Ústav termomechaniky AV ČR
Centrum diagnostiky materiálu
Veleslavínova 11
301 14 Plzeň

Výzkumná zpráva

METAPOST & Toolbox mmp

Petr Hora, Olga Červená

2005

Laboratoř modelování vln v tělesech

Centrum diagnostiky materiálu ÚT AV ČR

Tato zpráva byla sepsána pomocí KOMA-Scriptu a LaTeXu

Obsah

Seznam tabulek

Seznam schémat

Seznam obrázků

1 METAPOST

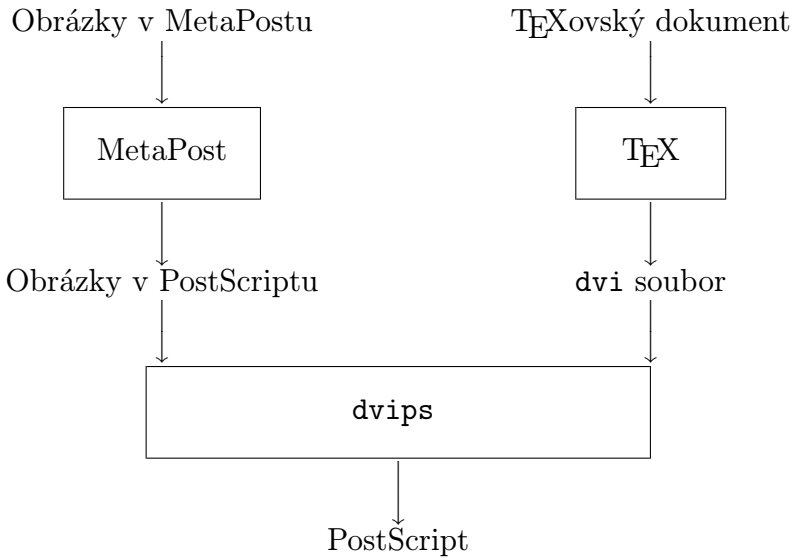
METAPOST je programovací jazyk určený pro tvorbu obrázků. (Jeho autorem je John D. Hobby.) Výsledným produktem je program v PostScriptu, který vykreslí daný obrázek. Na rozdíl od PostScriptu poskytuje mnoho zajímavých funkcí, díky kterým je možno kreslení zjednodušit a zrychlit. Mezi nejpoužívanější funkce bezesporu patří automatické řešení soustav lineárních rovnic, výpočet průniku křivek, plynulé navazování, automatický výpočet kontrolních bodů Beziérových křivek tak, aby byla výsledná křivka co nejhezčí, vkládání popisků, využití různých stylů kreslení čar. Vygenerované obrázky je možno použít mnoha způsoby, například je možné propojit je s LaTeXovským textem. Poznamenejme, že programy, jako např. GNUPLOT, fig2vect, 3DLDF, METAGRAF a pstoedit, vytvářejí také zdrojové soubory METAPOSTu.

1.1 Úvod

METAPOST je programovací jazyk stejně jako Knuthův METAFONT[1] [Knu86a] až na to, že výstupem jsou PostScriptové programy a nikoliv bitmapy. Z METAFONTu jsou převzaté základní nástroje pro vytváření obrázků a manipulaci s nimy. Jsou to čísla, páry souřadnic, kubické interpolace, afinní transformace, textové řetězce a booleovské veličiny. Další nástroje usnadňují spojování textů a grafiky a zpřístupňují speciální vlastnosti PostScriptů[2] jako je ořezávání, kreslení přerušovaných čar. Důležitým rysem převzatým z METAFONTu je schopnost řešit lineární rovnice, které jsou zadány implicitně. To umožňuje mnoha programům, aby byly napsány převážně deklarativně. Vytvářením složitých operací z jednoduchých je docíleno jak síly, tak i flexibility METAPOSTu.

METAPOST je zvláště vhodný pro generování obrázku pro technické dokumenty, kde některé aspekty obrázků mohou být matematicky nebo geometricky omezeny a tak je vhodnější jejich symbolické vyjádření. Jinými slovy, METAPOST nemá nahrazovat nástroj kreslení od ruky nebo dokonce interaktivní grafický editor. Je to programovací jazyk pro generování grafiky, zejména obrázků pro TeX[3] a troffovské dokumenty[4]. Obrázky pak mohou být začleněny do TeXovského dokumentu pomocí volně využitelného programu `dvips`, jak je patrno z obrázku 1.1.[5] Podobná procedura pracuje s troffem: výstupní procesor `dpost` obsahuje PostScriptové obrázky, které jsou požadovány troffovským `\X` příkazem.

[1] METAFONT je ochranná známka Addison Wesley Publishing company.
[2] PostScript je ochranná známka Adobe Systems Inc.
[3] TeX je ochranná známka the American Mathematical Society.
[4] Troff (typesetter run off) je jeden z nejstarších textových editorů v Unixu s kvalitními formátovacími možnostmi, s podporou řecké abecedy a mnoha speciálními znaky a matematickými symboly.
[5] Zdrojový program pro `dvips` je součástí distribuce web2c TeX. Podobné programy jsou dostupné z dalších zdrojů.



Obrázek 1.1: Diagram zpracování TEXovského dokumentu s obrázky v METAPOSTu.

K použití METAPOSTu se nejdříve vytvoří vstupní soubor v METAPOSTovém kódu a pak se vyvolá METAPOST obvykle zadáním příkazu ve tvaru

`mp` ⟨file name⟩

(Tato syntaxe může záviset na systému). METAPOSTové vstupní soubory mají název, který končí „`.mp`“. Tato část názvu může být při volání METAPOSTu vynechána. Pro vstupní soubor `foo.mp` příkaz

```
mp foo
```

vyvolá METAPOST a vytvoří výstupní soubory s názvy `foo.1` a `foo.2`. Některé konečné vstupy a výstupy jsou shrnuty do logovacího souboru se jménem `foo.log`.

Ten obsahuje chybová hlášení a některé interaktivně zapsané METAPOSTové příkazy.[6] Výpisový soubor začíná záhlavím, které říká, jaká verze METAPOSTu je užita.

Tento dokument uvádí METAPOSTový jazyk, zpočátku ty funkce, které jsou nejdůležitější a nejsnáze použitelné pro jednoduché aplikace. Prvních několik kapitol popisuje jazyk pro začínající uživatele s implicitními hodnotami klíčových parametrů. Některé funkce popsané v těchto kapitolách jsou součástí předdefinovaného balíku maker nazvaného Plain. Další kapitoly shrnují celý jazyk a rozlišují mezi primárními a předdefinovanými makry z balíku maker Plain. Pokročilí uživatelé se mohou dozvědět víc o METAPOSTu v knize *The METAFONTbook*. [Knu86a]

1.2 Základní příkazy kreslení

Nejjednodušší příkazy kreslení jsou ty, které generují rovné čáry.

Tedy

```
draw (20,20)--(0,0)
```

[6] Výzva `*` je užívána pro interaktivní vstup a výzva `**` indikuje, že je očekáván vstupní soubor. Tomu lze předejít vyvoláním METAPOSTu na soubor, který končí příkazem `end`.

kreslí diagonální čáru a

```
draw (20,20)--(0,0)--(0,30)--(30,0)--(0,0)
```

kreslí lomenou čáru:

Co je myšleno souřadnicemi `(30,0)`? METAPOST užívá základní systém souřadnic, shodný se souřadným systémem jazyka PostScript. To znamená, že `(30,0)` je 30 jednotek napravo od počátku, kde jednotka je $\frac{1}{72}$ palce (`bp`). Toto je implicitní *PostScriptový bod* narozdíl od standardního typografického bodu, který je $\frac{1}{72.27}$ palce (`pt`).

METAPOST užívá stejný název pro měrné jednotky jako TEX a METAFONT. Tedy `bp` odpovídá PostScriptovým bodům („big points“) a `pt` odpovídá typografickým bodům. Další měrné jednotky jsou `in` pro palce , `cm` pro centimetry a `mm` pro milimetry. Například

```
(2cm,2cm)--(0,0)--(0,3cm)--(3cm,0)--(0,0)
```

generuje větší verzi předešlého diagramu. Je správné zadat `0` místo `0cm`, protože `cm` je vlastně pouze převodní faktor a `0cm` jen násobí převodní faktor nulou. (METAPOST považuje vazby typu `2cm` za zkratky pro `2*cm`).

Často je výhodné si zavést vlastní měřítko, např. u. Pak lze definovat souřadnice v rámci u a později se rozhodnout, zda chceme pracovat s `u=1cm` nebo `u=0.5cm`. To umožňuje kontrolovat, co mění velikost a co ne. Změna u nebude mít vliv například na šířky čar.

Šířku čáry lze nastavit pomocí příkazu:

```
pickup pencircle scaled 4pt,
```

který nastaví šířku čáry pro následující příkaz `draw` na 4 body. (To je osmkrát širší než implicitní šířka čáry).

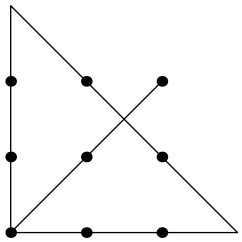
Široká čára, dokonce i čára nulové délky se zobrazí jako velká tučná tečka. Takto můžeme udělat síť z tučných teček pomocí příkazu `draw` pro každý bod sítě. Taková opakující se posloupnost příkazů `draw` se nejlépe zapíše jako dvojice vnořených smyček:

```
for i=0 upto 2:
  for j=0 upto 2:  draw (i*u,j*u);  endfor
endfor
```

Vnější smyčka běží pro $i = 0, 1, 2$ a vnitřní pro $j = 0, 1, 2$. Výsledkem je síť tvořená tři-krát-tři tučnými body, jak je vidět na obrázku 1.2. Obrázek také obsahuje větší verzi lomené čáry, která byla zobrazena dříve.

Všimněte si, že program v obrázku 1.2 začíná `beginfig(2)` a končí `endfig`. Jsou to makra, která vykonávají různé administrativní funkce a zajišťují, že výsledky ze všech `draw` příkazů budou shrnuty a přeloženy do PostScriptu. METAPOSTový vstup normálně obsahuje posloupnost dvojic příkazů `beginfig`, `endfig` s příkazem `end` za posledním z nich. Jestliže se soubor jmenuje `fig.mp`, výstup z příkazů `draw` mezi

```
beginfig(2);
u=1cm;
draw (2u,2u)--(0,0)--(0,3u)--(3u,0)--(0,0);
pickup pencircle scaled 4pt;
for i=0 upto 2:
  for j=0 upto 2:  draw (i*u,j*u);  endfor
endfor
endfig;
```



Obrázek 1.2: Možnosti METAPOSTu.

`beginfig(1)` a následujícím `endfig` je zapsán do souboru `fig.1`. Jinými slovy, číselný parametr makra `beginfig` určuje jméno odpovídajícího výstupního souboru. Je-li parametr záporný, výstupní soubor bude mít příponu `.ps`.

Co se dělá se všemi těmito PostScriptovými soubory? Jestliže máte výstupní ovladač, který umí pracovat se zapouzdřenými PostScriptovými obrázky, mohou být zahrnuty jako obrázky v TEXovských, LATEXovských nebo troffovských dokumentech. Jestliže váš standardní TEXovský adresář maker obsahuje soubor `epsf.tex`, pravděpodobně můžete začlenit `fig.1` do TEXovského dokumentu následovně:

```
\input epsf
   ⋮
$$\includegraphics{fig.1}$$
```

Makro `\includegraphics` vypočte kolik místa se má vynechat pro obrázek a užije TEXovský příkaz `\special` pro vložení požadovku na obrázk `fig.1`.

Pokud chceme začlenit obrázek do LATEXovského dokumentu pomocí překladače pdfLATEXu musí tento dokument obsahovat následující dvě řádky:

```
\usepackage{graphicx}
\DeclareGraphicsRule{*}{mps}{*}{}
```

Pokud použijeme překladače LATEXu, pak dokument musí obsahovat pouze první ze zmíněných řádků, tedy druhý řádek se v dokumentu vyskytovat nesmí. Obrázek `fig.1` v obou případech začleníme do dokumentu následovně:

```
\begin{figure}
   \includegraphics{fig.1}
   \caption[Popis odrázku] {Popis obrázku}
   \label{fig1}
\end{figure}
```

Také je možné zahrnout METAPOSTový výstup do *troffovského* dokumentu. Makro `-mpictures` definuje příkaz `.BP`, který zahrnuje zapouzdřený PostScriptový soubor. Například *troff* příkaz

```
.BP fig.1 3c 3c
```

zahrnuje `fig.1` a předepisuje, že výška i šířka obrázku bude tři centimetry.

1.3 Křivky

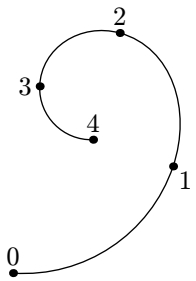
METAPOST kreslí stejně dobře křivky jako rovné čáry. Příkaz `draw` s body oddělenými `..` kreslí hladkou křivku procházející danými body. Například si prohlédněme výsledek příkazu

```
draw z0..z1..z2..z3..z4
```

po definici pěti následujících bodů:

```
z0 = (0,0);    z1 = (60,40);
z2 = (40,90);  z3 = (10,70);
z4 = (30,50);
```

Obrázek 1.3 ukazuje křivku procházející označenými body `z0` až `z4`.



Obrázek 1.3: Výsledek příkazu `draw z0..z1..z2..z3..z4`.

Existuje mnoho způsobů, jak nakreslit cestu procházející těmito pěti body. Vytvoření hladké uzavřené křivky, spojující bod `z4` s počátkem, docílíme přidáním `..cycle` k příkazu `draw`, jak je patrno na obrázku 1.4a. V jediném `draw` příkazu je také možné kombinovat křivky s přímými čárami, jak je znázorněno na obrázku 1.4b. Použijte `--` tam, kde chcete vykreslit přímou čáru a `..` kde chcete křivku. Tedy

```
draw z0..z1..z2..z3--z4--cycle
```

spojí křivkou body 0, 1, 2 a 3 a lomenou čáru vykreslí z bodu 3 do bodu 4 a zpět k bodu 0. Výsledek je v podstatě stejný, jako když se použijí dva příkazy

```
draw z0..z1..z2..z3
```

a

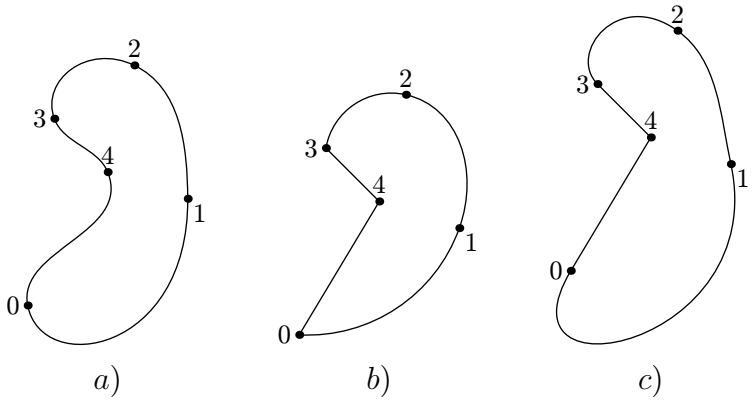
```
draw z3--z4--z0
```

Pro vykreslení přímé čáry je možné také použít `---` místo `--`. To však ovlivní celkový tvar cesty, jak je znázorněno na obrázku 1.4c. Význam `---` bude vysvětlen v kapitole 1.3.2.

1.3.1 Bézierovy křivky

Když má METAPOST vykreslit hladkou křivku procházející posloupností bodů, vytváří po částech kubickou křivku se spojitou směrnicí a přibližně plynulým zakřivením. To znamená, že následující předpis cesty

```
z0..z1..z2..z3..z4..z5
```

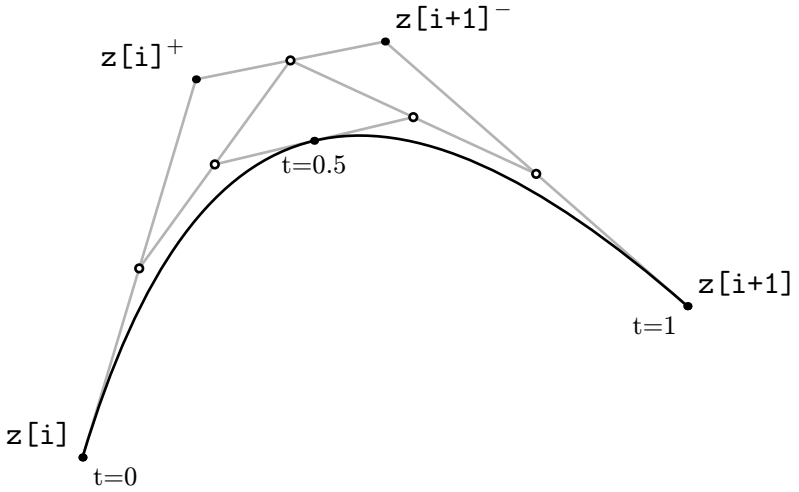


Obrázek 1.4: a) Výsledek příkazu `draw z0..z1..z2..z3..z4..cycle`;
b) výsledek příkazu `draw z0..z1..z2..z3--z4--cycle`;
c) výsledek příkazu `draw z0..z1..z2..z3---z4---cycle`.

má za výsledek křivku, která může být zapsána parametricky jako $(X(t), Y(t))$ pro $0 \leq t \leq 5$, kde $X(t)$ a $Y(t)$ jsou po částech kubické funkce. Tedy existují různé dvojice kubických funkcí pro každý celočíselně ohraničený t-interval. Jestliže `z0` $= (x_0, y_0)$, `z1` $= (x_1, y_1)$, `z2` $= (x_2, y_2)$, ..., METAPOST vybere Bézierovy kontrolní body (x_0^+, y_0^+), (x_1^-, y_1^-), (x_1^+, y_1^+), ..., kde

$$\begin{aligned} X(t+i) &= (1-t)^3 x_i + 3t(1-t)^2 x_i^+ + 3t^2(1-t)x_{i+1}^- + t^3 x_{i+1}, \\ Y(t+i) &= (1-t)^3 y_i + 3t(1-t)^2 y_i^+ + 3t^2(1-t)y_{i+1}^- + t^3 y_{i+1} \end{aligned}$$

pro $0 \leq t \leq 1$. Z těchto vztahů je možno odvodit jednoduchou grafickou konstrukci bodu s parametrem $t = 0,5$, která je založena na půlení úseček:



Obrázek 1.5: Konstrukce Bézierovy křivky.

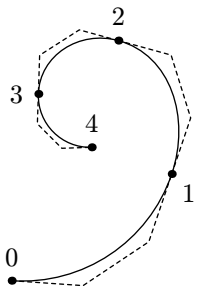
Opakováním konstrukce s využitím kontrolních bodů označených prázdným kroužkem bychom dostali body s parametry $t = 0,25$ a $t = 0,75$ atd. Přesná pravidla pro výběr Bézierových řídicích bodů jsou popsána v [Hob86] a v *The METAFONT book* [Knu86a]. Aby cesta měla spojitou směrnici v (x_i, y_i), musí se vstupní i výstupní

směry v $(X(i), Y(i))$ shodovat. Tedy vektory

$$(x_i - x_i^-,\ y_i - y_i^-) \qquad \text{a} \qquad (x_i^+ - x_i,\ y_i^+ - y_i)$$

musí mít stejný směr; tj. (x_i, y_i) musí být na spojnici mezi (x_i^-, y_i^-) a (x_i^+, y_i^+). Tato situace je znázorněna na obrázku 1.6, kde Bézierovy kontrolní body vybrané METAPOSTem jsou spojeny přerušovanými čarami. Pro ty, kteří jsou dobře seznámeni se zajímavými vlastnostmi této konstrukce, METAPOST umožňuje kontrolní body určit přímo v následujícím tvaru:

```
draw (0,0)..controls (26.8,-1.8) and (51.4,14.6)
 ..(60,40)..controls (67.1,61.0) and (59.8,84.6)
 ..(40,90)..controls (25.4,94.0) and (10.5,84.5)
 ..(10,70)..controls ( 9.6,58.8) and (18.8,49.6)
 ..(30,50);
```



Obrázek 1.6: Výsledek příkazu `draw z0..z1..z2..z3..z4` s řídícím mnohoúhelníkem.

1.3.2 Specifikace směru, napětí a zkřivení

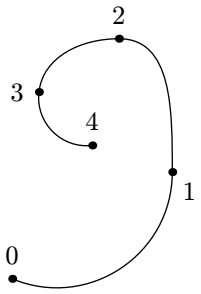
METAPOST poskytuje mnoho způsobů, jak kontrolovat chování křivek, kromě určení kontrolních bodů. Například, některé body cesty mohou být určeny jako vertikální nebo horizontální extrémy. Jestliže je například `z1` horizontální extrém a `z2` je například vertikální extrém, lze určit, že $(X(t), Y(t))$ by měla jít nahoru k `z1` a doleva k `z2`:

```
draw z0..z1{up}..z2{left}..z3..z4;
```

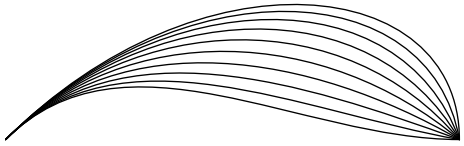
Výsledek znázorněný na obrázku 1.7 má požadované vertikální a horizontální extrémy v `z1` a `z2`, ale křivka není tak hladká jako na obrázku 1.3. Důvodem je velká nesouvislost v zakřivení v bodě `z1`. Pokud by nebyl určen směr v `z1`, METAPOSTový překladač by vybral směr zakřivení nad bodem `z1` téměř stejný, jako pod ním.

Jak může volba směrů v daných bodech určit, zda zakřivení bude souvislé? Odpovědí je, že křivky používané v METAPOSTu pocházejí ze systému křivek, které jsou určeny svými koncovými body a směry v nich. Obrázky 1.8 a 1.9 dávají dobrou představu o tom, jak tento systém křivek vypadá.

