brně č. j. P/2–1183/97 ze dne 11. 3. 1997.