Obrázky 1.8 a 1.9 ukazují několik nových METAPOSTových funkcí. První z nich je operátor `dir`, který udává úhel ve stupních a generuje jednotkový vektor v tomto

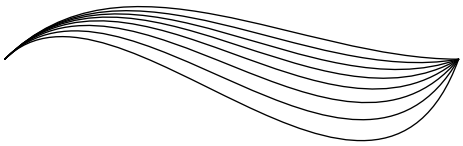


Obrázek 1.7: Výsledek příkazu `draw z0..z1{up}..z2{left}..z3..z4`.



```
beginfig(8)
for a=0 upto 9:
  draw (0,0){dir 45}..{dir -10a}(6cm,0);
endfor
endfig;
```

Obrázek 1.8: Vliv směru na tvar křivky.



```
beginfig(9)
for a=0 upto 7:
  draw (0,0){dir 45}..{dir 10a}(6cm,0);
endfor
endfig;
```

Obrázek 1.9: Vliv směru na tvar křivky, pokračování.

směru. Tedy příkaz `dir 0` je shodný s `right` a `dir 90` je shodný s `up`. Předdefinované jsou také směrové vektory `left` a `down` pro `dir 180` a `dir 270`.

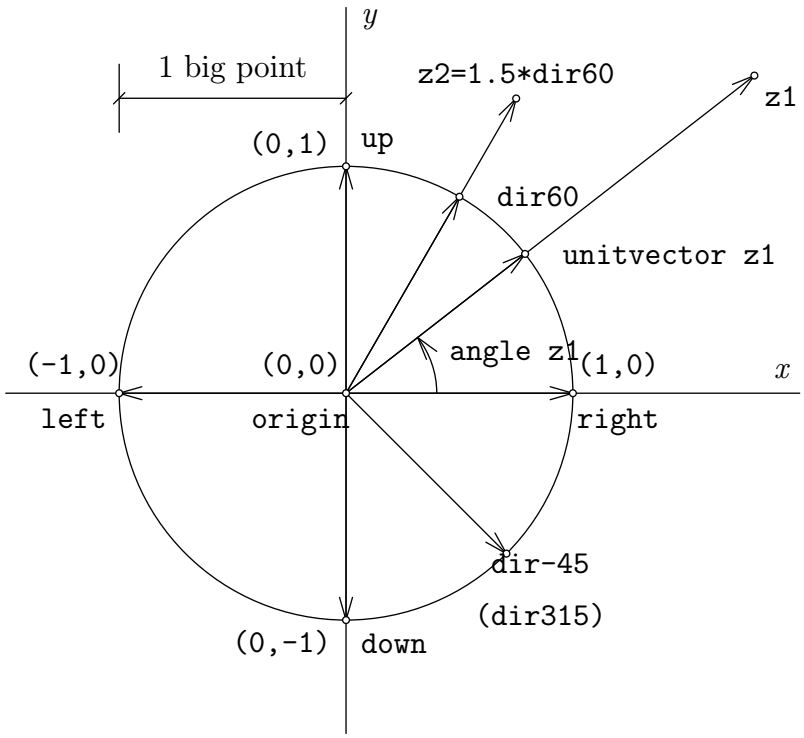
Směrové vektory zadané v `{}` mohou být libovolné délky a mohou být zadány před bodem stejně jako za ním. Je dokonce možné pro cestu určit směr před i za bodem. Například cesta obsahující

```
..{dir 60}(10,0){up}..
```

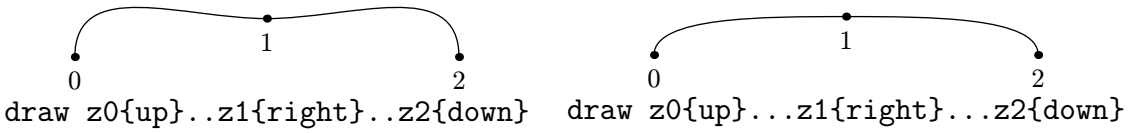
vytváří křivku s rohem v bodě $(10, 0)$.

Na obrázku 1.10 jsou znázorněny jednotlivé předdefinované směrové vektory.

Všimněte si, že některé křivky na obrázku 1.8 mají inflexní body. To je nezbytné k vytváření hladkých křivek jako na obrázku 1.4a, ale je to pravděpodobně nežádoucí, pokud se jedná o vertikální a horizontální extrémní body jako na obrázku 1.11a. Je-li `z1` předpokládaný nejvyšší bod křivky, tak ho může být dosaženo užitím `...` místo `..` ve specifikaci cesty, jak je vidět na obrázku 1.11b. Význam `...` je „vyber cestu bez inflexních bodů, jestliže to směry koncových bodů umožňují“. (To umožňuje vyhnou se inflexím v obrázku 1.8, ale ne v obrázku 1.9).



Obrázek 1.10: Předdefinované směrové vektory.



Obrázek 1.11: Odstranění inflexních bodů z křivky.

Další způsob, jak upravit špatné chování cesty, je zvýšit parameter „napětí“. Užitím `..` v cestě, se parametr napětí nastaví na implicitní hodnotu 1. Pokud je některá část křivky poněkud divoká, lze napětí libovolně zvýšit. Je-li obrázek 1.12a považován za „příliš divoký“, příkaz `draw` v následujícím tvaru zvýší napětí mezi `z1` a `z2`:

```
draw z0..z1..tension 1.3..z2..z3
```

Toto vytvoří obrázek 1.12b. Pro asymetrický výsledek jako na obrázku 1.12c, příkaz `draw` vypadá

```
draw z0..z1..tension 1.5 and 1..z2..z3
```

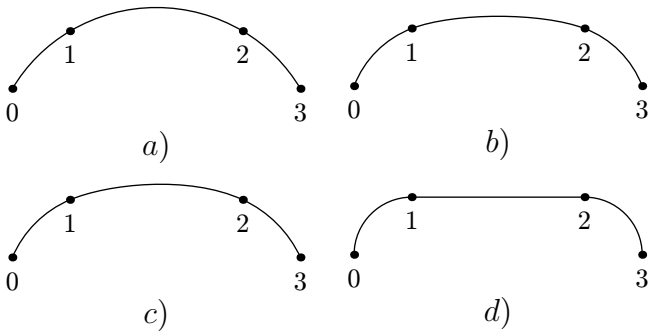
Parametr napětí může být menší než jedna, ale musí být alespoň $\frac{3}{4}$. Pokud zvolíme parametr napětí $= \infty$, vykreslí se rovná čára viz 1.12d a můžeme tedy použít `---` , neboť příkaz

```
draw z0..z1..tension infinity..z2..z3
```

je ekvivalentní s

```
draw z0..z1---z2..z3
```

METAPOSTposkytuje příkaz `tensepath`, který se aplikuje na cestu a vrací cestu jako lomenou čáru (Mezi jednotlivé body cesty vloží `---`.).

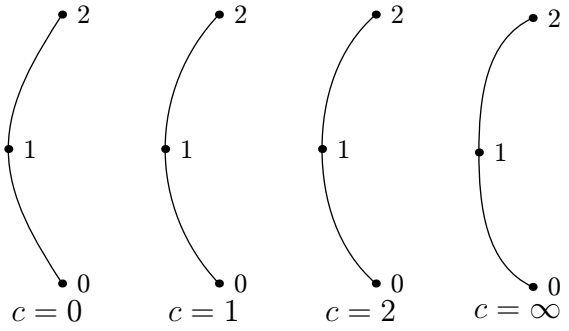


Obrázek 1.12: Výsledek `draw z0..z1..tension` α `and` β `..z2..z3`
pro různé hodnoty α a β:
a) $\alpha = \beta = 1$; b) $\alpha = \beta = 1.3$; c) $\alpha = 1.5$, $\beta = 1$; d) $\alpha = \beta = \infty$.

METAPOSTová cesta má také parametr zvaný „zakřivení - curl“, který ovlivňuje konce cesty. Není-li určen směr, pak první a poslední část necyklické cesty jsou přibližně oblouky kružnice jako v případě $c = 1$ na obrázku 1.13. Jinou hodnotu parametru zakřivení c zadáme pomocí `{curl c}`. Tedy

```
draw z0{curl c}..z1..{curl c}z2
```

nastavuje parametry zakřivení pro body `z0` a `z2`. Malé hodnoty parametru snižují zakřivení v uvedeném koncovém bodu cesty, zatímco vysoké hodnoty zvyšují křivost, jak je patrno z obrázku 1.13. Především hodnoty blízké nule vytváří téměř nulové zakřivení.



Obrázek 1.13: Výsledek `draw z0{curl c}..z1..{curl c}z2`
pro různé hodnoty parametru zakřivení c.

1.3.3 Přehled syntaxe pro cesty

Existuje ještě několik dalších rysů METAPOSTové syntaxe pro cesty, ale ty nejsou příliš zajímavé. METAFONT užívá stejnou syntaxi pro cesty a tak lze zaujaté zájemce odkázat na [Knu86a, chapter 14]. Přehled syntaxe pro cesty na schématu 1.1 (svislá čárka (`|`) ve schématu má význam „nebo“) zahrnuje všechno doposud probrané včetně `--` , `---` , `..` a `...`, které jsou spíše makry než základními konstrukcemi.

⟨path expression⟩ → ⟨path subexpression⟩
| ⟨path subexpression⟩⟨direction specifier⟩
| ⟨path subexpression⟩⟨path join⟩ `cycle`
⟨path subexpression⟩ → ⟨path knot⟩
| ⟨path expression⟩⟨path join⟩⟨path knot⟩
⟨path join⟩ → `--`
| ⟨direction specifier⟩⟨basic path join⟩⟨direction specifier⟩
⟨direction specifier⟩ → ⟨empty⟩
| `{curl` ⟨numeric expression⟩`}`
| `{`⟨pair expression⟩`}`
| `{`⟨numeric expression⟩`,`⟨numeric expression⟩`}`
⟨basic path join⟩ → `..` | `...` | `---` | `..`⟨tension⟩`..` | `..`⟨controls⟩`..`
⟨tension⟩ → `tension`⟨numeric primary⟩
| `tension`⟨numeric primary⟩`and`⟨numeric primary⟩
⟨controls⟩ → `controls`⟨pair primary⟩
| `controls`⟨pair primary⟩`and`⟨pair primary⟩

Schéma 1.1: Syntaxe pro konstrukci cesty

Nyní pár poznámek k sémantice: Jestliže je ⟨direction specifier⟩ před ⟨path knot⟩ neprázdný, ale ne za ním nebo naopak, použije se směrový specifikátor (nebo velikost zakřivení) na vstupní i výstupní části cesty. Podobné uspořádání platí, když specifikace ⟨controls⟩ udává pouze jeden ⟨pair primary⟩. Tedy

```
..controls (30,20)..
```

je shodné s

```
...controls (30,20) and (30,20)..
```

To, co se na schématu 1.1 nazývá ⟨pair primary⟩ je buď pár souřadnic jako `(30,20)` nebo proměnná `z`, která reprezentuje souřadnice bodu. Výraz ⟨path knot⟩ je podobný, až na to, že může přibírat další tvary jako výraz `path` v závorkách. Základní i další výrazy různých typů budou probírány zcela obecně v kapitole 1.5.

Na obrázku 1.14 jsou shrnuty již probrané konstrukce cest.

1.4 Lineární rovnice

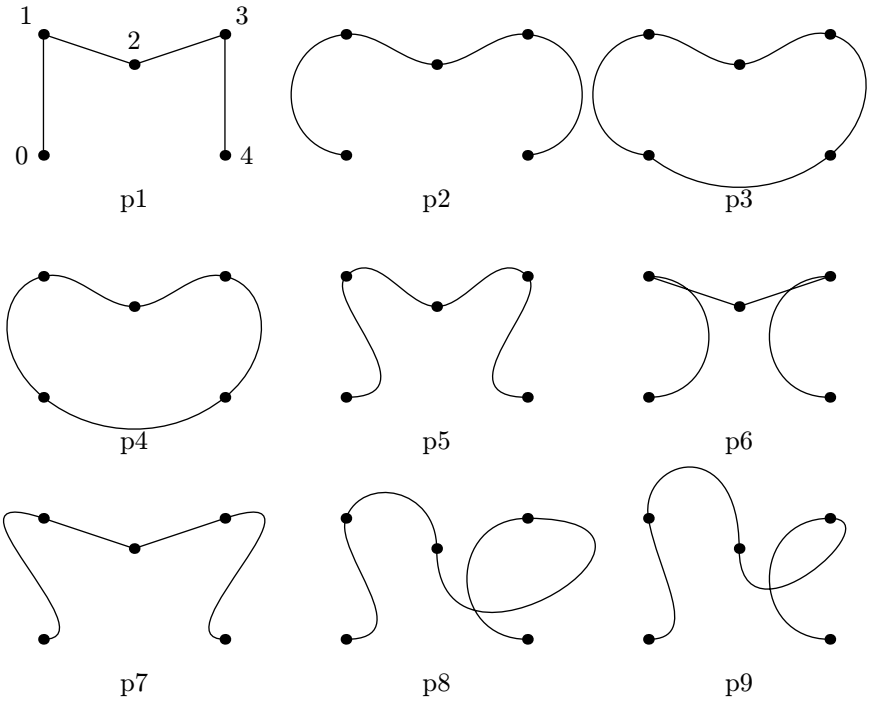
Zajímavá vlastnost převzatá z METAFONTu je schopnost řešit lineární rovnice tak, že program může být napsán v částečně deklarativním tvaru. Například, METAPOSTový překladač může číst

```
a+b=3; 2*a=b+3;
```

a odvodit, že $a = 2$ a $b = 1$. Stejné rovnice mohou být napsány zhuštěně jejich spojením pomocí vícenásobného použití znaménka rovnosti:

```
a+b = 2*a-b = 3;
```

```
p1 = z0--z1--z2--z3--z4;
p2 = z0..z1..z2..z3..z4;
p3 = z0..z1..z2..z3..z4..z0;
p4 = z0..z1..z2..z3..z4..cycle;
p5 = z0{dir0}..z1..z2..z3..{dir0}z4;
p6 = z0{dir0}..z1--z2--z3..{dir0}z4;
p7 = z0{dir0}..z1---z2---z3..{dir0}z4;
p8 = z0{right}..z1..{dir270}z2..z3{dir180}..{dir0}z4;
p9 = z0{right}..z1..tension.75..z2{dir270}..tension1.5and3..z3{dir180}
     ..{dir0}z4;
```



Obrázek 1.14: Ukázky tvorby cest.

Jakkoliv rovnice zadáte, vždy pak můžete zadat příkaz

```
show a,b;
```

k zobrazení hodnot `a` a `b`. METAPOST odpoví vypsáním

```
>> 2
>> 1
```

do logovacího souboru.

Všimněte si, že `=` není přiřazovací operátor; jednoduše udává, že levá strana se rovná pravé straně. Tedy `a=a+1` vyvolá chybové hlášení „inconsistent equation.“ Způsob, jak zvýšit hodnotu `a`, je použít přiřazovací operátor `:=` následujícím způsobem:

```
a:=a+1;
```

Jinými slovy `:=` je určen pro změnu existující hodnoty, `=` je pro zadávání lineárních rovnic k jejich řešení.

Neexistuje žádné omezení pro směšování rovnic a přiřazovacích operátorů jako v následujícím příkladě:

```
a = 2; b = a; a := 3; c = a;
```

Po prvních dvou rovnicích je nastaveno `a` a `b` rovno 2, přiřazovací operátor pak mění `a` na 3 bez vlivu na `b`. Konečná hodnota `c` je 3, protože je rovna nové hodnotě `a`. Obecně je přiřazovací operátor vyhodnocen prvním výpočtem nové hodnoty, pak je odstraněna stará hodnota ze všech rovnic existujících před přidělením nové hodnoty.

1.4.1 Rovnice a páry souřadnic

METAPOST může řešit také lineární rovnice obsahující páry souřadnic. Už jsme viděli mnoho triviálních příkladů rovnic ve tvaru jako

```
z1=(0,.2in).
```

Každá strana rovnice musí být tvořena sčítáním nebo odčítáním souřadnic a jejich násobením nebo dělením známou číselnou veličinou. Jiný způsob pojmenování dvouhodnotové proměnné bude probrán později, ale výhodný je zápis `z`⟨number⟩, protože je to zkrácený zápis pro

`(x`⟨number⟩`, y`⟨number⟩`)`.

Toto umožňuje zadávat hodnotu proměnné `z` zadáním rovnice zahrnující její souřadnice. Například, body `z1`, `z2`, `z3`, a `z6` na obrázku 1.15 byly zadány pomocí následujících rovnic:

```
z1=-z2=(.2in,0);
x3=-x6=.3in;
x3+y3=x6+y6=1.1in;
```

Přesně stejné body mohou být získány přímo zadáním hodnot jejich souřadnic:

```
z1=(.2in,0); z2=(-.2in,0);
z3=(.3in,.8in); z6=(-.3in,1.4in);
```

Po přečtení rovnic, zná METAPOSTový překladač hodnoty `z1`, `z2`, `z3` a `z6`. Následující úkol ve vytváření obrázku 1.15 je definovat body `z4` a `z5` rovnoměrně rozmístěné na spojnici mezi `z3` a `z6`. Tato operace vyskytuje často, proto má pro ni METAPOST speciální syntaxi. Tato zprostředkující vazba

```
z4=1/3[z3,z6]
```

znamená, že `z4` je v $\frac{1}{3}$ cesty z $z3$ do $z6$; tj.,

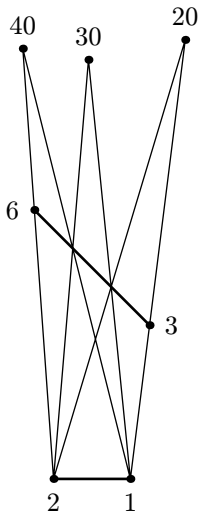
$$\texttt{z4} = \texttt{z3} + \frac{1}{3}(\texttt{z6} - \texttt{z3}).$$

Podobně

```
z5=2/3[z3,z6]
```

určuje `z5` ve $\frac{2}{3}$ cesty z $z3$ do $z6$.

```
beginfig(15);
z1=-z2=(.2in,0);
x3=-x6=.3in;
x3+y3=x6+y6=1.1in;
z4=1/3[z3,z6];
z5=2/3[z3,z6];
z20=whatever[z1,z3]=whatever[z2,z4];
z30=whatever[z1,z4]=whatever[z2,z5];
z40=whatever[z1,z5]=whatever[z2,z6];
draw z1--z20--z2--z30--z1--z40--z2;
pickup pencircle scaled 1pt;
draw z1--z2;
draw z3--z6;
endfig;
```



Obrázek 1.15: Použití lineárních rovnic v METAPOSTu.

Zprostředkující vazba může být užita k vyjádření, že některý bod je na neznámé pozici na přímce určené dvěmi známými body. Například můžeme zavést novou proměnnou `aa` a napsat něco jako

```
z20=aa[z1,z3];
```

Toto vyjadřuje, že `z20` je umístěn v nějakém neznámém poměru `aa` ze spojnice mezi body `z1` a `z3`. Další taková rovnice obsahující jinou přímku je schopna stanovit hodnotu `z20`. K vyjádření, že `z20` je na průsečíku přímek `z1-z3` a `z2-z4`, zavádíme jinou proměnnou `ab` a určujeme

```
z20=ab[z2,z4];
```

Toto dovoluje METAPOSTu řešit rovnice pro `x20`, `y20`, `aa`, a `ab`.

Vymýšlet nové názvy proměnných jako `aa` a `ab` je poněkud obtížné. Tomu se lze vyhnout pomocí makra `whatever`. Toto makro generuje vždy novou anonymní proměnnou. Tedy příkaz

```
z20=whatever[z1,z3]=whatever[z2,z4]
```

nastavuje `z20` stejně jako předtím, až na to, že používá `whatever` ke generování dvou *rozdílných* anonymních proměnných místo `aa` a `ab`. Toto je způsob, jakým se na obrázku 1.15 určuje `z20`, `z30` a `z40`.

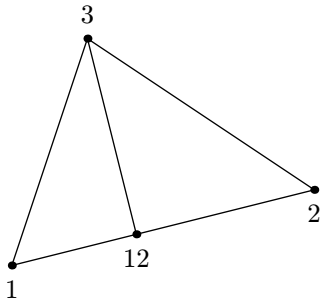
Na obrázku 1.16 je použito makro `whatever` pro vykreslení výšky trojúhelníka.

1.4.2 Operace s neznámými

Systém rovnic, jakých bylo použito v obrázku 1.15 může být dán v libovolném pořadí, pokud jsou všechny rovnice lineární. Všechny proměnné mohou být určeny dříve, než jsou potřeba. Znamená to, že rovnice

```
z1=-z2=(.2in,0);
x3=-x6=.3in;
```

```
beginfig(16);
z1=origin;
z2=(40mm,10mm);
z3=(10mm,30mm);
z12=whatever[z1,z2];
z12=z3+whatever*dir(angle(z2-z1)-90);
draw z12--z3--z1--z2--z3;
dotlabels.bot(1,2,12);
dotlabels.top(3);
endfig;
```



Obrázek 1.16: Použití `whatever` pro vykreslení výšky trojúhelníka.

```
x3+y3=x6+y6=1.1in;
z4=1/3[z3,z6];
z5=2/3[z3,z6];
```

postačují k určení `z1` až `z6`, bez ohledu na to, v jakém pořadí jsou rovnice zadány. Na druhé straně

```
z20=whatever[z1,z3]
```

je přípustné pouze tehdy, když předtím byla určena známá hodnota rozdílu `z3 − z1`, protože rovnice je ekvivalentní s

```
z20 = z1 + whatever*(z3-z1)
```

a lineární požadavek nedovoluje násobení neznámých složek `z3 − z1` anonymním výsledkem `whatever`. Obecné pravidlo je, že nemůžete násobit dvě neznámé veličiny, dělit neznámou veličinou a ani v příkazu `draw` nemůže být použita neznámá veličina. Vzhledem k tomu, že jsou povoleny pouze lineární rovnice, METAPOSTový překladač umí snadno řešit rovnice a sledovat, které hodnoty jsou známé.

Nejpřirozenější způsob jak zaručit, že METAPOST dokáže pracovat s výrazy typu

```
whatever[z1,z3]
```

je zaručit, že proměnné `z1` a `z3` jsou obě známé. Toto však není vždy nutné, protože METAPOST může být schopen odvodit známou hodnotu pro `z3 − z1`, aniž by byla kterákoliv z `z1` a `z3` předem známa. Například, METAPOST připouští rovnice

```
z3=z1+(.1in,.6in);  z20=whatever[z1,z3];,
```

ale není schopen přesně stanovit jakoukoli ze složek `z1`, `z3` nebo `z20`.

Tyto rovnice dávají částečnou informaci o `z1`, `z3` a `z20`. Dobrý způsob jak to ukázat, je zadat další rovnici jako

```
x20-x1=(y20-y1)/6;
```

To vyvolá chybové hlášení „`! Redundant equation .`“ METAPOST předpokládá, že se mu pokoušíte sdělit něco nového, tak vás bude varovat, když zadáte nadbytečnou rovnici. Jestliže by nová rovnice byla

```
(x20-x1)-(y20-y1)/6=1in;
```

chybové hlášení bude

```
! Inconsistent equation (off by 71.99979).
```

Toto chybové hlášení znázorňuje zaokrouhlovací chybu v METAPOSTovém mechanismu řešení lineárních rovnic. Zaokrouhlovací chyba obvykle není vážný problém. Ale pravděpodobně způsobí potíže, pokud se pokusíte hledat průsečík dvou téměř rovnoběžných přímek.

1.5 Výrazy

Nyní se více systematicky zaměříme na METAPOSTový jazyk. Ukázali jsme, že existují číselné proměnné a páry souřadnic a ty mohou být kombinovány do určité cesty pro příkazy `draw`. Také jsme ukázali, které proměnné mohou být použity v lineárních rovnicích. Nehovořili jsme však o všech operacích a typech dat, které se mohou v rovnicích vyskytovat.

Je možné experimentovat s výrazy zahrnující jakýkoliv níže uvedený typ dat užitím příkazu

`show` ⟨expression⟩

k vybídnutí METAPOSTu k tisku symbolické reprezentace hodnoty každého výrazu.

Jsou-li číselné hodnoty známé, je každá zapsána na nové řádce uvedené „>>“. Další typy výsledků jsou zapsány podobně kromě složitých hodnot, které někdy nejsou zapsány ve standardním výstupu. Toto vytváří odkaz na logovací soubor, který vypadá takto:

```
>> picture (see the transcript file)
```

Jestliže chcete, aby na váš terminál byly vypsány úplné výsledky příkazu `show`, přiřaďte vnitřní proměnné `tracingonline` kladnou hodnotu. Pokud si chcete vyzkoušet interaktivní zadávání výrazů na vašem počítači, můžete si vytvořit následující krátký soubor, nazvaný `expr.mp`:

```
string s[]; s1="abra";
path p[]; p1=(0,0)..(3,3); p2=(0,0)..(3,3)..cycle;
tracingonline:=1; scrollmode;
forever: message "gimme an expr: "; s0:=readstring;
show scantokens s0; endfor
```

Pokud čtete tuto kapitolu poprvé, nepotřebujete rozumět tomu, co je v souboru `expr.mp`. Řádek 1 deklaruje, že všechny proměnné ve tvaru s_k budou typu `string` a nastavuje s_1 na hodnotu `"abra"`. Řádek 2 deklaruje, že všechny proměnné ve tvaru p_k budou typu `path` a nastavuje p_1 a p_2 na jednoduché cesty. Řádek 3 říká METAPOST píše informace současně na terminál i do logovacího souboru; je taky nastaven `scrollmode`, který znamená, že se počítač nezastaví po chybových hláškách. Řádky 4 a 5 nastavují nekonečnou smyčku ve které METAPOST čte výrazy z terminálu a vrací jejich odpovídající hodnoty. Některé příklady jsou uvedeny v tabulce 1.1

Váš zápis	*Výsledek*	*Váš zápis*	*Výsledek*
1/(1/3)	3.00005	4096	4095.99998 (chyba)
infinity	4095.99998	1000*1000	32767.99998 (chyba)
infinity+epsilon	4096	100*100	10000
.1(100*100)	1000.06104	b+a	a+b
b+a−2b	a−b	1/3[a,b]	0.66667a+0.33333b
0[a,b]	a	a*b	b (chyba)
1/b	b (chyba)	sqrt 100*100	1000
sqrt(100*100)	100	0<1	true
a+1>a	true	a>=b	false (chyba)
”abc”<=”b”	true	(1,2)<>(0,4)	true
(1,2)<(0,4)	false	(1,a)>(0,b)	true
numeric a	true	known a	false
not pen a	true	(0>1) or (a<a)	false
0>1 or a<a	a (chyba)	max(1,−2,4)	4
min(1,−2,4)	−2	max(”a”,”b”,”ab”)	”b”
min(”a”,”b”,”ab”)	”a”	max((1,5),(0,6),(1,4))	(1,5)
min((1,5),(0,6),(1,4))	(0,6)	max(.5a+1,.5a−1)	0.5a+1
floor −3.14159	−4	ceiling −3.14159	−3
round(3.5,−3.5)	(4,−3)	abs(3,4)	5
length(3,4)	5	3++4	5
5+−+4	3	dir −90	(0,−1)
angle(1,1)	45	(1,2)+(3,4)	(4,6)
1/3(3,10)	(1,3.33333)	z2−z1	(−x1+x2,−y1+y2)
z xscaled 2 yscaled 1/2	(2x,0.5y)	z shifted (2,3)	(x+2,y+3)
(1,2)*(3,4)	(3,4) (chyba)	(1,2)zscaled(3,4)	(−5,10)
(a,b)dotprod(3,4)	3a+4b	s1&”cad”&s1	”abracadabra”
length s1	4	length p2	2
point .5 of p1	(1.5,1.5)	point infinity of p1	(3,3)
point −1 of p1	(0,0)	direction 1 of p2	(−4,4)

Tabulka 1.1: Příklad interaktivního zadávání příkazů.

1.5.1 Datové typy

Každý objekt v METAPOSTu má svůj datový typ. METAPOST podporuje následující datové typy: `numeric` pro uložení čísla, `pair` pro uložení souřadnic bodu, `path` pro uložení cesty, tedy křivky, `transform` pro uložení afinní transformace, `color` pro uložení barvy, `string` pro uložení řetězce, `boolean` pro uložení proměnné booleovského typu, `picture` pro uložení obrázku a `pen` pro uložení pera.

Čísla v METAPOSTu jsou reprezentována v aritmetice s pevnou řádovou čárkou jako celočíselné násobky zlomku $\frac{1}{65536}$. Jejich absolutní hodnota musí být menší než 4096, ale průběžné výsledky mohou být až osmkrát větší. Toto by neměl být problém, protože 4096 PostScriptových bodů je více než 1.4 metrů. Jestliže potřebujete pracovat s čísly o velikosti 4096 nebo větší, nastavení vnitřní proměnné `warningcheck` na nulu potlačí varovné zprávy o velkých numerických veličinách.

Souřadnice bodů jsou popisovány dvojicí čísel. Viděli jsme, že souřadnice se užívají v příkazech `draw`. Souřadnice mohou být vzájemně sčítány, odčítány, užity ve zprostředkující vazbách nebo násobeny a děleny čísly.

Cesty byly už probírány v souvislosti s příkazy `draw`, ale nebylo zmíněno, že jsou to objekty první třídy, které mohou být ukládány a dále zpracovány. Cesta reprezentuje lomenou čáru nebo křivku, danou parametrickým vyjádřením.

Další datový typ reprezentuje libovolnou afinní transformaci. Transformací může být libovolná kombinace změn velikosti, otáčení, zkosení a posunutí. Transformace může být aplikována na cesty, obrázky, pera a transformace.

Datový typ `color` je podobný souřadnicovému typu, jen má tři složky místo dvou. Stejně jako souřadnice, mohou se i barvy navzájem sčítat, odčítat, dále násobit a dělit reálným číslem. Barvy mohou být také určeny předdefinovanými konstantami `black`, `white`, `red`, `green`, `blue`, nebo mohou být zadány explicitně červená, zelená a modrá složka. Černá je `(0,0,0)` a bílá je `(1,1,1)`. Úroveň šedé `(.4,.4,.4)` může být určena pomocí `0.4white`. Neexistuje omezení na barvy, jen žádná ze složek nesmí překročit interval $[0, 1]$. Barva je zadána v PostScriptovém výstupním souboru. METAPOST řeší lineární rovnice obsahující barvy stejně jako je řeší pro cesty.

Řetězce jsou reprezentovány posloupností znaků. Proměnné typu řetězec jsou zadávány v uvozovkách `"takto"`. Řetězce nemohou obsahovat uvozovky nebo odřádkování, ale lze konstruovat řetězce obsahující libovolnou posloupnost osmibitových znaků.

Booleovské proměnné nabývají hodnot `true` a `false` a existují pro ně operátory `and`, `or`, `not`. Vztahy `=` a `<>` testují všechny typy objektů, zda se rovnají či nerovnají. Porovnávací vztahy `<`, `<=`, `>` a `>=` jsou definovány lexikograficky pro řetězce a obvyklým způsobem pro čísla. Relace jsou také definovány pro booleovské proměnné, souřadnice, barvy a transformace, ale srovnávací pravidla zde nemá cenu probírat.

Výsledky příkazů pro kreslení jsou ukládány do speciálních proměnných typu `picture`. Příkaz `draw` ukládá své výsledky do proměnné nazvané `currentpicture`. Obrázky lze přidávat k jiným obrázkům a je možné na ně aplikovat afinní transformace.

Poslední datový typ je pero. Hlavní funkcí pera v METAPOSTu není jen stanovení tloušťku čáry, ale pero může být také použito k dosažení kaligrafických efektů. Příkaz

`pickup` ⟨pen expression⟩

⟨primary⟩ → ⟨variable⟩
| `(`⟨expression⟩`)`
| ⟨nullary op⟩
| ⟨of operator⟩⟨expression⟩`of`⟨primary⟩
| ⟨unary op⟩⟨primary⟩
⟨secondary⟩ → ⟨primary⟩
| ⟨secondary⟩⟨primary binop⟩⟨primary⟩
⟨tertiary⟩ → ⟨secondary⟩
| ⟨tertiary⟩⟨secondary binop⟩⟨secondary⟩
⟨expression⟩ → ⟨tertiary⟩
| ⟨expression⟩⟨tertiary binop⟩⟨tertiary⟩

Schéma 1.2: Souhrnná syntaxe pro výrazy

způsobí, že dané pero bude použito v následujících `draw` příkazech. Normálně je ⟨pen expression⟩ ve tvaru

`pencircle scaled` ⟨numeric primary⟩.

Toto definuje kruhové pero, které vytváří čáry s konstantní tloušťkou. Má-li být dosaženo kaligrafických efektů, lze nastavit pero na eliptické nebo polygonální.

1.5.2 Operátory

Vytvářet výrazy z devíti základních typů proměnných lze mnoha různými způsoby, ale většina operací odpovídá zcela jednoduché syntaxi se čtyřmi úrovněmi priorit, jak je vidět na schématu 1.2. Existují úrovně primární, sekundární, terciální a výrazy každého ze základních typů, takže pravidla syntaxe by mohly být zaměřena na práci s položkami jako ⟨numeric primary⟩, ⟨boolean tertiary⟩, atd. To umožňuje, že výsledný typ operace závisí na volbě operátoru a typu jeho operandů. Například vztah `<` je ⟨tertiary binary⟩, který může být aplikován na ⟨numeric expression⟩ a ⟨numeric tertiary⟩ dává ⟨boolean expression⟩. Stejný operátor může přijmout další typy operandů jako ⟨string expression⟩ a ⟨string tertiary⟩, ale pokud se typy operandů nebudou rovnat, vypíše se zpráva o chybě.

Operátory násobení a dělění `*` a `/` jsou příkladem toho, co se ve schématu 1.2 nazývá ⟨primary binop⟩. Každý může akceptovat dva číselné operandy nebo jeden číselný operand a jeden operand typu souřadnice nebo barva. Operátor umocňování `**` je ⟨primary binop⟩, který požaduje dva číselné operandy. Jeho umístění ve stejné úrovni priorit jako násobení a dělení má nešťastný důsledek, že `3*a**2` znamená $(3a)^2$, místo $3(a^2)$. Protože negace je aplikována v primární úrovni, mění se také význam `-a**2` na $(-a)^2$. Naštěstí, odčítání má nižší prioritu, takže `a-b**2` znamená $a - (b^2)$ místo $(a - b)^2$.

Další ⟨primary binop⟩ je operátor `dotprod`, který počítá skalární součin dvou párů souřadnic. Tedy `z1 dotprod z2` je ekvivalentní s `x1*y1 + x2*y2`.

Operátory sčítání `+` a `-` jsou ⟨secondary binops⟩ a pracují s čísly, souřadnicemi nebo barvami a vytváří výsledky stejného typu. Další operátory, které patří do této kategorie jsou „Pythagorické sčítání“ `++` a „Pythagorické odčítání“ `+-+`: `a++b` znamená

$\sqrt{a^2+b^2}$ a `a+-+b` znamená $\sqrt{a^2-b^2}$. Existuje mnoho dalších operátorů, které bychom mohli vyjmenovat. Nejdůležitější jsou booleovské operátory `and` a `or`. Operátor `and` je ⟨primary binop⟩ a operátor `or` je ⟨secondary binop⟩.

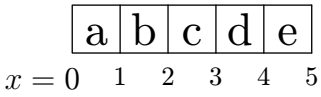
Základní operace s řetězci jsou slučování a tvorba podřetězců. ⟨tertiary binop⟩ `&` provádí slučování; například,

```
"abc" & "de"
```

vytváří řetězec `"abcde"`. Pro konstrukci podřetězců, je ⟨of operator⟩ `substring` užíván takto:

`substring` ⟨pair expression⟩ `of` ⟨string primary⟩

⟨pair expression⟩ určuje, která část řetězce bude vybrána. Za tímto účelem, je řetěz oindexován přirozenými čísly, která jsou umístěna *mezi* znaky. Lze si to představit tak, že řetězec je zapsán na čtverečkovaném papíře tak, že první znak zabírá místo mezi nulou a jedničkou na x ové ose a následující znak zaujímá místo v rozmezí $1 \leq x \leq 2$, atd. Tedy řetězec `"abcde"` by měl být myšlen takto



a `substring (2,4) of "abcde"` je `"cd"`.

Některé operátory nemají žádné argumenty. Příklad toho, co se na schématu 1.2 nazývá ⟨nullary op⟩ je `nullpicture`, který vrací zcela prázdný obrázek.

Základní syntaxe na schématu 1.2 zahrnuje pouze aspekty syntaxe výrazů, které jsou relativně typově-nezávislé. Například, složitá syntaxe cesty daná na schématu 1.1 dává alternativní pravidla pro tvorbu ⟨path expression⟩. Další pravidlo

⟨path knot⟩ → ⟨pair tertiary⟩ | ⟨path tertiary⟩

vysvětluje význam ⟨path knot⟩ na schématu 1.1. To znamená, že cesta

```
z1+(1,1){right}..z2
```

nepotřebuje závorky kolem `z1+(1,1)`.

1.5.3 Zlomky, zprostředkování a unární operátory

Zprostředkující vazby nejsou uvedeny v základní syntaxi výrazů na schématu 1.2. Tyto zprostředkující vazby jsou analyzovány v ⟨primární⟩ úrovni, a tak obecné pravidlo pro jejich vytváření je

⟨primary⟩ → ⟨numeric atom⟩`[`⟨expression⟩`,`⟨expression⟩`]`

kde každý ⟨výraz⟩ může být typu číslo, souřadnice nebo barva. ⟨numeric atom⟩ ve zprostředkující vazbě je zvlášť jednoduchý typ z ⟨numeric primary⟩, jak se ukázáno na schématu 1.3. Význam toho všeho je, že počáteční parametr ve zprostředkující vazbě musí být umístěn v závorkách pokud to není proměnná, kladné číslo nebo kladný zlomek. Například výrazy

`-1[a,b]` a `(-1)[a,b]`

⟨numeric primary⟩ → ⟨numeric atom⟩
| ⟨numeric atom⟩`[`⟨numeric expression⟩`,`⟨numeric expression⟩`]`
| ⟨of operator⟩⟨expression⟩`of`⟨primary⟩
| ⟨unary op⟩⟨primary⟩
⟨numeric atom⟩ → ⟨numeric variable⟩
| ⟨number or fraction⟩
| `(`⟨numeric expression⟩`)`
| ⟨numeric nullary op⟩
⟨number or fraction⟩ → ⟨number⟩`/`⟨number⟩
| ⟨number not followed by '`/`⟨number⟩'⟩

Schéma 1.3: Syntaktická pravidla pro ⟨numeric primary⟩

jsou velmi rozdílné: první je $-b$ protože je ekvivalentní s výrazem `-(1[a,b])`; druhý je $a-(b-a)$ tedy $2a-b$.

Za povšimnutí v syntaktických pravidlech na schématu 1.3 stojí, že vazba operátoru `/` je nejtěsnější, jsou-li jeho operandy čísla. Tedy `2/3` je ⟨numeric atom⟩ zatímco `(1+1)/3` je pouze ⟨numeric secondary⟩. Použití ⟨primary binop⟩ jako `sqrt` objasňuje rozdíl:

```
sqrt 2/3
```

znamená $\sqrt{\frac{2}{3}}$ zatímco

```
sqrt(1+1)/3
```

znamená $\sqrt{2}/3$. Operátory jako `sqrt` mohou být zapsány ve standardním funkcionálním zápisu, ale často je zbytečné argumenty závorkovat. Toto platí pro každou funkci, která je analyzována jako ⟨primary binop⟩. Například `abs(x)` a `abs x` obojí počítá absolutní hodnotu z `x`. Totéž platí pro funkce `round`, `floor`, `ceiling`, `sind` a `cosd`. Poslední dvě z nich počítají trigonometrické funkce úhlů ve stupních.

Ne všechny unární operátory pracují s číselnými argumenty a vrací číselné výsledky. Například operátor `abs` může být aplikován na pár souřadnic k výpočtu eukleidovské délky vektoru. Použití operátoru `unitvector` na pár souřadnic vytváří stejný pár přepočtený na jednotkovou eukleidovskou délku, viz obr. 1.10. Operátor `decimal` zpracovává číslo a vrací řetězec. Operátor `angle` (viz obr. 1.10) pracuje se souřadnicemi a počítá dvouparametrický arkustangens; tj. `angle` je inverzí operátoru `dir`, který byl zmíněn v kapitole 1.3.2. Existuje také operátor `cycle`, který požaduje ⟨path primary⟩ a vrací booleovské výsledky označující, zda je cesta uzavřená křivka.

Celá třída dalších operátorů řadí výrazy a vrací booleovské výsledky. Typy označené jako `pair` mohou operovat se všemi ⟨primary⟩ typy a vrací booleovské výsledky označující, zda argument je `pair`. Podobně každý z následujících operátorů může být použit jako unární operátor: `numeric`, `boolean`, `color`, `string`, `transform`, `path`, `pen` a `picture`. Kromě právě testovaných ⟨primary⟩ typů, můžete použít operátory `known` a `unknown` k otestování, zda se jedná o známou hodnotu.

V určitém kontextu se může i číslo chovat jako operátor. Toto se vztahuje na trik, který umožňuje `3x` a `3cm` použít jako alternativu k `3*x` a `3*cm`. Pravidlo je, že ⟨číslo nebo

zlomek⟩, za kterým nenásleduje `+`, `-`, nebo další ⟨číslo nebo zlomek⟩ může sloužit jako ⟨primary binop⟩. Takto `2/3x` jsou dvě třetiny z `x` ale `(2)/3x` je $\frac{2}{3x}$ a `3 3` je nepřípustné.

Existují také operátory, které vyjímají číselné hodnoty (části) ze souřadnic, barev a dokonce i transformací. Jestliže `p` je ⟨pair primary⟩, `xpart p` a `ypart p` vyjímá její složky tak, že

```
(xpart p, ypart p)
```

je ekvivalentní s `p` i když `p` je neznámý pár souřadnic, který byl použitý v lineární rovnici. Podobně barva `c` je ekvivalentní s

```
(redpart c, greenpart c, bluepart c)
```

Specifikátory částí transformací budou probrány později.

1.6 Proměnné

METAPOST dovoluje kombinovat názvy proměnných jako x.a, `x2r`, `y2r`, a `z2r`, kde `z2r` znamená `(x2r,y2r)` a `z.a` znamená `(x.a,y.a)`. Ve skutečnosti existuje široká třída takových přípon, že `z`⟨suffix⟩ znamená

$$(x\langle\text{suffix}\rangle,\ y\langle\text{suffix}\rangle).$$

Protože ⟨suffix⟩ je tvořen tokeny, začneme několika poznámkami o tokenech.

1.6.1 Tokeny

METAPOSTový vstupní soubor je zpracováván jako posloupnost čísel, řetězců a symbolických tokenů. Číslo se skládá z posloupnosti číslic, která může obsahovat desetinnou čárku. Technicky vzato je znaménko minus před záporným číslem samostatný token. Protože METAPOST užívá aritmetiku s pevnou čárkou, nerozumí exponenciálnímu zápisu `6.02E23`. METAPOST to bude interpretovat jako číslo 6.02, následované symbolickým tokenem `E` a číslem 23.

Všechno mezi dvojicí uvozovek `"` je řetězec . Je nepřípustné, aby řetězec začínal na jedné řádce a končil na další. Řetězec také nemůže obsahovat uvozovky `"` ani nic jiného, než tisknutelné ASCII znaky. Pokud přesto chceme uvozovky vysázet, použijeme předdefinovanou konstantu `ditto`.

Všechno ostatní na vstupní řádce, kromě čísel a řetězců, se rozpadne na symbolické tokeny. Symbolický token je posloupnost jednoho nebo více podobných znaků, kde znaky jsou „podobné“, jestliže se vyskytují ve stejné řádce tabulky 1.2.

Tedy `A_alpha` a `+-+` jsou symbolické tokeny, ale `!=` je interpretováno jako dva tokeny a `x34` je symbolický token následovaný číslem. Protože jsou závorky `[` a `]` vyjmenovány na řádkách pod sebou, symbolické tokeny, které je obsahují jsou `[`, `[[`, `[[[`, atd. a `]`, `]]`, atd.

Některé tokeny nejsou uvedeny v tabulce 1.2, protože potřebují zvláštní zpracování. Čtyři znaky `,;()` jsou „samotáři“: každá čárka, středník nebo kulatá závorka je samostatná část, i když se vyskytují po sobě. Tedy `(())` jsou čtyři tokeny a ne jeden nebo

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz
:<=>|
#&@$
/*\
+-
!?
,‘
^~
{}
[
]
```

Tabulka 1.2: Řetězcové třídy pro tokeny

dva. Znak procento je velmi speciální, protože uvozuje komentáře. Znak procento a vše, co je za ním až do konce řádky, je ignorováno.

Další speciální znak je tečka. Dvě a více teček společně vytváří společně symbolický token, ale samostatná tečka je ignorována a tečka uvedená před nebo za cifrou je součástí čísla. Tedy `..` a `...` jsou symbolické tokeny, zatím co `a.b` jsou dva tokeny `a` a `b`. Je obvyklé použít tečky k oddělení tokenů v této podobě, když je jmenovaná proměnná delší než jeden token.

1.6.2 Deklarace proměnných

Jméno proměnné je symbolický token nebo posloupnost symbolických tokenů. Většina symbolických tokenů může být jménem proměnné, ale žádný předdefinovaný význam jako `draw`, `+` nebo `..` není přípustný; to znamená, že jména proměnných nemohou být makra ani primitivy METAPOSTu. Toto menší omezení dovoluje překvapivě širokou třídu názvů proměnných: `alpha`, `==>`, `@&#$&` a `~~` jsou všechno přípustné názvy proměnných. Takové symbolické tokeny bez speciálního významu se nazývají *tagy*.

Jméno proměnné může být posloupnost tagů jako `f.bot` nebo `f.top`. Ideou je poskytnout některý ze zápisů Pascalu nebo struktur C. Je také možné simulovat pole použitím názvů proměnných, které obsahují čísla stejně jako symbolické tokeny. Například název proměnné `x2r` obsahuje tag `x`, číslo 2 a tag `r`. Proměnná se může také jmenovat `x3r` nebo dokonce `x3.14r`. Tyto proměnné mohou být považovány za pole prostřednictvím následující konstrukce `x[i]r`, kde `i` má odpovídající číselnou hodnotu. Souhrnná syntaxe pro názvy proměnných je ukázána na schématu 1.4.

Proměnné jako `x2` a `y2` přibírají číselné hodnoty standardně. Tak můžeme využít skutečnosti, že `z`⟨suffix⟩ je zkratka pro

$$(x\langle\text{suffix}\rangle,\, y\langle\text{suffix}\rangle,\,)$$

ke generování párových proměnných, je-li to potřeba. Makro `beginfig` vymaže předcházející hodnoty tím, že začíná tagy `x` nebo `y` tak, že bloky `beginfig` ... `endfig`

mezi sebou neinterferují. Jinými slovy proměnné, které začínají `x`, `y`, `z`, jsou lokální v obrázku, který je využívá. Obecný mechanismus pro tvorbu lokálních proměnných bude probírán v kapitole 1.9.1.

Typ deklarací umožňuje používat téměř všech názvů, přičemž maže všechny předchozí hodnoty, které by mohly způsobit interferenci. Například deklarace

```
pair pp, a.b;
```

dělá z `pp` a `a.b` neznámé páry souřadnic. Taková deklarace není striktně lokální, protože `pp` a `a.b` nejsou automaticky obnoveny na jejich původní hodnotu na konci aktuálního obrázku. Samozřejmě, že jsou znovu nastaveny na neznámé páry souřadnic, jestliže se deklarace opakuje.

Deklarace jsou stejné pro ostatních osm typů: `numeric`, `path`, `transform`, `color`, `string`, `boolean`, `picture` a `pen`. Jediné omezení je, že v deklaracích proměnných nemůžete zadávat explicitní číselné indexy. Nezadávejte neplatnou deklaraci

```
numeric q1, q2, q3;
```

místo toho k deklaraci celého pole použijte symbol pro všeobecný index `[]`:

```
numeric q[];
```

Můžete také deklarovat „vícerozměrná“ pole. Po deklaraci

```
path p[]q[], pq[][];
```

jsou obě `p2q3` i `pq1.4 5` cesty.

Vnitřní proměnné jako `tracingonline` nemohou být deklarovány v normálním tvaru. Všechny vnitřní proměnné probírané v tomto manuálu jsou předem definované a nemusí být deklarovány. Existuje však způsob, jak deklarovat proměnnou, která by se měla chovat jako nově vytvořená vnitřní proměnná. Deklarace je `newinternal` následovaná seznamem symbolických tokenů. Například

```
newinternal a, b, c;
```

způsobuje, že `a`, `b` a `c` se chovají jako vnitřní proměnné. Takové proměnné mají vždy známé číselné hodnoty, které mohou být změněny pouze přiřazovacím operátorem `:=`. Počáteční hodnota vnitřních proměnných je nula až na to, že balík maker Plain dává některým z nich nenulové počáteční hodnoty. (Makra Plain jsou automaticky předem načtena, jak je vysvětleno v kapitole 1.1.)

⟨variable⟩ → ⟨tag⟩⟨suffix⟩
⟨suffix⟩ → ⟨empty⟩ | ⟨suffix⟩⟨subscript⟩ | ⟨suffix⟩⟨tag⟩
⟨subscript⟩ → ⟨number⟩ | `[`⟨numeric expression⟩`]`

Schéma 1.4: Syntaxe pro názvy proměnných.

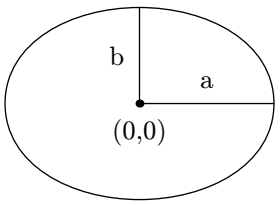
1.7 Vkládání textů a obrázků

METAPOST má několik způsobů, jak vložit do obrázku text nebo obrázek. Nejjednodušší způsob, jak to udělat, je použít příkaz `label`

`label`⟨label suffix⟩`(`⟨string or picture expression⟩`,` ⟨pair expression⟩`);`

Výraz ⟨string or picture expression⟩ vyjadřuje, co má být zobrazeno a ⟨pair expression⟩ udává pozici. Není-li uveden ⟨label suffix⟩, střed textu nebo obrázku je umístěn na zadané souřadnice. Jestliže popíšete nějakou část obrázku, pravděpodobně chcete vyrovnat popisky tak, aby se nepřekrývaly. Toto je znázorněno na obrázku 1.17, kde `"a"` je umístěno nad střed úsečky a `"b"` je nalevo od středu úsečky. Toho je dosaženo užitím `label.top` pro `"a"` a `label.lft` pro `"b"`, jak je ukázáno na obrázku.

```
beginfig(17);
a=.7in; b=.5in;
z0=(0,0);
z1=-z3=(a,0);
z2=-z4=(0,b);
draw z1..z2..z3..z4..cycle;
draw z1--z0--z2;
label.top("a", .5[z0,z1]);
label.lft("b", .5[z0,z2]);
dotlabel.bot("(0,0)", z0);
endfig;
```



Obrázek 1.17: Ukázka použití příkazů `label` a `dotlabel`.

⟨label suffix⟩ určuje pozici textu vzhledem k daným souřadnicím. Celková množina možností je

⟨label suffix⟩ → ⟨empty⟩ | `lft` | `rt` | `top` | `bot` | `ulft` | `urt` | `llft` | `lrt`,

kde `lft` a `rt` znamená vlevo a vpravo a `llft`, `ulft`, atd. znamená dole vlevo, nahoře vlevo atd. Skutečná vzdálenost, o kterou bude text vzdálen v jakémkoliv směru od zadaných souřadnic, je určena vnitřní proměnnou `labeloffset` (implicitně nastavená na 3`bp`). Obrázek 1.18 názorně ukazuje polohu textu nebo obrázku při použití jednotlivých přípon a dále použití proměnné `labeloffset`.

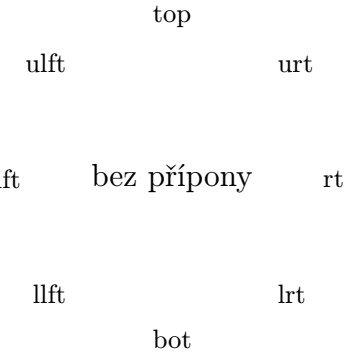
Obrázek 1.17 také objasňuje příkaz `dotlabel`. Je stejný jako příkaz `label` až na to, že přidává tečku v zadaných souřadnicích. Například

```
dotlabel.bot("(0,0)", z0)
```

umístí tečku v `z0` a pod tento bod umístí „(0,0)“. Velikost tečky řídí vnitřní proměnná `dotlabeldiam`, která je implicitně nastavena na 3`bp`. Další možností je použít makro `thelabel`. To má stejnou syntaxi jako příkazy `label` a `dotlabel` až na to, že vrací text jako ⟨picture primary⟩ místo toho, aby ho skutečně vykreslilo. Tedy

```
label.bot("(0,0)", z0)
```

```
beginfig(18)
labeloffset:=2cm;
z1=(0,0);
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);
endfig;
```



Obrázek 1.18: Přípony používané při umísťování textu.

je ekvivalentní s

```
draw thelabel.bot("(0,0)", z0)
```

Pro jednoduché aplikace popisování obrázků můžete normálně vystačit s `label` a `dotlabel`. Můžete také používat zkrácenou formu příkazu `dotlabel`, která vás ušetří mnoha psaní, jestliže máte mnoho bodů `z0`, `z1`, `z.a`, `z.b`, atd. a chcete jako popisy použít přípony `z`. Příkaz

```
dotlabels.rt(0, 1, a);
```

je ekvivalentní s

```
dotlabel.rt("0",z0); dotlabel.rt("1",z1); dotlabel.rt("a",z.a);
```

Tedy argumentem `dotlabels` je seznam přípon, které jsou pro proměnnou `z` známy a ⟨label suffix⟩ zadaný s `dotlabels` je použit k umístění popisů všech bodů.

Pokud máme přípon mnoho a jsou pouze číselné, můžeme s výhodou použít příkaz

`range` ⟨numeric⟩ `thru` ⟨numeric⟩

který nám vrátí čárkou oddělený seznam čísel. Například `range 5 thru 7` vrátí `5,6,7`.

Existuje také příkaz `labels`, který je analogický s `dotlabels`, ale jeho použití nedoporučujeme, protože jsou zde problémy kompatibility s METAFONTem. Některé verze balíku maker Plain definují `labels` synonymně s `dotlabels`.

Proměnná `defaultfont` určuje font textu užívaný v příkazech `label` a `dotlabel`. Počáteční hodnota `defaultfont` je `"cmr10"`, ale může být změněna přiřazením

```
defaultfont:="Times-Roman"
```

Existuje také číselná veličina nazvaná `defaultscale`, která určuje velikost. Jestliže je `defaultscale` 1, dostanete „normální velikost“, která je obvykle 10 bodů, ale ta může být také změněna. Například

```
defaultscale := 1.2
```

zvětšuje text o dvacet procent. Neznáte-li normální velikost a chcete si být jisti, že velikost textu bude, řekněme 12 bodů, můžete použít operátor `fontsize` ke stanovení normální velikosti:

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

Jestliže chcete změnit `defaultfont`, název nového fontu by měl být takový, který zná TEX protože MetaPost získá informaci o výšce a šířce přečtením souboru `tfm`. (Toto je vysvětleno v *The TEXbook*. [Knu86b]). Mělo by být možné užít vestavěné PostScriptové fonty, ale jejich názvy závisí na systému. Některé systémy mohou užívat `rptmr` nebo `ps-times-roman` místo `Times-Roman`. Používat TEXovský font `cmr10` je poněkud riskantní, protože neobsahuje znak mezery a některé ASCII symboly. Navíc, METAPOST neužívá ligatury a kerningové informace, které pocházejí z TEXovského fontu.

1.7.1 Sázení vlastních textů

TEX může být použit k vytváření složitějších textů. Použijeme-li

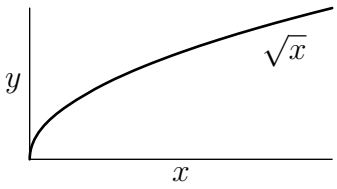
`btex` ⟨typesetting commands⟩ `etex`

v METAPOSTovém vstupním souboru, bude ⟨typesetting commands⟩ zpracován TEXem a převeden do datového typu `picture` (ve skutečnosti ⟨picture primary⟩), který může být užit v příkazech `label` nebo `dotlabel`. Mezery po `btex` a před `etex` jsou ignorovány. Například příkaz

```
label.lrt(btex $\sqrt x$ etex, (3,sqrt 3)*u)
```

na obrázku 1.19 umisťuje text $\sqrt{x}$ vpravo dolu od bodu `(3,sqrt 3)*u`.

```
beginfig(19);
numeric u;
u = 1cm;
draw (0,2u)--(0,0)--(4u,0);
pickup pencircle scaled 1pt;
draw (0,0){up}
  for i=1 upto 8: ..(i/2,sqrt(i/2))*u  endfor;
label.lrt(btex $\sqrt x$ etex, (3,sqrt 3)*u);
label.bot(btex $x$ etex, (2u,0));
label.lft(btex $y$ etex, (0,u));
endfig;
```



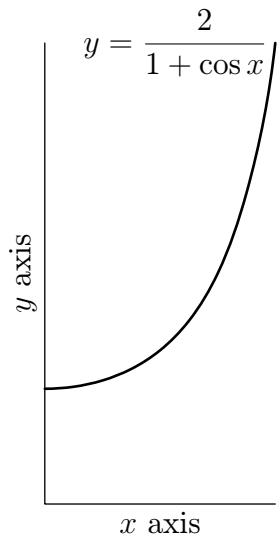
Obrázek 1.19: Použití `btex etex` při popisu obrázku.

Obrázek 1.20 ilustruje některé další komplikované věci, které lze s texty dělat. Protože výsledek `btex` ... `etex` je typu `picture`, může být zpracováván jako obrázek. Především je možné na obrázek aplikovat transformace. Dosud jsme neprobírali syntaxi, ale ⟨picture secondary⟩ může být

⟨picture secondary⟩ `rotated` ⟨numeric primary⟩

Toto je použito na obrázku 1.20 k rotaci textu „y axis“ tak, že je umístěn vertikálně.

```
beginfig(20);
numeric ux, uy;
120ux=1.2in; 4uy=2.4in;
draw (0,4uy)--(0,0)--(120ux,0);
pickup pencircle scaled 1pt;
draw (0,uy){right}
  for ix=1 upto 8:
    ..(15ix*ux, uy*2/(1+cosd 15ix))
  endfor;
label.bot(btex $x$ axis etex, (60ux,0));
label.lft(btex $y$ axis etex rotated 90,
          (0,2uy));
label.lft(
  btex $\displaystyle y={2\over1+\cos x}$ etex,
  (120ux, 4uy));
endfig;
```



Obrázek 1.20: Složitější popisy obrázku.

Další problém na obrázku 1.20 je užití zobrazené rovnice

$$y = \frac{2}{1+\cos x}$$

jako textu. Bylo by přirozenější ji zapsat takto

```
$$y={2\over 1+\cos x}$$,
```

ale to nelze, protože TEX sází text v „horizontálním módu“.

Zde je ukázáno, jak se TEXovský „materiál“ překládá do tvaru přijatelného pro METAPOST : METAPOSTový procesor přeskočí bloky `btex` ... `etex` a v závislosti na procesoru je přeloží na nízkou úroveň METAPOSTových příkazů. Jestliže je hlavní soubor `fig.mp`, přeložený TEXovský materiál je uložen do souboru `fig.mpx`. Toto se děje tiše, bez jakéhokoliv uživatelova zásahu, ale může to selhat, jestliže některý z `btex` ... `etex` bloků obsahuje chybný TEXovský příkaz. Pak se v logovacím souboru objeví hláška: „! Unable to make mpx file.“ .

TEXovské definice maker nebo některé další TEXovské příkazy mohou být uzavřeny do bloku `verbatimtex` ... `etex`. Rozdíl mezi `btex` a `verbatimtex` je, že první generuje `picture`, zatímco druhý dodává materiál pro zpracování TEXem. Například jestliže chcete, aby TEX vysázel text použitím maker definovaných v `mymac.tex`, váš METAPOSTový vstupní soubor by měl vypadat následovně:

```
verbatimtex \input mymac etex
beginfig(1);
    ⋮
label(btex ⟨TEX material using mymac.tex⟩ etex, ⟨some coordinates⟩);
    ⋮
```

V Unixových[7] systémech, může být prostředí specifikováno tak, že bloky `btex` ... `etex` a `verbatimtex` ... `etex` jsou troffovské místo TEXovských. Jestliže využijeme této možnosti, je dobré začít váš METAPOSTový vstupní soubor přiřazením `prologues:=1`. Nastavením této vnitřní proměnné na kladnou hodnotu způsobí, že výstup bude formátovaný jako „strukturovaný PostScript“ vytvořený za předpokladu, že text pochází z vestavěných PostScriptových fontů. Toto dělá METAPOSTový výstup lépe přenosným, ale má to i zajímavou nevýhodu : Obecně, když použijete TEXovské fonty, to nefunguje, protože programy, které překládají TEXovský výstup do PostScriptu potřebují vytvořit speciální předpisy pro TEXovské fonty ve vložených obrázcích a standardní PostScript pravidla s tím nepočítají. Detaily pro to, jak zahrnout PostScriptové obrázky do článku vytvořeném v TEXu nebo troffu jsou závislé na systému. Mohou být nalezeny v manuálech nebo jiných on-line dokumentacích. Soubor `dvips.tex` je distribuován elektronicky spolu s dvips TEXovským výstupním procesorem.

Do okolí `verbatimtex` ... `etex` smíme zapsat např. celou hlavičku kódu LATEXu až po `\begin{document}` (včetně). Např. pro soubor METAPOSTu, který používá češtinu pro popisy a je spouštěný pod Windows, použijeme tuto hlavičku:

```
verbatimtex
%&latex
\documentclass[12pt]{article}
\usepackage[IL2]{fontenc}
\usepackage[cp1250]{inputenc}

\begin{document}
etex
```

1.7.2 Operátor `infont`

Bez ohledu na to, zda použijete TEX nebo troff, přidávání textu do obrázků dělá základní METAPOSTový operátor `infont`. Je ⟨primary binop⟩ a bere ⟨string secondary⟩ jako levý parametr a ⟨string primary⟩ jako pravý parametr. Levý parametr je text a pravý parametr je název fontu. Výsledkem operace je ⟨picture secondary⟩, který může být různými způsoby transformován. Jednou možností je zvětšení obrázku zadaným faktorem pomocí syntaxe

⟨picture secondary⟩ `scaled` ⟨numeric primary⟩.

Tak `label("text",z0)` je ekvivalentní

`label("text" infont defaultfont scaled defaultscale, z0)`.

Není-li výhodné použít řetězec pro levý parametr `infont`, můžete použít

`char` ⟨numeric primary⟩

k výběru znaku, založeném na číselné pozici ve fontu. Tak

`char(n+64) infont "Times-Roman"`

je obrázek obsahující znak `n+64` ve fontu Times-Roman.

[7] Unix je ochranná známka Unix Systems Laboratories.

1.7.3 Měření textu

METAPOST vytváří běžné fyzikální rozměry obrázků generovaných operátorem `infont`. Existují unární operátory `llcorner`, `lrcorner`, `urcorner`, `ulcorner` a `center`, které pracují s ⟨picture primary⟩ a vrací rohy „bounding boxu“ jak je zobrazeno na obrázku 1.21. Operátor `center` lze aplikovat také na ⟨path primary⟩ a ⟨pen primary⟩. V METAPOSTu verze 0.30 a vyšší `llcorner`, `lrcorner`, atd. akceptují také všechny tři typy argumentů. Omezení na typ argumentu není pro operátory rohů moc důležité, protože jejich hlavním smyslem je umožnit příkazům `label` a `dotlabel` pořádně vycentrovat jejich text. Předdefinované makro

`bbox` ⟨picture primary⟩

nalézá obdélníkovou cestu, která reprezentuje `bounding box` daného obrázku. Je-li `p` typu picture, pak `bbox p` je ekvivalentní s

```
(llcorner p--lrcorner p--urcorner p--ulcorner p--cycle)
```

až na to , že poskytuje navíc malý prostor okolo `p`, který je určený vnitřní proměnnou `bboxmargin` (implicitně je nastavena na 2`bp`).

Obrázek 1.21: Bounding box a jeho rohové body.

Všimněte si, že METAPOST počítá bounding box z obrázku `btex ... etex` stejným způsobem jako TEX. To je celkem přirozené, ale má to jisté důsledky s ohledem na skutečnost, že TEX má funkce `\strut` a `\rlap`, které umožňují TEXovským uživatelům lhát o rozměrech boxu.

Když jsou TEXovské příkazy, které lžou o rozměrech boxu přeloženy do nízké úrovně METAPOSTového kódu, příkaz

`setbounds` ⟨picture variable⟩ `to` ⟨path expression⟩

nastaví bounding box daného ⟨picture variable⟩ na zadanou cestou. Abychom získali opravdový bounding box takového obrázku, přiřadíme kladnou hodnotu vnitřní proměnnép `truecorners`:[8] t.j.

```
show urcorner btex $\bullet$\rlap{ A} etex
```

vytváří ” `>> (4.9813,6.8078)`” zatímco

```
truecorners:=1; show urcorner btex $\bullet$\rlap{ A} etex
```

vytváří ”`>> (15.7742,6.8078)`”.

[8] Funkce `setbounds` a `truecorners` lze nalézt pouze v METAPOSTu verze 0.30 a vyšší

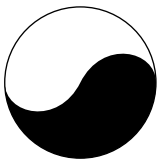
1.8 Pokročilá grafika

Všechny příklady v předchozích kapitolách byly jednoduché čáry s přidaným popisem. Tato kapitola popisuje stínování a nástroje pro generování ne zcela jednoduchých kreseb. Stínování se provádí pomocí příkazu `fill` . V nejjednodušším tvaru je příkaz `fill` aplikován na cestu, která udává hranici oblasti k vykreslení. V syntaxi

`fill` ⟨path expression⟩

by argumentem měla být uzavřená cesta, tj. cesta, která popisuje křivku pomocí zápisu `..cycle` nebo `--cycle`. Například příkaz `fill` na obrázku 1.22 sestrojí a uzavře cestu rozšířením půlkruhové cesty `p`. Tato cesta je orientovaná proti směru hodinových ručiček, ale na tom nezáleží, protože příkaz `fill` užívá PostScriptové pravidlo [Ado86] nenulového počtu závitů.

```
beginfig(22);
path p;
p = (-1cm,0)..(0,-1cm)..(1cm,0);
fill p{up}..(0,0){-1,-2}..{up}cycle;
draw p..(0,1cm)..cycle;
endfig;
```



Obrázek 1.22: Použití příkazu `fill`.

Obecně příkaz `fill`

`fill` ⟨path expression⟩ `withcolor` ⟨color expression⟩

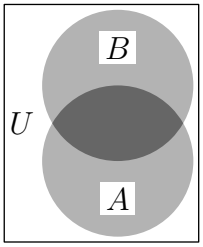
specifikuje odstín šedé, případně (máte-li barevnou tiskárnu) některou jinou barvu.

Obrázek 1.23 ilustruje několik aplikací příkazu `fill` k vyplnění plochy různými odstíny šedé. Cesty jsou překrývající se kruhy `a` a `b` a cesta `ab`, která ohraničuje průnik obou kruhů. Kruhy `a` a `b` jsou odvozeny z předdefinované cesty `fullcircle`, která aproximuje kruh s jednotkovým průměrem se středem v počátku. Existuje také předdefinovaná cesta `halfcircle`, která je částí `fullcircle` nad x-ovou osou. Cesta `ab` je inicializována pomocí makra `buildcycle`, které bude brzy zmíněno.

Vyplnění kruhu `a` světle šedou barvou `.7white`, stejně jako kruhu `b`, dvakrát vyplní oblast, kde se kruhy překrývají. Pravidlem je, že každý příkaz `fill` přiřadí danou barvu všem bodům pokryté oblasti, smaže všechno co zde předtím bylo včetně čar, textů a výplní. Je tedy důležité zadat příkazy `fill` ve správném pořadí. Ve výše uvedeném příkladu získává překrývající se oblast dvakrát stejnou barvu, zůstává tedy po prvních dvou příkazech `fill` světla šedá. Třetí příkaz `fill` přiřadí překrývající se oblasti tmavší barvu `.4white`.

V tomto bodě mají kruhy a překrývající se oblast konečné barvy, ale nejsou zde žádná přerušení textu. Přerušení (vypnutí) je dosaženo příkazy `unfill`, které účinně vymažou oblast ohraničenou `bbox pa` a `bbox pb`. Přesněji, `unfill` je zkratkou pro vyplnění `withcolor background`, kde `background` je normálně rovno `white`, což odpovídá tisku na bílý papír. Je-li to nutné, lze přiřadit novou barvu hodnotě `background`.

```
beginfig(23);
path a, b, ab;
a = fullcircle scaled 2cm;
b = a shifted (0,1cm);
ab = buildcycle(a, b);
picture pa, pb;
pa = thelabel(btex $A$ etex, (0,-.5cm));
pb = thelabel(btex $B$ etex, (0,1.5cm));
fill a withcolor .7white;
fill b withcolor .7white;
fill ab withcolor .4white;
unfill bbox pa; draw pa;
unfill bbox pb; draw pb;
label.lft(btex $U$ etex, (-1cm,.5cm));
draw bbox currentpicture;
endfig;
```



Obrázek 1.23: Použití příkazu `buildcycle`.

Před vlastním vykreslením textů, které je třeba uložit v obrázcích `pa` a `pb`, je možné změřit jejich `bounding box`. Makro `thelabel` vytvoří takové obrázky a posune je na správnou pozici tak, že jsou připraveny na vykreslení. Použitím výsledných obrázků v příkazech `draw` ve tvaru

`draw` ⟨picture expression⟩

je přidá do `currentpicture` tak, že přepíší část toho, co už bylo vykresleno. Bílé obdélníky na obrázku 1.23, vzniklé pomocí příkazu `unfill`, způsobí právě toto přepsání.

1.8.1 Sestavování uzavřených cest

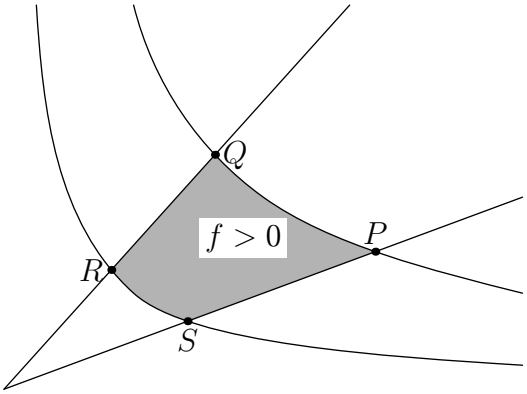
Příkaz `buildcycle` konstruuje cestu k užití maker `fill` nebo `unfill`. Jsou-li dány dvě nebo více cest, makro `buildcycle` se je snaží spojit v uzavřenou cestu.

Příklad použití makra `buildcycle` je uveden na obrázku 1.24. Argument `buildcycle` má zde více než dvě cesty a každá dvojice po sobě jdoucích cest má jediný průsečík. Například přímka `q0.5` a křivka `p2` se protínají pouze v bodě P; křivka `p2` a přímka `q1.5` se protínají pouze v bodě Q. Ve skutečnosti je každý z bodů P, Q, R, S jediným průsečíkem a výsledek

```
buildcycle(q0.5, p2, q1.5, p4)
```

uvažuje `q0.5` z S do P, pak `p2` z P do Q, pak `q1.5` z Q do R, a konečně `p4` z R zpět do S. Prozkoumání METAPOSTového kódu z obrázku 1.24 prozrazuje, že musíme jít zpět podél `p2` abychom se dostali z P do Q. Toto pracuje perfektně, jakmile jsou body průsečíků jednoznačně definovány, ale může to způsobit neočekávané výsledky, jestliže se dvojice cest protínají více než jednou.

```
beginfig(24);
h=2in; w=2.7in;
path p[], q[], pp;
for i=2 upto 4: ii:=i**2;
  p[i] = (w/ii,h){1,-ii}...(w/i,h/i)...(w,h/ii){ii,-1};
endfor
q0.5 = (0,0)--(w,0.5h);
q1.5 = (0,0)--(w/1.5,h);
pp = buildcycle(q0.5, p2, q1.5, p4);
fill pp withcolor .7white;
z0=center pp;
picture lab; lab=thelabel(btex $f>0$ etex, z0);
unfill bbox lab; draw lab;
draw q0.5; draw p2; draw q1.5; draw p4;
dotlabel.top(btex $P$ etex, p2 intersectionpoint q0.5);
dotlabel.rt(btex $Q$ etex, p2 intersectionpoint q1.5);
dotlabel.lft(btex $R$ etex, p4 intersectionpoint q1.5);
dotlabel.bot(btex $S$ etex, p4 intersectionpoint q0.5);
endfig;
```



Obrázek 1.24: Další použití příkazu `buildcycle`.

Obecné pravidlo pro makro `buildcycle` je, že

$$\texttt{buildcycle}(p_1\texttt{,}\, p_2\texttt{,}\, p_3\texttt{,}\, \ldots\texttt{,}\, p_k)$$

vybírá průsečíky mezi každou dvojicí p_i a p_{i+1} jako poslední možný na p_i a první možný na p_{i+1}. Není jednoduché pravidlo pro řešení kolizí mezi těmito dvěmi cíli. Měli byste se tedy vyhnout případu, kdy se jeden průsečík vyskytuje později na p_i a jiný průsečík se vyskytuje dříve na p_{i+1}.

Přednost pro průsečíky co nejpozdější na p_i a co nejdřívější na p_{i+1} vede k nejednoznačnému řešení pro výběr první (přední) části cesty.

Pro cyklické cesty odpovídá „dřívější“ a „pozdější“ počátečnímu/koncovému bodu, do kterého se vracíte, když užijete „`..cycle`“.

Přímější způsob, jak pracovat s průsečíky cest je přes ⟨secondary binop⟩ `intersectionpoint`. Toto makro hledá bod, ve kterém se dvě cesty kříží. Jestliže existuje těchto bodů

více, vybere právě jeden; jestliže neexistuje žádný průsečík, makro vygeneruje chybovou hlášku. Příklad použití makra `intersectionpoint` je na obrázku 1.24, kde hledá body P, Q, R a S.

1.8.2 Parametrické operace s cestami

Makro `intersectionpoint` je založeno na základní operaci `intersectiontimes`. Je to jedna z mnoho operací, která pracuje s cestami parametricky. Nachází průsečíky dvou cest zavedením parametru „času“ pro každou cestu. To se vztahuje k parametrizačnímu schématu v kapitole 1.3, které popisuje cesty jako po částech kubické křivky $(X(t), Y(t))$, kde rozsah t je od nuly do počtu částí křivky. Jinými slovy: je-li cesta určena jako spojnice posloupností bodů, kde $t = 0$ v prvním bodě, pak $t = 1$ v dalším a $t = 2$ a dalším, atd. Výsledkem

```
a intersectiontimes b
```

je $(-1, -1)$ jestliže neexistuje žádný průsečík; jinak získáme dvojici (t_a, t_b), kde t_a je čas na cestě `a`, ve kterém protíná cestu `b` a t_b je odpovídající čas na cestě `b`.

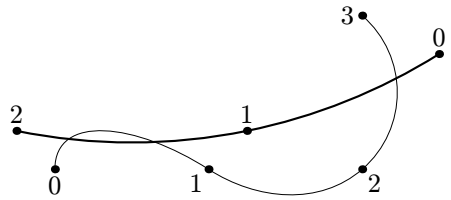
Například předpokládejme, že cesta `a` je vyznačena tenkou čarou na obrázku 1.25 a cesta `b` is vyznačena silnější čarou. Označují-li popisky časové okamžiky na cestách, dvojice časových hodnot spočtená příkazem

```
a intersectiontimes b
```

musí být jedna z

$$(0.25, 1.77),\ (0.75, 1.40) \text{nebo } (2.58, 0.24),$$

v závislosti na tom, který ze tří průsečíků byl METAPOSTovým překladačem vybrán. Přesná pravidla pro výběr mezi více průsečíky jsou poněkud složitá, ale v tomto příkladu se vyberou hodnoty času $(0.25, 1.77)$. Nižší časové okamžiky jsou upřednostňovány před vyššími tak, že (t_a, t_b) upřednostněno před (t'_a, t'_b) když $t'_a < t_a$ a $t_b < t'_b$. Jestliže žádná z možností neminimalizuje obě složky t_a a t_b, dostává složka t_a přednost, ale pravidla se komplikují, jestliže neexistuje žádné celé číslo mezi t_a a t'_a. (Více detailů viz *The METAFONTbook.*[Knu86a, Chapter 14])



Obrázek 1.25: Cesty s vyznačením časových hodnot.

Operátor `intersectiontimes` je pružnější než `intersectionpoint`, protože s časovými hodnotami na cestě lze dělat různé věci. Jednou z nejdůležitějších je právě zjištění „kde je cesta `p` v čase `t`?“. Vazba

`point` ⟨numeric expression⟩ `of` ⟨path primary⟩

odpovídá na tuto otázku, viz obrázek 1.26. Je-li ⟨numeric expression⟩ menší než nula nebo větší než hodnota přiřazená poslednímu bodu na cestě, vazba `point of` normálně poskytne koncový bod cesty. Proto je běžné užít předdefinovanou konstantu `infinity` (rovnu 4095.99998) jako ⟨numeric expression⟩ ve vazbě `point of` k označení konce cesty.

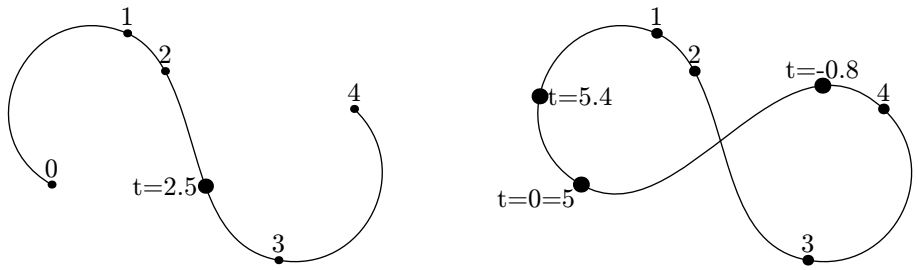
Takové „nekonečné“ časové hodnoty nefungují v cyklických cestách, protože časové okamžiky mimo normální rozsah mohou být v takovém případě ovládány modulární aritmetikou; tj. cyklická cesta `p` procházející body z_0, z_1, z_2, …, z_{n-1} má normální rozsah parametrů $0 \le t < n$, ale

`point t of p`

může být počítán pro každý t prvním krácením t modulo n. Není-li modul n běžně dostupný,

`length` ⟨path primary⟩

dává celočíselnou hodnotu horní meze normálního rozsahu parametru času pro danou cestu.



Obrázek 1.26: Použití příkazu `point of`.

METAPOST užívá stejný soulad mezi časovými hodnotami a body na cestě k ohodnocení operátoru `subpath`. Syntaxe pro tento operátor je

`subpath` ⟨pair expression⟩ `of` ⟨path primary⟩

Je-li hodnota ⟨pair expression⟩ rovna (t_1, t_2) a ⟨path primary⟩ je p, výsledkem je cesta shodná s cestou p z `point` t_1 do `point` t_2 . Je-li $t_2 < t_1$, `subpath` jde proti směru p. Použití operátoru `subpath` je na obrázku 1.27.

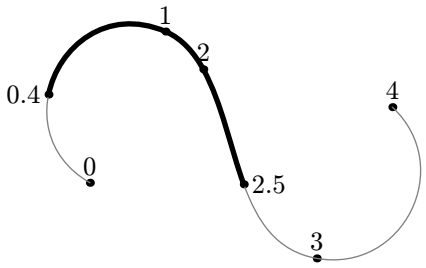
Důležitá operace založená na operátoru `subpath` je ⟨tertiary binop⟩ `cutbefore`. Pro křížící se cesty p_1 a p_2,

p_1 `cutbefore` p_2

je ekvivalentní s

`subpath (xpart(`p_1 `intersectiontimes` p_2`), length` p_1`) of` p_1

až na to, že také nastavuje proměnnou typu cesta `cuttings` pro část p_1, která byla oříznuta. Jinými slovy `cutbefore` vrací jeho první parametr s částí, kterou před průsečíkem uřízne. Jestliže je více průsečíků, snaží se oříznout co nejmenší část. Když se



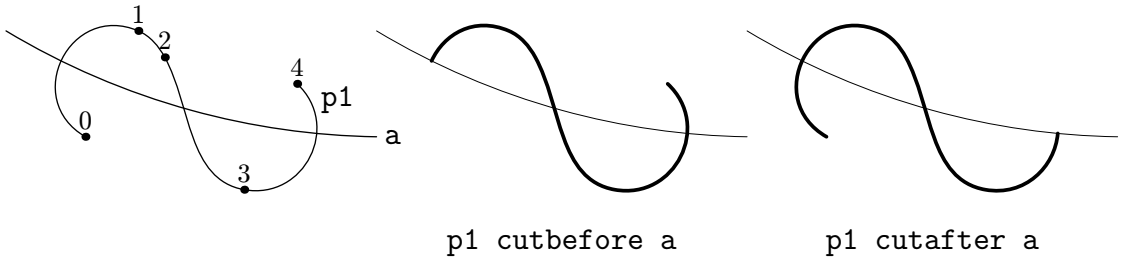
`subpath(0.4,2.5) of p1`

Obrázek 1.27: Použití příkazu `subpath of`.

cesty nekříží, `cutbefore` vrací její první argument. Existuje také podobný ⟨tertiary binop⟩ nazvaný `cutafter`, který používá `cutbefore` s časem obráceným podle svého prvního argumentu. Tedy

$$p_1 \texttt{ cutafter } p_2$$

se snaží oříznout p_1 za posledním průsečíkem s p_2. Ukázka chování operátorů `cutbefore` a `cutafter` je znázorněna na obrázku 1.28.



Obrázek 1.28: Použití příkazů `cutbefore of` a `cutafter of`.

Pro hladké navázání cest slouží operátor `softjoin` . Před použitím tohoto operátoru je třeba nastavit vnitřní proměnnou `join_radius` na odpovídající hodnotu poloměru zaoblení. Příklad použití operátoru `softjoin` je na obrázku 1.29.

Další operátor

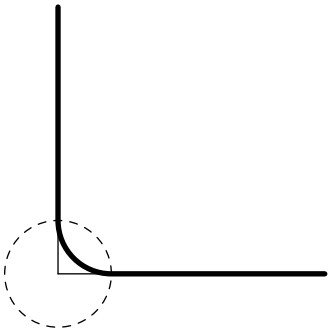
`direction` ⟨numeric expression⟩ `of` ⟨path primary⟩

hledá vektor ve směru ⟨path primary⟩. Je definovaný pro všechny časové okamžiky podobně jako vazba `point of`. Výsledný směrový vektor má správnou orientaci a libovolnou velikost, viz obrázek 1.30. Kombinací vazeb `point of` a `direction of` vznikne rovnice tečny, jak je znázorněno na obrázku 1.31.

Jestliže chcete nalézt bod na křivce, ve kterém má tečna požadovaný sklon, operátor `directiontime` převrátí operaci `direction of`. Zadání směrového vektoru a cesty

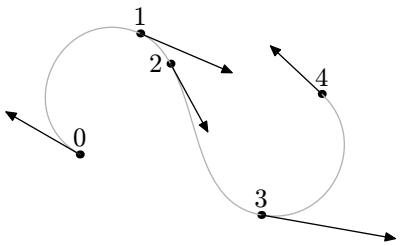
`directiontime` ⟨pair expression⟩ `of` ⟨path primary⟩

```
beginfig(29);
path p, q, r, c;
p=(100,0)--(0,0);
q=(0,0)--(0,100);
join_radius:=20;
r=p softjoin q;
c=fullcircle scaled 2join_radius;
draw p;
draw q;
draw c dashed evenly;
pickup pencircle scaled 2;
draw r;
endfig;
```



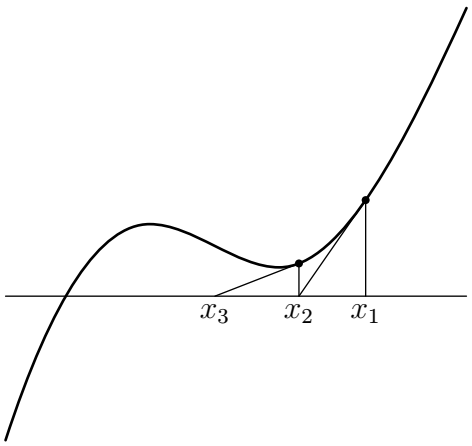
Obrázek 1.29: Použití příkazu `softjoin`.

```
beginfig(30);
path p[]; numeric u; pair t[];
u:=8mm;
z0=(0,0);
z1=(1u, 2u); z2=(1.5u,1.5u);
z3=(3u,-1u); z4=(4u,1u);
dotlabels.top(0,1,3,4);
dotlabels.lft(2);
p1=z0..z1..z2..z3..z4;
draw p1 withcolor .7 white;
for i=0 upto 4:
  t[i]:= direction i of p1;
  drawarrow ((0,0)--t[i]) shifted (z[i]);
endfor;
endfig;
```



Obrázek 1.30: Použití příkazu `direction of`.

```
beginfig(31);
numeric scf, #, t[];
3.2scf = 2.4in;
path fun;
# = .1;  % Keep the function single-valued
fun = ((0,-1#)..(1,.5#){right}..(1.9,.2#){right}..{curl .1}(3.2,2#))
  yscaled(1/#) scaled scf;
x1 = 2.5scf;
for i=1 upto 2:
  (t[i],whatever) =
    fun intersectiontimes ((x[i],-infinity)--(x[i],infinity));
  z[i] = point t[i] of fun;
  z[i]-(x[i+1],0) = whatever*direction t[i] of fun;
  draw (x[i],0)--z[i]--(x[i+1],0);
  fill fullcircle scaled 3bp shifted z[i];
endfor
label.bot(btex $x_1$ etex, (x1,0));
label.bot(btex $x_2$ etex, (x2,0));
label.bot(btex $x_3$ etex, (x3,0));
draw (0,0)--(3.2scf,0);
pickup pencircle scaled 1pt;
draw fun;
endfig;
```



Obrázek 1.31: Kombinace vazeb `point of` a `direction of`.

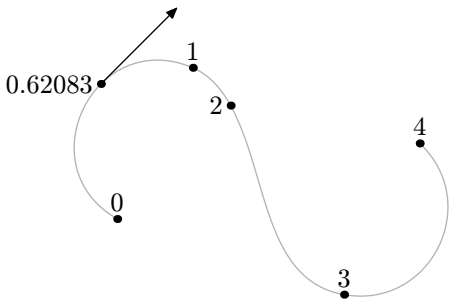
vrací číselnou hodnotu, která udává první čas t, ve kterém má cesta požadovaný směr. (Jestliže takový čas neexistuje, výsledkem je -1). Například, jestliže `p` je cesta na obrázku 1.32, `directiontime (1,1) of p` vrací 0.62083.

Existuje také předdefinované makro

`directionpoint` ⟨pair expression⟩ `of` ⟨path primary⟩

které hledá první bod cesty, ve kterém je dosaženo zadaného směru, viz obrázek 1.32. Jestliže se směr na cestě nevyskytuje makro `directionpoint` vypíše chybovou hlášku a vrátí souřadnice počátečního bodu cesty.

```
beginfig(32);
path p[]; numeric u, t; pair b;
u:=10mm;
z0=(0,0); z1=(1u,2u); z2=(1.5u,1.5u);
z3=(3u,-1u); z4=(4u,1u);
p1=z0..z1..z2..z3..z4;
draw p1 withcolor .7 white;
t=directiontime (1,1) of p1;
b=directionpoint (1,1) of p1;
dotlabels.top(0,1,3,4);
dotlabels.lft(2);
dotlabel.lft(decimal(t),b);
drawarrow b--(b+(1u,1u));
endfig;
```



Obrázek 1.32: Použití příkazů `directiontime` a `directionpoint`.

Operátory `arclength` a `arctime of` odpovídají „času“ na cestě, související s délkou cesty.[9] Výraz

`arclength` ⟨path primary⟩

udává délku cesty. Je-li `p` cesta a `a` je číslo mezi 0 a `arclength p`,

`arctime a of p`

udá čas `t` takový, že

`arclength subpath (0,t) of p` = `a`.

Operátor `turningnumber` lze aplikovat na uzavřenou cestu a vrací hodnotu `1`, pokud cesta probíhá proti směru hodinových ručiček, `-1`, pokud cesta probíhá po směru hodinových ručiček, a `0`, pokud nelze rozhodnout.

Příbuzný operátor `counterclockwise` se opět aplikuje na uzavřenou cestu a pokud je tato cesta ve směru hodinových ručiček, obrací její směr.

[9] Operátory `arclength` a `arctime` jsou pouze v METAPOSTu verze 0.50 a vyšší.

1.8.3 Afinní transformace

Všimněte si, že cesta `fun` na obrázku 1.31 je nejdříve vytvořena jako

```
(0,-.1)..(1,.05){right}..(1.9,.02){right}..{curl .1}(3.2,.2)
```

a pak operátory `yscaled` a `scaled` přizpůsobují tvar a velikost cesty. Jak název napovídá, výraz obsahující „`yscaled 10`" násobí y-ovou souřadnici deseti tak, že každý bod (x, y) v původní cestě odpovídá bodu $(x, 10y)$ v transformované cestě.

Existuje sedm transformačních operátorů včetně `scaled` a `yscaled`, jejichž vstupním argumentem je číslo nebo pár souřadnic:

$$\begin{aligned}
(x,y)\ \texttt{shifted}\ (a,b) &= (x+a,\ y+b);\\
(x,y)\ \texttt{rotated}\ \theta &= (x\cos\theta - y\sin\theta,\ x\sin\theta + y\cos\theta);\\
(x,y)\ \texttt{slanted}\ a &= (x+ay,\ y);\\
(x,y)\ \texttt{scaled}\ a &= (ax,\ ay);\\
(x,y)\ \texttt{xscaled}\ a &= (ax,\ y);\\
(x,y)\ \texttt{yscaled}\ a &= (x,\ ay);\\
(x,y)\ \texttt{zscaled}\ (a,b) &= (ax-by,\ bx+ay).
\end{aligned}$$

Chování většiny těchto operací je zřejmé až na `zscaled`, které může být chápáno jako násobení komplexním číslem. Výsledkem `zscaled` (a, b) je otočení a zvětšení tak, aby bylo $(1, 0)$ mapováno do (a, b). Výsledkem `rotated` θ je otáčení o θ stupňů proti směru hodinových ručiček.

Syntaxe pro transformační výrazy a transformační operátory je dána na schématu 1.5. Ta obsahuje dvě další možnosti pro ⟨transformer⟩:

$$\texttt{reflectededabout}(p, q)$$

osovou souměrnost podle přímky procházející body p a q; a

$$\texttt{rotatedaround}(p, \theta)$$

otáčení o úhel θ, který je zadaný ve stupních, kolem středu p proti směru hodinových ručiček.

Na obrázku 1.33 jsou shrnuty jednotlivé transformace.

Existuje také unární operátor `inverse`, který pracuje s transformací a hledá jinou transformaci, která ruší účinek původní transformace. Tedy jestliže

$$p = q\ \texttt{transformed}\ T$$

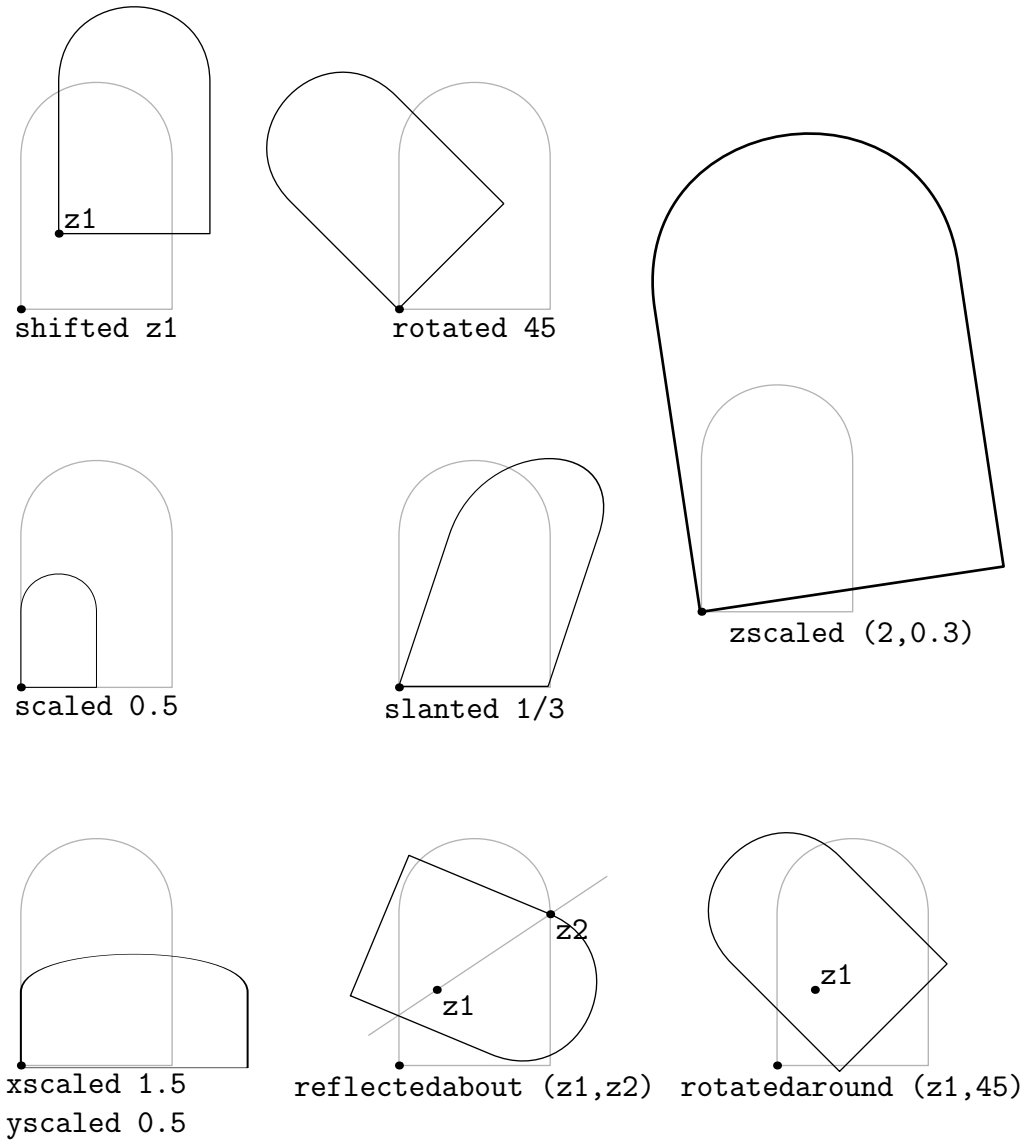
pak

$$q = p\ \texttt{transformed inverse}\ T.$$

Operátor `inverse` nemůže zpracovávat neznámé transformace , ale jak už jsme viděli, můžeme napsat

$$\texttt{T =}\ \langle\text{transform expression}\rangle$$

kde `T` není zatím zadaná hodnota.



Obrázek 1.33: Ukázka jednotlivých afinních transformací.

⟨pair secondary⟩ → ⟨pair secondary⟩⟨transformer⟩
⟨path secondary⟩ → ⟨path secondary⟩⟨transformer⟩
⟨picture secondary⟩ → ⟨picture secondary⟩⟨transformer⟩
⟨pen secondary⟩ → ⟨pen secondary⟩⟨transformer⟩
⟨transform secondary⟩ → ⟨transform secondary⟩⟨transformer⟩

⟨transformer⟩ → `rotated`⟨numeric primary⟩
| `scaled`⟨numeric primary⟩
| `shifted`⟨pair primary⟩
| `slanted`⟨numeric primary⟩
| `transformed`⟨transform primary⟩
| `xscaled`⟨numeric primary⟩
| `yscaled`⟨numeric primary⟩
| `zscaled`⟨pair primary⟩
| `reflectedabout(`⟨pair expression⟩`,`⟨pair expression⟩`)`
| `rotatedaround(`⟨pair expression⟩`,`⟨numeric expression⟩`)`

Schéma 1.5: Syntaxe pro transformační a příbuzné operátory.

Pokud označíme bod (x, y) 3-složkovým vektorem $(xy1)^T$, každá z výše uvedených transformací může být popsána afinní maticí. Například rotaci o úhel θ okolo počátku proti směru hodinových ručiček a posunutí o (a, b) lze vyjádřit následujícími maticemi:

$$\texttt{rotated}\theta = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad \texttt{shifted}(a,b) = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix}.$$

Snadno se ověří, že

$$\texttt{rotatedaround}((a,b),\theta) = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix}.$$

Afinní matice pro `zscaled` (a,b) je:

$$\texttt{zscaled}(a,b) = \begin{pmatrix} a & -b & 0 \\ b & a & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Obecný tvar afinní matice `T` je

$$\texttt{T} = \begin{pmatrix} T_{xx} & T_{xy} & T_x \\ T_{yx} & T_{yy} & T_y \\ 0 & 0 & 1 \end{pmatrix}.$$

Odpovídající transformace ve 2D je

$$(x, y) \leftrightarrow (T_{xx}x + T_{xy}y + T_x+,\ T_{yx}x + T_{yy}y + T_y).$$

Toto zobrazení je zcela určeno šesticí $(T_x, T_y, T_{xx}, T_{xy}, T_{yx}, T_{yy})$. Informace o zobrazení může být uložena do proměnné typu `transform` a následně použita v příkazu `transformed`.

Existují tři způsoby, jak definovat transformaci:

- *Pomocí základních transformací.* Například

  ```
  transform T;
  T = identity shifted (-1,0) rotated 60 shifted (1,0);
  ```

  definuje transformaci `T` jako složení posunutí o $(-1, 0)$, rotaci okolo počátku o $60°$ a posunutí o vektor $(1, 0)$. `identity` je jednotková matice.

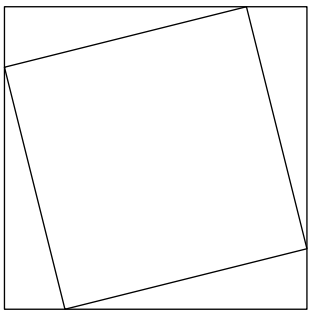
- *Zadáním prvků transformační matice.* Šestice parametrů $(T_x, T_y, T_{xx}, T_{xy}, T_{yx}, T_{yy})$, které definují transformaci `T` může být zadána přímo jako `xpart T`, `ypart T`, `xxpart T`, `xypart T`, `yxpart T` a `yypart T`. Například

  ```
  transform T;
   xpart T =  ypart T =  1;
  xxpart T = yypart T =  0;
  xypart T = yxpart T = -1;
  ```

  definuje osovou souměrnost s osou danou body $(1, 0)$ a $(0, 1)$.

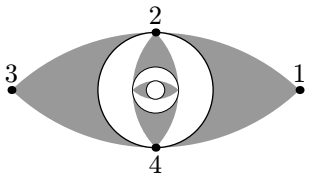
- *Zadáním obrazů třech bodů.* Neznámou transformaci je možno aplikovat na známý bod a použít v lineární rovnici. Například:

  ```
  beginfig(1)
  pair A,B,C,D;
  u:=4cm;
  A=(0,0); B=(u,0); C=(u,u); D=(0,u);

  transform T;
  A transformed T = 1/5[A,B];
  B transformed T = 1/5[B,C];
  C transformed T = 1/5[C,D];

  path p;
  p = A--B--C--D--cycle;
  draw p;
  draw p transformed T;
  endfig;
  ```



Na obrázku 1.34 je demonstrována možnost kombinace výše uvedených způsobů zadání transformační matice.

```
beginfig(34);
path p[];
p1 = fullcircle scaled .6in;
z1=(.75in,0)=-z3;
z2=directionpoint left of p1=-z4;
p2 = z1..z2..{curl1}z3..z4..{curl 1}cycle;
fill p2 withcolor .4[white,black];
unfill p1;
draw p1;
transform T;
z1 transformed T = z2;
z3 transformed T = z4;
xxpart T=yypart T;  yxpart T=-xypart T;
picture pic;
pic = currentpicture;
for i=1 upto 2:
  pic:=pic transformed T;
  draw pic;
endfor
dotlabels.top(1,2,3); dotlabels.bot(4);
endfig;
```



Obrázek 1.34: *Fraktální* obrázek získaný afinními transformacemi.

1.8.4 Přerušované čáry

METAPOSTový jazyk poskytuje mnoho způsobů jak měnit vzhled čar kromě změny jejich tloušťky. Jeden způsob je použít přerušované čáry (viz obrázek 1.6). Syntaxe je

`draw` ⟨path expression⟩ `dashed` ⟨dash pattern⟩

kde ⟨dash pattern⟩ je vlastně speciální typ ⟨picture expression⟩. Existuje předdefinovaný ⟨dash pattern⟩ nazvaný `evenly`, který dělá pomlčky dlouhé 3 PostScriptové body oddělené stejně dlouhými mezerami. Další předdefinovaný vzor `withdots` vytváří tečkovanou čáru s tečkami vzdálenými 5 PostScriptových bodů.[10] Pro tečky nebo pomlčky více vzdálené ⟨dash pattern⟩ může měnit měřítko jak je ukázáno na obrázku 1.35.

```
. . . . . dashed withdots scaled 2
. . . . . dashed withdots
— — — — — — dashed evenly scaled 4
– – – – – – – – – – – – dashed evenly scaled 2
------------------------ dashed evenly
```

Obrázek 1.35: Přerušované čáry a odpovídající vzory.

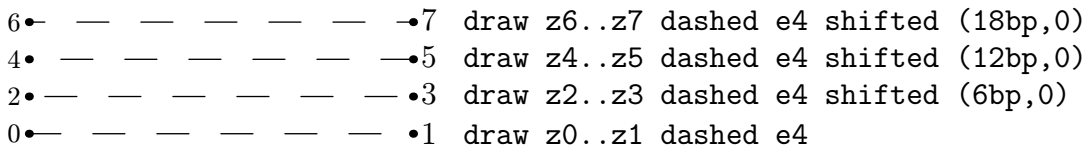
Další způsob jak změnit vzor čárkování, je změnit její periodu posunutím v horizontálním směru. Posunutí doprava způsobí, že se čárky posunou dopředu podél cesty a

[10] `withdots` se vyskytuje pouze v METAPOSTu verze 0.50 a vyšší.

⟨dash pattern⟩ → `dashpattern(`⟨on/off list⟩`)`
⟨on/off list⟩ → ⟨on/off list⟩⟨on/off clause⟩ | ⟨on/off clause⟩
⟨on/off clause⟩ → `on`⟨numeric tertiary⟩ | `off`⟨numeric tertiary⟩

Schéma 1.6: Syntaxe pro funkci `dashpattern`

posunutí doleva je posune proti směru cesty. Obrázek 1.36 zobrazuje výsledek. Vzor čárkování může být chápán jako nekonečně se opakující vzor natažený podél vodorovné čáry, kde je vyčárkována část ležící vpravo od $y$ové osy.



Obrázek 1.36: Další přerušované čáry, `e4` odpovídá vzoru `evenly scaled 4`.

Jestliže posunete vzor čárkování tak, že y-ová osa kříží střed čárky, pak první čárka bude zkrácena. Tedy čára se vzorem čárkování `e4` začíná s čárkou dlouhou 12bp, následovanou 12bp dlouhou mezerou a další 12bp dlouhou čárkou, atd., zatímco `e4 shifted (-6bp,0)` vytvoří čárku 6bp dlouhou, mezeru 12bp, pak čárku 12bp, atd. Tento vzor může být určen přímo pomocí funkce `dashpattern` :

```
dashpattern(on 6bp off 12bp on 6bp)
```

To znamená „kresli prvních 6bp čáru, přeskoč následujících 12bp, pak nakresli dalších 6bp a to opakuj.“ Jestliže čára, která má být vykreslena, je delší než 30bp, posledních 6bp prvního otisku vzoru čárkování bude spojena s prvními 6bp následujícího otisku a vytvoří čárku 12bp dlouhou. Obecná syntaxe pro funkci `dashpattern` je ukázána na schématu 1.6.

Protože `dash pattern` je speciální druh proměnné typu `picture`, funkce `dashpattern` vrací `picture`. Není nezbytně nutné znát strukturu takového obrázku, tak běžný čtenář pravděpodobně přeskočí na kapitolu 1.8.5. Pro ty, kteří ji chtějí znát, malé experimentování ukáže, že pokud `d` je

```
dashpattern(on 6bp off 12bp on 6bp),
```

potom `llcorner d` je $(0, 24)$ a `urcorner d` je $(24, 24)$. Nakreslení `d` přímo bez použití tohoto jako vzoru čárkování vykreslí dvě tenké vodorovné čárky:

Čáry v tomto příkladu jsou určeny tak, že mají nulovou tloušťku. To nevadí, neboť tloušťka čáry je ignorována, pokud je obrázek použit jako vzor čárkování.

Obecné pravilo vyhodnocení obrázku `d` jako vzoru čárkování je, že úseky čar v `d` jsou promítnuty do x-ové osy a výsledný vzor je opakován do nekonečna v obou směrech

spojením konců jednotlivých kopií vzorů. Skutečné délky čárek jsou získány umístěním počátku do $x = 0$ a prohlížením kladným x-ovým směrem.

K lepší představě „opakování do nekonečna“, nechť $P(\mathtt{d})$ je projekcí `d` do x-ové osy a nechť $\mathrm{shift}(P(\mathtt{d}), x)$ je výsledek posunutí `d` o x. Vzor vyplývající z nekonečného opakování je

$$\bigcup_{\text{integers } n} \mathrm{shift}(P(d),\, n \cdot \ell(d)),$$

kde $\ell(d)$ měří délku $P(d)$. Nejpřijatelnější definice této délky je $d_{\max} - d_{\min}$, kde $[d_{\min}, d_{\max}]$ je rozsah x-ových souřadnic v $P(d)$. Ve skutečnosti METAPOST užívá

$$\max(|y_0(\mathtt{d})|\,,\, d_{\max} - d_{\min}),$$

kde $y_0(\mathtt{d})$ je y-ová souřadnice obsahu `d`. Obsah `d` může ležet na vodorovné čáře, ale jestliže neleží, METAPOSTový překladač vybere y-ovou souřadnici, která se vyskytuje v `d`.

Obrázek užitý jako vzor čárkování nesmí obsahovat žádný text ani vyplněné oblasti, ale smí obsahovat čáru, která je sama o sobě přerušovaná. To umožní dělat malé čárky uvnitř větších čar jak je zobrazeno na obrázku 1.37

```
beginfig(37);
draw dashpattern(on 15bp off 15bp) dashed evenly;
picture p;
p=currentpicture;
currentpicture:=nullpicture;
draw fullcircle scaled 1cm xscaled 3 dashed p;
endfig;
```

Obrázek 1.37: Složitý vzor čárkování.

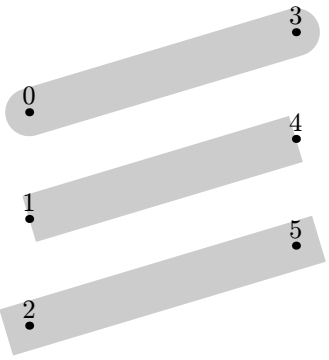
1.8.5 Další možnosti

Mohli jste si všimnout, že přerušované čáry vytvořené pomocí `dashed evenly` se jeví více černé než bílé. To je vlivem parametru `linecap`, který určuje vzhled konců čar stejně jako konců čárek. Existuje několik dalších způsobů, jak ovlivnit vzhled objektů kreslených METAPOSTem.

Parametr `linecap` má tři různá nastavení stejně jako PostScript. Plain METAPOST dává této vnitřní proměnné implicitní hodnotu `rounded`, která způsobuje, že čárky budou nakresleny se zakulacenými konci, jako čára z `z0` do `z3` na obrázku 1.38. Nastavení `linecap := butt` uřízne konce a zarovná je tak, že čárky vytvořené pomocí `dashed evenly` mají délku 3bp, místo 3bp plus tloušťka čáry. Nastavením `linecap := squared` získáte také čtvercové konce, jako je to na čáře z `z2` do `z5` na obrázku 1.38.

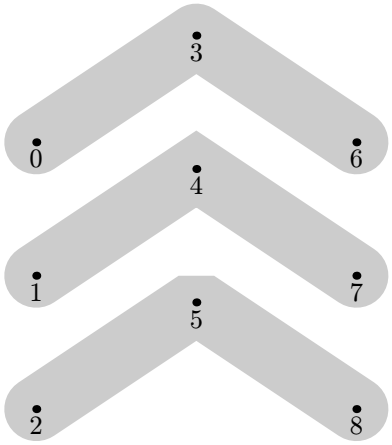
Další parametr převzatý z PostScriptu ovlivňuje způsob, jakým příkaz `draw` pracuje s ostrými rohy v kreslené cestě. Parametr `linejoin` může být `rounded`, `beveled` nebo

```
beginfig(38);
for i=0 upto 2:
  z[i]=(0,40i); z[i+3]-z[i]=(100,30);
endfor
pickup pencircle scaled 18;
draw z0..z3 withcolor .8white;
linecap:=butt;
draw z1..z4 withcolor .8white;
linecap:=squared;
draw z2..z5 withcolor .8white;
dotlabels.top(0,1,2,3,4,5);
endfig; linecap:=rounded;
```



Obrázek 1.38: Vliv parametru `linecap` na konce čar.

```
beginfig(39);
for i=0 upto 2:
  z[i]=(0,50i); z[i+3]-z[i]=(60,40);
  z[i+6]-z[i]=(120,0);
endfor
pickup pencircle scaled 24;
draw z0--z3--z6 withcolor .8white;
linejoin:=mitered;
draw z1..z4--z7 withcolor .8white;
linejoin:=beveled;
draw z2..z5--z8 withcolor .8white;
dotlabels.bot(0,1,2,3,4,5,6,7,8);
endfig; linejoin:=rounded;
```



Obrázek 1.39: Vliv parametru `linejoin` na vykreslování rohů cest.

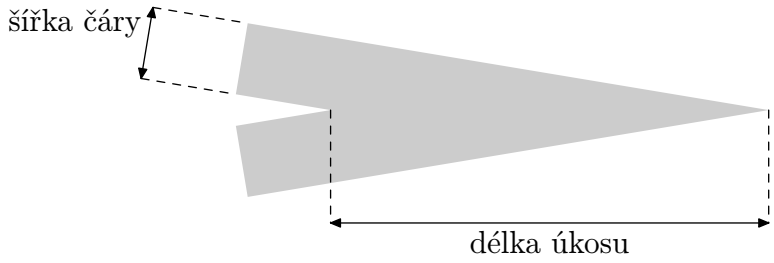
`mitered` jak je ukázáno na obrázku 1.39. Implicitní hodnota pro Plain METAPOST je `rounded`, která představuje kreslení kruhovým štětcem.

Jestliže `linejoin` je `mitered`, ostré úhly generují dlouhé ostré tahy, jak je ukázáno na obrázku 1.40. Jelikož toto může být nepřijatelné, existuje vnitřní proměnná nazvaná `miterlimit`, která určuje, jaká maximální situace může nastat, než se spoj `mitered` změní na `beveled`. Pro Plain METAPOST, `miterlimit` má implicitní hodnotu 10.0 a čára se zkosí, jestliže poměr délky úkosu k šířce čáry dosáhne této hodnoty.

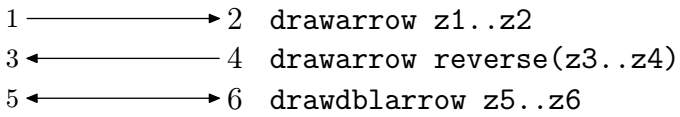
Parametry `linecap`, `linejoin`, a `miterlimit` jsou důležité, protože mají vliv na to, co bude nakresleno v pozadí. Například Plain METAPOST má příkazy pro kreslení šipek, hroty šipek jsou mírně zakulaceny jestliže `linejoin` je `rounded`. Výsledek závisí na šířce čáry a je jemný na implicitní šířce čáry 0.5bp, jak je vidět na obrázku 1.41.

Kreslení šipek jako na obrázku 1.41 je důvodem k zadání

`drawarrow` ⟨path expression⟩



Obrázek 1.40: `miterlimit` určuje poměr délky úkosu k šířce čáry.



Obrázek 1.41: Tři způsoby kreslení šipek.

místo `draw` ⟨path expression⟩. Tyto tahy jsou dány cestou s hrotem šipky v posledním bodě cesty. Jestliže chcete hrot šipky na začátku cesty, použijte operátor `reverse`, který z původní cesty vytváří novou s opačným časem; tj. pro cestu `p` s `length p` $= n$,

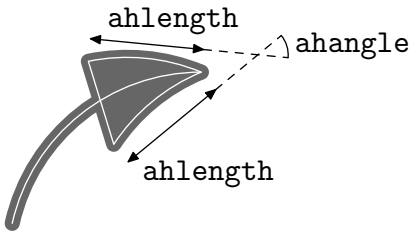
`point` t `of reverse p` and `point` $n - t$ `of p`

znamenají totéž.

Jak je ukázáno na obrázku 1.41, příkaz začínající

`drawdblarrow` ⟨path expression⟩

kreslí šipku s dvěma hroty. Je zaručeno, že velikost hrotu je větší než šířka čáry, ale měla by být rozšířena, jestliže je šířka čáry příliš veliká. To lze udělat přiřazením nové hodnoty vnitřní proměnné `ahlength`, která určuje délku hrotu špičky jak je ukázáno na obrázku 1.42. Zvýšení `ahlength` z implicitní hodnoty 4 PostScriptové body na 1.5 centimetrů vytváří velký hrot šipky na obrázku 1.42. Existuje také parametr `ahangle`, který upravuje úhel a špičku hrotu šipky. Implicitní hodnota úhlu je 45°, jak je zobrazeno na obrázku.



Obrázek 1.42: Hrot šipky s označením klíčových parametrů,
cesta užitá k vykreslení je vyznačena bílou barvou.

Hrot šipky je tvořen vyplněnou trojúhelníkovou oblastí, která je bíle vyznačena na obrázku 1.42 a pak obtažena právě drženým perem. Tato kombinace vyplnění a kreslení

může být spojena do jediného příkazu `filldraw`:

`filldraw` ⟨path expression⟩ ⟨optional `dashed` and `withcolor` and `withpen` clauses⟩;

⟨path expression⟩ může být cyklicky uzavřena jako trojúhelníková cesta na obrázku 1.42. Tato cesta by se neměla plést s cestou - argumentem příkazu `drawarrow`, která je vyznačena na obrázku bíle.

Bílé čáry, jako je tato na obrázku, mohou být vytvořeny příkazem `undraw`. Toto je mazání verze `draw`, která kreslí `withcolor background` stejně jako to dělá příkaz `unfill`. Existuje také příkaz `unfilldraw`.

Příkazy `filldraw`, `undraw` a `unfilldraw` a všechny příkazy pro kreslení šipek jako jsou `fill` a `draw` využívají možnosti nastavení `dashed`, `withpen`, a `withcolor`. Jestliže máte mnoho kreslicích příkazů, je dobré mít možnost aplikovat nastavení jako `withcolor 0.8white` všem bez toho abychom to psaly opakovaně jako na obrázcích 1.38 a 1.39. Příkaz pro tento účel je

`drawoptions(`⟨text⟩`)`

kde parametr ⟨text⟩ zadává posloupnost voleb `dashed`, `withcolor` a `withpen`, které budou automaticky aplikovány pro všechny příkazy kreslení. Jestliže určíte

`drawoptions(withcolor .5[black,white])`

a pak chcete nakreslit černou čáru, můžete zrušit `drawoptions` specifikací

`draw` ⟨path expression⟩ `withcolor black`

K vypnutí všech `drawoptions` dohromady, zadejte prázdný seznam:

`drawoptions()`

(Toto se děje automaticky, když je spuštěno makro `beginfig`.)

Jelikož jsou nepodstatné možnosti ignorovány, nemůže uškodit, jestliže zadáte příkaz

`drawoptions(dashed evenly)`

následovaný posloupností příkazů `draw` a `fill`. Nedává smysl, jestliže použijete přerušovanou čáru při vyplňování. Tedy při použití příkazu `fill` je `dashed evenly` ignorováno. Ukazuje se, že

`drawoptions(withpen` ⟨pen expression⟩`)`

má vliv na příkazy `fill` stejně jako `draw`. Existuje proměnná `currentpen` taková, že `fill ... withpen currentpen` je ekvivalentní příkazu `filldraw`.

Co přesně znamená, že možnosti kreslení ovlivní ty příkazy, kde dávají smysl? Možnost `dashed` ⟨dash pattern⟩ ovlivní pouze příkazy

`draw` ⟨path expression⟩

a text, který se vyskytuje v ⟨picture expression⟩ jako parametr příkazu

`draw` ⟨picture expression⟩

je ovlivněn pouze volbou `withcolor` ⟨color expression⟩. Pro všechny další kombinace příkazů kreslení a možností existuje nějaký účinek. Možnost aplikovaná na příkaz `draw` ⟨picture expression⟩ bude obvykle ovlivňovat některou část obrázku, ale ne ostatní. Například volby `dashed` nebo `withpen` budou mít vliv na všechny čáry v obrázku, ale ne na texty.

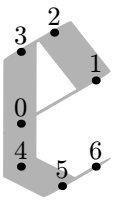
1.8.6 Pera

V předešlé kapitole bylo uvedeno množství příkladů `pickup` ⟨pen expression⟩ a `withpen` ⟨pen expression⟩, ale nebyly tam žádné příklady „`pen`" výrazů kromě

`pencircle scaled` ⟨numeric primary⟩

které vytváří čáru předepsané šířky. Pro kaligrafické efekty, jaké jsou na obrázku 1.43, můžete použít některý z transformačních operátorů probraných v kapitole 1.8.3. Výchozí bod takových transformací je `pencircle`, kruh o průměru jeden PostScriptový bod. Tedy afinní transformace vytvářejí kruhový nebo eliptický tvar pera. Šířka čáry kreslená perem závisí na tom, jaký úhel svírá s delší osou elipsy.

```
beginfig(43);
pickup pencircle scaled .2in yscaled .08 rotated 30;
x0=x3=x4;
z1-z0 = .45in*dir 30;
z2-z3 = whatever*(z1-z0);
z6-z5 = whatever*(z1-z0);
z1-z6 = 1.2*(z3-z0);
rt x3 = lft x2;
x5 = .55[x4,x6];
y4 = y6;
lft x3 = bot y5 = 0;
top y2 = .9in;
draw z0--z1--z2--z3--z4--z5--z6 withcolor .7white;
dotlabels.top(0,1,2,3,4,5,6);
endfig;
```



Obrázek 1.43: Kaligrafický obrázek.

Obrázek 1.43 znázorňuje operátory `lft`, `rt`, `top` a `bot`, které odpovídají na otázku „Je-li aktuální pero umístěno na pozici zadané parametrem, kde bude levá, pravá, horní a dolní hrana?" V tomto případě je aktuální pero elipsa zadaná příkazem `pickup` a její `bounding box` je 0.1734 palce široká a 0.1010 palce vysoká, tak `rt x3` je `x3`$+$`0.0867in` a `bot y5` je `y5` $-$ `0.0505in`. Operátory `lft`, `rt`, `top` a `bot` také připouští parametry typu `pair`. V tom případě se x-ové a y-ové souřadnice vypočítávají z bodu ve tvaru pera umístěného nejvíc vlevo, vpravo, nahoře nebo dole. Například

$$\texttt{rt}(x,y) = (x,y) + (\texttt{0.0867in}, \texttt{0.0496in})$$

pro pero na obrázku 1.43. Povšimněte si, že `beginfig` znovu nastaví aktuální pero na implicitní hodnotu

`pencircle scaled 0.5bp`

na začátku každého obrázku. Tato hodnota může být kdykoliv znovu zvolena zadáním příkazu `pickup defaultpen`.

Vzhledem ke kompatibilitě s METAFONTem, METAPOST také připouští polygonální tvar pera. Existují předdefinované tvary per: `pensquare`, `penrazor` a `penspeck`. Dokonce existuje operátor nazvaný `makepen`, který z cesty ve tvaru konvexního mnohoúhelníku dělá pero takového tvaru a velikosti. Jestliže cesta není zcela konvexní nebo

polygonální operátor `makepen` narovná hrany a/nebo obere některé vrcholy (viz obrázek 1.44). Například `pensquare` je ekvivalentní s

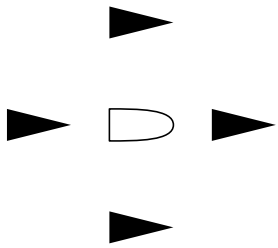
```
makepen((-.5,-.5)--(.5,-.5)--(.5,.5)--(-.5,.5)--cycle)
```

Inverzní k `makepen` je operátor `makepath`, který zpracovává ⟨pen primary⟩ a vrací odpovídající cestu. `makepath pencircle` vytváří kruhovou cestu identickou s `fullcircle`. Toto pracuje také pro polygonální pera tak, že

`makepath makepen` ⟨path expression⟩

bere libovolnou uzavřenou cestu a změní ji na konvexní mnohoúhelník.

```
beginfig(44)
path p;
pen brko;
z1=(16,16); z2=(0,0);
z3=(16,-16); z4=(32,0);
p=(0,0)--(0,10){right}...(20,5)...{left}cycle;
brko=makepen p;
draw p shifted (32,0);
for i=1 upto 4:
   draw z[i] scaled 2 withpen brko;
endfor;
endfig;
```



Obrázek 1.44: Použití příkazu `makepen`.

METAPOST poskytuje i příkazy pro vykreslování cest s proměnnou tloušťkou a nakloněním pera. Jedná se o příkazy `penpos` a `penstroke`. Příklad použití je uveden na obrázku 1.45. Podrobný popis syntaxe je uveden v [Knu86a].

1.8.7 Speciální příkazy pro vytváření cest

Někdy je potřeba vykreslit značně komplikované křivky. Pro tyto účely disponuje METAPOST několika funkcemi. Jednou z nich je `flex`. Konstrukce $flex(z_1, z_2, z_3)$ vytvoří cestu $z_1..z_2\{z_3-z_1\}..z_3$ a podobně $flex(z_1, z_2, z_3, z_4)$ cestu $\{z_1..z_2\{z_4-z_1\}..z_3\{z_4-z_1\}..z_4\}$; obecně

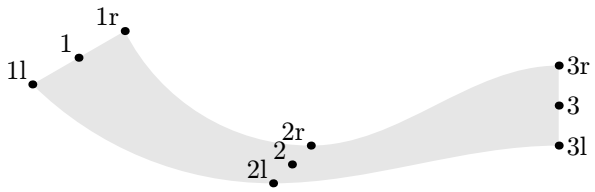
$$flex(z_1, z_2, ..., z_{n-1}, z_n)$$

je zkratkou pro cestu

$$z_1..z_2\{z_n - z_1\}..\cdots..z_{n-1}\{z_n - z_1\}..z_n.$$

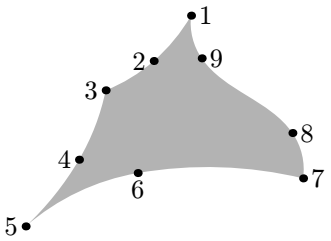
Ze dvou koncových bodů, z_1 a z_n, a jednoho nebo více mezilehlých bodů funkce `flex` určí cestu, u které se z mezilehlých bodů vychází ve směru daného koncovými body. Použití příkazu je demonstrováno na obrázku 1.46.

```
beginfig(45);
z1 = (0,40); z2 = (80,0); x3 = 180; y3l=y2r;
penpos1(40,30); penpos2(20,45); penpos3(30,90);
drawoptions(withcolor 0.9white);
penstroke z1e..z2e{right}..{right}z3e;
drawoptions();
dotlabels.ulft(1,1r,1l);
dotlabels.ulft(2,2r,2l);
dotlabels.rt(3,3r,3l);
endfig;
```



Obrázek 1.45: Použití příkazů `penpos` a `penstroke`.

```
beginfig(46);
z1=(0,39); z2=(-14,22); z3=(-32,11);
z4=(-42,-15); z5=(-62,-40); z6=(-20,-20);
z7=(42,-22); z8=(38,-5); z9=(4,23);
fill flex(z1,z2,z3) & flex(z3,z4,z5)
     & flex(z5,z6,z7) & flex(z7,z8,z9,z1)
     & cycle withcolor 0.7white;
dotlabels.lft(2,3,4,5);
dotlabels.bot(6);
dotlabels.rt(1,7,8,9);
endfig;
```

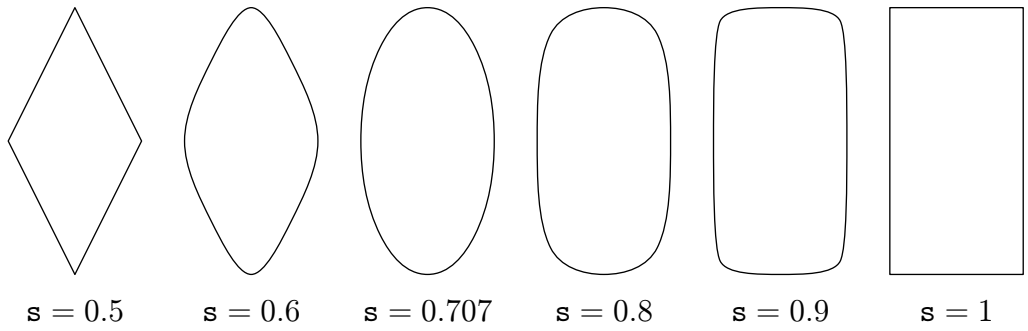


Obrázek 1.46: Použití operátoru `flex`.

Další užitečnou funkcí je `superellipse`, která slouží k vykreslování oválů. Syntaxe této funkce je

```
superellipse(r,t,l,b,s)
```

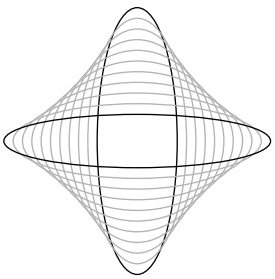
kde argumenty postupně znamenají: pravý, horní, levý a dolní bod a `s` označuje parametr, který řídí tvar oválu. `s` by mělo být mezi 0.5 (diamant) a 1.0 (obdélník); preferovány jsou hodnoty z okolí 0.75. Normální elipse odpovídá hodnota parametru `s`$= \sqrt{2}/2 \approx 0.707$. Na obrázku 1.47 jsou zobrazeny ovály pro různé hodnoty parametru `s`.



Obrázek 1.47: Výsledky operátoru `superellipse` v závislosti na parametru `s` pro hodnoty `r=(25,0), t=(0,50), l=(-25,0), b=(0,-50)`.

Schopnost získat klíčové a řídící body cesty umožňuje definovat zajímavé operace, jako např. funkci `interpath`, která umožňuje interpolovat dvě cesty (morfing). Například, `interpath(1/3,p,q)` pro cesty `p` a `q` délky n vytvoří cestu délky n, jejíž body jsou `1/3[point t of p, point t of q]`, kde $0 \leq t \leq n$. Ukázka *morfingu* je na obrázku 1.48.

```
beginfig(48);
path p, q;
p=fullcircle xscaled 0.3 scaled 100;
q=fullcircle yscaled 0.2 scaled 100;
draw p;
for n=1 upto 9:
  draw interpath(n/10, p, q) withcolor 0.7white;
endfor
draw q;
endfig;
```



Obrázek 1.48: Použití operátoru `interpath`.

1.8.8 Ořezávání a nízkoúrovňové příkazy pro kreslení

Příkazy takové jako `draw`, `fill`, `filldraw` a `unfill` jsou součástí balíku maker Plain a jsou definovány pomocí jednodušších příkazů. Hlavní rozdíl mezi příkazy pro kreslení,

⟨addto command⟩ →
`addto`⟨picture variable⟩`also`⟨picture expression⟩⟨option list⟩
| `addto`⟨picture variable⟩`contour`⟨path expression⟩⟨option list⟩
| `addto`⟨picture variable⟩`doublepath`⟨path expression⟩⟨option list⟩
⟨option list⟩ → ⟨empty⟩ | ⟨drawing option⟩⟨option list⟩
⟨drawing option⟩ → `withcolor`⟨color expression⟩
| `withpen`⟨pen expression⟩ | `dashed`⟨picture expression⟩

Schéma 1.7: Syntaxe pro jednoduché kreslící příkazy.

které byly probrány v předešlé kapitole a jednodušší verzí je, že všechny jednoduché příkazy vyžadují, abyste určily proměnnou typu `picture` k uchování výsledku. Pro `fill`, `draw` a příbuzné příkazy výsledky se vždy uchovávají do proměnné `currentpicture`. Syntaxe pro jednoduché kreslící příkazy, které vám dovolují specifikovat proměnnou typu `picture` je znázorněna na schématu 1.7.

Syntaxe pro jednoduché kreslící příkazy je kompatibilní s METAFONTem. Tabulka 1.3 ukazuje, jak jednoduché kreslící příkazy souvisí s příkazy `draw` a `fill`. Každý z příkazů v prvním sloupci tabulky může být zakončen svým vlastním ⟨option list⟩, což je ekvivalentní s připojením ⟨option list⟩ k odpovídajícímu zápisu v druhém sloupci tabulky. Například

`draw` *p* `withpen pencircle`

je ekvivalentní s

`addto currentpicture doublepath` *p* `withpen currentpen withpen pencircle`

kde `currentpen` je zvláštní proměnná typu `pen`, která si vždy pamatuje poslední použité pero. Druhá podmínka `withpen` tiše potlačí `withpen currentpen` z rozšíření `draw`.

příkaz	odpovídající základní příkazy
`draw` *pic*	`addto currentpicture also` *pic*
`draw` *p*	`addto currentpicture doublepath` *p* `withpen` *q*
`fill` *c*	`addto currentpicture contour` *c*
`filldraw` *c*	`addto currentpicture contour` *c* `withpen` *q*
`undraw` *pic*	`addto currentpicture also` *pic* `withcolor` *b*
`undraw` *p*	`addto currentpicture doublepath` *p* `withpen` *q* `withcolor` *b*
`unfill` *c*	`addto currentpicture contour` *c* `withcolor` *b*
`unfilldraw` *c*	`addto currentpicture contour` *c* `withpen` *q* `withcolor` *b*

Tabulka 1.3: Běžné příkazy pro kreslení a odpovídající základní verze, kde *q* je `currentpen`, *b* je `background`, *p* je libovolná cesta, *c* je cyklická cesta a *pic* je ⟨picture expression⟩. Všimněte si, že neprázdný `drawoptions` by komplikoval vstupy ve druhém sloupci.

Od verze METAPOSTu 0.6 lze využívat i nové makro

`image`(⟨drawing commands⟩),

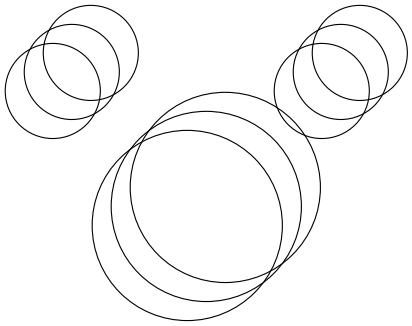
které vrací obrázek vytvořený posloupností kreslicích příkazů.

Následující příklad demonstruje použití nízkoúrovňových příkazů, makra `image` a proměnné `currentpicture`. Postupně jsou uvedeny tři zdrojové kódy, jejichž výsledkem je vždy obr. 1.49.

```
beginfig(49)
numeric u; u:=1.5cm;
picture obr;
obr:=nullpicture;
addto obr doublepath fullcircle scaled 2u;
addto obr doublepath fullcircle scaled 1u shifted (2u*dir45);
addto obr doublepath fullcircle scaled 1u shifted (2u*dir135);
draw obr withpen currentpen;
draw obr shifted (.2u,.2u) withpen currentpen;
draw obr shifted (.4u,.4u) withpen currentpen;
endfig;
```

```
beginfig(49)
numeric u; u:=1.5cm;
picture obr;
obr = image(
        draw fullcircle scaled 2u;
        draw fullcircle scaled 1u shifted (2u*dir45);
        draw fullcircle scaled 1u shifted (2u*dir135););
draw obr;
draw obr shifted (.2u,.2u);
draw obr shifted (.4u,.4u);
endfig;
```

```
beginfig(49)
numeric u; u:=1.5cm;
picture obr;
draw fullcircle scaled 2u;
draw fullcircle scaled 1u shifted (2u*dir45);
draw fullcircle scaled 1u shifted (2u*dir135);
obr := currentpicture;
currentpicture := nullpicture;
draw obr;
draw obr shifted (.2u,.2u);
draw obr shifted (.4u,.4u);
endfig;
```



Obrázek 1.49: Použití nízkoúrovňových příkazů, `currentpicture` nebo `image`.

Dva další jednoduché příkazy kreslení nepřijímají žádné podmínky. Jeden příkaz je `setbounds`, který byl probrán v kapitole 1.7.3; další příkaz je `clip`:

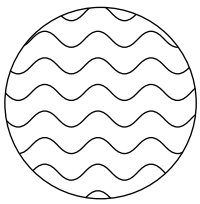
`clip` ⟨picture variable⟩ `to` ⟨path expression⟩

Příkaz upraví obsah ⟨picture variable⟩ tak, že odstraní vše vně zadané cyklické cesty. Neexistuje „vyšší úroveň“ tohoto příkazu, a tak musíte použít

`clip currentpicture to` ⟨path expression⟩

jestliže chcete oříznout `currentpicture`. Obrázek 1.50 znázorňuje ořezávání.

```
beginfig(50);
path p[];
p1 = (0,0){curl 0}..(5pt,-3pt)..{curl 0}(10pt,0);
p2 = p1..(p1 yscaled-1 shifted(10pt,0));
p0 = p2;
for i=1 upto 3:
  p0:=p0.. p2 shifted (i*20pt,0);
endfor
for j=0 upto 8:
  draw p0 shifted (0,j*10pt);
endfor
p3 = fullcircle shifted (.5,.5) scaled 72pt;
clip currentpicture to p3;
draw p3;
endfig;
```



Obrázek 1.50: Oříznutí obrázku.

Všechny jednoduché kreslící operace by byly nepoužitelné bez poslední operace nazvané `shipout`. Příkaz

`shipout` ⟨picture expression⟩

vypíše obrázek jako PostScriptový soubor, jehož jméno končí *.nnn*, kde `nnn` je celočíselné vyjádření hodnoty vnitřní proměnné `charcode`. (Název „`charcode`" je pro kompatibilitu s METAFONTem.) Normálně `beginfig` nastaví `charcode` a `endfig` vyvolá `shipout`.

1.9 Makra

Jak bylo zmíněno dříve, METAPOST má množinu vestavěných maker nazvaných Plain macro package a některé příkazy probírané v předchozích kapitolách jsou definovány jako makra místo toho, aby byly vestavěny do METAPOSTu. Smyslem této kapitoly je vysvětlit, jak psát taková makra.

Makra bez vstupních parametrů jsou velmi jednoduchá.

Definice makra

`def` ⟨symbolic token⟩ `=` ⟨replacement text⟩ `enddef`

⟨symbolic token⟩ je zkratkou pro ⟨replacement text⟩, kde ⟨replacement text⟩ může být vlastně posloupnost tokenů. Například, balík maker Plain by mohl definovat příkaz `fill` jako:

```
def fill = addto currentpicture contour enddef
```

Makra s parametry jsou podobná, až na to, že říkají, jak použít parametry v ⟨replacement text⟩. Například makro `rotatedaround` je definované jako:

```
def rotatedaround(expr z, d) =
  shifted -z rotated d shifted z enddef
```

`expr` v této definici znamená, že formální parametry `z` a `d` mohou být libovolné výrazy. (Měla by to být dvojice výrazů, ale METAPOSTový překladač to hned nekontroluje.)

Protože METAPOST je interpretovaný jazyk, makra s argumenty jsou něco jako podprogramy. Makra podporují lokální proměnné, smyčky a podmíněné příkazy.

1.9.1 Skupiny

Skupiny v METAPOSTu jsou pro funkce a lokální proměnné nezbytné. Skupina je posloupnost příkazů, případně následovaná výrazem, která zajistí, že jisté symbolické tokeny mohou mít své původní významy obnovené na konci skupiny. Jestliže skupina končí výrazem, chová se skupina jako volání funkce, která vrací výraz. Neboli, skupina je pouze složený příkaz. Syntaxe pro skupinu je

`begingroup` ⟨statement list⟩ `endgroup`

nebo

`begingroup` ⟨statement list⟩ ⟨expression⟩ `endgroup`

kde ⟨statement list⟩ je posloupnost příkazů oddělených středníkem. Skupina s ⟨expression⟩ se chová jako ⟨primary⟩ na schématu 1.2 nebo jako ⟨numeric atom⟩ na schématu 1.3.

Proměnné se stávají lokálními pomocí příkazu

`save` ⟨symbolic token list⟩

kde ⟨symbolic token list⟩ je čárkami oddělený seznam tokenů:

⟨symbolic token list⟩ → ⟨symbolic token⟩
| ⟨symbolic token⟩,⟨symbolic token list⟩

Všechny proměnné, jejichž jména začínají jedním z předepsaných symbolických tokenů, se stávají neznámými proměnnými a jejich aktuální hodnoty jsou uloženy pro obnovení na konci aktuální skupiny. (Je-li například název proměnné `m`, jedná se o proměnné `m`, `m.4`, `m.n`, atd.) Jestliže je příkaz `save` užit mimo skupinu, původní hodnoty jsou jednoduše vymazány. Hlavní smysl příkazu `save` je umožnit makrům používat proměnné bez křížení s existujícími proměnnými nebo proměnnými v dalších voláních stejného makra.

Hodnoty proměnné `p` se mohou měnit následovně:

```
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
```

Protože `beginfig` je makro, které začíná `begingroup` a `endfig` je makro, které končí `endgroup`, každý obrázek v METAPOSTovém souboru se chová jako skupina. Toto umožňuje obrázkům mít lokální proměnné. V kapitole 1.6.2 jsme již viděli, že jména proměnných začínající `x` nebo `y` jsou lokální proměnné (na začátku každého obrázku mají neznámé hodnoty a tyto hodnoty jsou na konci každého obrázku zapomenuty). Následující příklad ilustruje jak jednotlivá umístění fungují:

```
x23 = 3.1;
beginfig(17);
    ⋮
y3a=1; x23=2;
    ⋮
endfig;
show x23, y3a;
```

Výsledek příkazu `show`

```
>> 3.1
>> y3a
```

ukazuje, že `x23` se vrací ke své původní hodnotě `3.1` a `y3a` je zcela neznámá, jako bylo na řádku `beginfig(17)`. Toho, že proměnné `x` a `y` jsou v obrázku lokální, je dosaženo příkazem

```
save x,y
```

který je součástí makra `beginfig`.

Předdefinované makro `whatever` má ⟨replacement text⟩

`begingroup save ?; ? endgroup`

Vrací neznámou číselnou veličinu, ale ta není déle nazývána otazníkem, protože jméno je lokální ve skupině. Požadování jména přes `show whatever` vytváří

`>> %CAPSULE`*nnnn*

kde *nnnn* je identifikační číslo, které je vybráno, když `save` otazník zmizí. Přestože je příkaz `save` univerzální, nelze ho použít k lokálním změnám některých vnitřních METAPOSTových proměnných. Příkaz jako

`save linecap`

může způsobit, že METAPOST dočasně zapomene speciální význam této proměnné a pouze ji udělá neznámým číslem. Jestliže chcete nakreslit jednu přerušovanou čáru s `linecap:=butt` a potom se vrátit zpátky na předešlou hodnotu, můžete použít příkaz `interim` jako v následujícím:

```
begingroup
interimlinecap := butt;
draw ⟨path expression⟩ dashedevenly;
endgroup
```

Toto uloží hodnotu vnitřní proměnné `linecap` a dočasně ji dá novou hodnotu aniž zapomene, že `linecap` je vnitřní proměnná. Obecná syntaxe je

`interim` ⟨internal variable⟩:=⟨numeric expression⟩

1.9.2 Makra s parametry a příkaz `if`

Základní myšlenkou pro makra s parametry je dosažení větší pružnosti tím, že poskytneme pomocné informace k průchodu makrem. Už jsme viděli, že definice maker mohou mít formální parametry, které reprezentují výrazy pro zadání, když je makro voláno. Například makro definované jako

`def rotatedaround(expr z, d) =` ⟨replacement text⟩ `enddef`

lze v METAPOSTu volat ve tvaru

`rotatedaround(`⟨expression⟩`,`⟨expression⟩`)`

Klíčové slovo `expr` v definici makra znamená, že parametry mohou být výrazy libovolného typu. Když definice určuje `(expr z, d)`, formální parametry `z` a `d` se chovají jako proměnné příslušných typů. Uvnitř ⟨replacement text⟩ mohou být užity výrazy stejně jako proměnné, ale nemohou být znovu deklarovány ani přiřazeny. Neexistuje žádné omezení na neznámé nebo částečně známé argumenty. Tedy definice

```
def midpoint(expr a, b) = (.5[a,b]) enddef
```

pracuje bezchybně i pro neznámé `a` a `b`. Rovnice jako

```
midpoint(z1,z2) = (1,1)
```

může být použita pro určení `z1` a `z2`.

Všimněte si, že shora uvedená definice pro `midpoint` pracuje pro čísla, páry nebo barvy, pokud oba parametry jsou stejného typu. Jestliže z nějakého důvodu chcete vytvořit jiné makro `middlepoint`, které pracuje pro samostatnou cestu nebo obrázek, bude nezbytné udělat `if` test na typ argumentu. Využívá se skutečnosti, že existuje unární operátor

`path` ⟨primary⟩

který vrací booleovský výsledek indikující, zda argument je cesta. Protože základní `if` test má syntaxi

`if` ⟨boolean expression⟩`:` ⟨balanced tokens⟩ `else:` ⟨balanced tokens⟩ `fi`

kde ⟨balanced tokens⟩ může být cokoli, co se nekříží s `if` a `fi`. Výsledné makro `middlepoint` s testem na typ vypadá takto:

```
def middlepoint(expr a) = if path a: (point .5*length a of a)
  else: .5(llcorner a + urcorner a) fi enddef;
```

Celková syntaxe pro `if` je zobrazena na schématu 1.8. Ta povoluje vícenásobné `if` testy

`if` e_1`: ... else: if` e_2`: ... else: ... fi fi`

což lze psát zkráceně jako

`if` e_1`: ... elseif` e_2`: ... else: ... fi`

kde e_1 a e_2 představují booleovské výrazy.

Všimněte si, že `if` testy nejsou příkazy a ⟨balanced tokens⟩ v syntaxi mohou být libovolné posloupnosti souměrných tokenů a nemusí tvořit kompletní výraz nebo příkaz. Tedy na úkor srozumitelnosti by šlo makro `middlepoint` přepsat následovně:

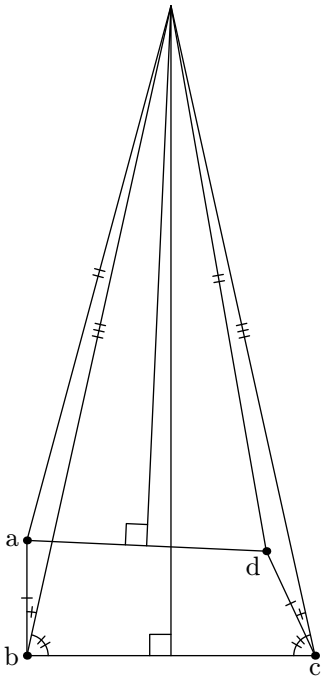
```
def middlepoint(expr a) = if path a: (point .5*length a of
  else: .5(llcorner a + urcorner fi  a) enddef;
```

Skutečný smysl maker a `if` testů je zautomatizovat opakované úkoly a umožnit, aby důležité úkoly byly řešeny samostatně. Například obrázek 1.51 užívá makra `draw_marked`, `mark_angle` a `mark_rt_angle`.

⟨if test⟩ → `if`⟨boolean expression⟩`:`⟨balanced tokens⟩⟨alternatives⟩`fi`
⟨alternatives⟩ → ⟨empty⟩
| `else:`⟨balanced tokens⟩
| `elseif`⟨boolean expression⟩`:`⟨balanced tokens⟩⟨alternatives⟩

Schéma 1.8: Syntaxe pro testy `if`.

```
beginfig(51);
pair a,b,c,d;
b=(0,0); c=(1.5in,0); a=(0,.6in);
d-c = (a-b) rotated 25;
dotlabel.lft("a",a); dotlabel.lft("b",b);
dotlabel.bot("c",c); dotlabel.llft("d",d);
z0=.5[a,d]; z1=.5[b,c];
(z.p-z0) dotprod (d-a) = 0;
(z.p-z1) dotprod (c-b) = 0;
draw a--d; draw b--c;
draw z0--z.p--z1;
draw_marked(a--b, 1);
draw_marked(c--d, 1);
draw_marked(a--z.p, 2);
draw_marked(d--z.p, 2);
draw_marked(b--z.p, 3);
draw_marked(c--z.p, 3);
mark_angle(z.p, b, a, 1);
mark_angle(z.p, c, d, 1);
mark_angle(z.p, c, b, 2);
mark_angle(c, b, z.p, 2);
mark_rt_angle(z.p, z0, a);
mark_rt_angle(z.p, z1, b);
endfig;
```



Obrázek 1.51: Ukázka použití maker.

Úkolem makra `draw_marked` je nakreslit cestu s daným počtem křížků poblíž jejího středu. Vhodným počátečním podproblémem je kreslení jednotlivých křížků kolmých k cestě `p` ve stejném čase `t`, což dělá makro `draw_mark` tak, že nejdříve nalezne vektor `dm` kolmý k cestě `p` v čase `t`. Pro zjednodušení umístění křížku je definováno makro `draw_marked`, které bere délku cesty `a` podle `p` a použije operátor `arctime` k vypočtení `t`.

Výpis maker použitých v obrázku 1.51:

```
marksize=4pt;

def draw_mark(expr p, a) =
  begingroup
  save t, dm; pair dm;
  t = arctime a of p;
  dm = marksize*unitvector direction t of p
    rotated 90;
  draw (-.5dm.. .5dm) shifted point t of p;
  endgroup
enddef;
```

```
def draw_marked(expr p, n) =
  begingroup
  save amid;
  amid = .5*arclength p;
  for i=-(n-1)/2 upto (n-1)/2:
    draw_mark(p, amid+.6marksize*i);
  endfor
  draw p;
  endgroup
enddef;

angle_radius=8pt;

def mark_angle(expr a, b, c, n) =
  begingroup
  save s, p; path p;
  p = unitvector(a-b){(a-b)rotated 90}..unitvector(c-b);
  s = .9marksize/length(point 1 of p - point 0 of p);
  if s<angle_radius: s:=angle_radius; fi
  draw_marked(p scaled s shifted b, n);
  endgroup
enddef;

def mark_rt_angle(expr a, b, c) =
  draw ((1,0)--(1,1)--(0,1))
       zscaled (angle_radius*unitvector(a-b)) shifted b
enddef;
```

Při podproblému kreslení jednotlivých značek mimo cestu `draw_marked` musí vykreslit cestu a volá `draw_mark` s příslušnou hodnotou délky oblouku. Makro `draw_marked` užívá `n` rovnoměrně rozložených `a` hodnot vycentrovaných na `.5*arclength p`.

Protože `draw_marked` pracuje i pro křivé čáry, může být použito i v makru `mark_angle`. Dané body `a`, `b` a `c` definují úhel proti směru hodinových ručiček v bodě `b`, `mark_angle` vygeneruje malý oblouk, který svírají úsečky `ba` a `bc`. Makro to dělá tak, že vytvoří oblouk `p` o poloměru jedna a vypočte měřítko `s`, které ho zvětší, aby byl zřetelně vidět.

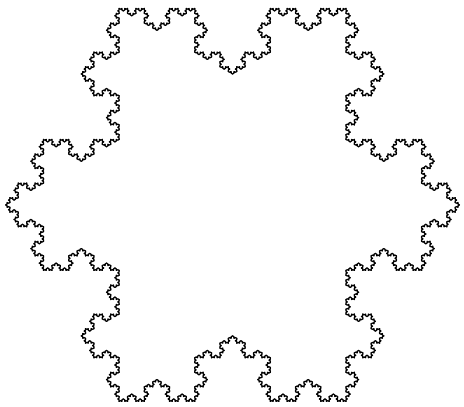
Makro `mark_rt_angle` je mnohem jednodušší. Vezme obecný pravý úhel a použije operátor `zscaled` k jeho otočení a případné změně velikosti.

Makra lze volat i rekurzivně, jak dokazuje kód uvedený na obrázku 1.52

1.9.3 Parametry typu `suffix` a `text`

Parametry maker nemusí být vždy výrazy jako v předchozích příkladech. Záměna klíčového slova `expr` za `suffix` nebo `text` v definici makra označuje, že parametry jsou názvy proměnných nebo libovolné posloupnosti tokenů. Například existuje předdefinované makro nazvané `hide`, které zpracovává textový parametr a interpretuje ho jako posloupnost příkazů i když nakonec vytváří prázdný ⟨replacement text⟩. Jinými slovy,

```
beginfig(52)
u:=3cm;
vardef koch(expr A,B,n) =
  save C; pair C;
  C = A rotatedaround(1/3[A,B], 120);
  if n>0:
    koch( A,        1/3[A,B], n-1);
    koch( 1/3[A,B], C,        n-1);
    koch( C,        2/3[A,B], n-1);
    koch( 2/3[A,B], B,        n-1);
  else:
    draw A--1/3[A,B]--C--2/3[A,B]--B;
  fi;
enddef;
z0=(u,0);
z1=z0 rotated 120;
z2=z1 rotated 120;
koch( z0, z1, 4 );
koch( z1, z2, 4 );
koch( z2, z0, 4 );
endfig;
```



Obrázek 1.52: Ukázka rekurzivního volání.

`hide` zpracuje svůj parametr a pak následuje další token, jako by se nic nestalo. Tedy

```
show hide(numeric a,b; a+b=3; a-b=1) a;
```

tiskne

```
>> 2
```

Kdyby makro `hide` nebylo předdefinováno, mohlo by být definováno následovně:

```
def ignore(expr a) = enddef;
def hide(text t) = ignore(begingroup t; 0 endgroup) enddef;
```

Příkazy reprezentované textovým parametrem `t` budou vyhodnoceny jako část skupiny, která tvoří parametr makra `ignore`. Protože `ignore` má prázdný ⟨replacement text⟩, výraz makra `hide` nakonec nic nevytvoří.

Další příklad předdefinovaného makra s textovým parametrem je `dashpattern`. Definice `dashpattern` začíná

```
def dashpattern(text t) =
  begingroup save on, off;
```

pak definuje `on` a `off` jako makra, která vytvoří požadovaný obrázek.

Textové parametry jsou velmi obecné. Jestliže chcete předat název proměnné makru, je lepší ho deklarovat jako parametr typu `suffix`. Například

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

definuje makro, které bude požadovat libovolnou číselnou proměnnou, přičte k ní jedničku a vrátí novou hodnotu. Protože názvy proměnných mohou být delší než jeden token,

```
incr(a3b)
```

je akceptovatelný, když je `a3b` číselná proměnná. Parametry typu `suffix` jsou o něco obecnější než názvy proměnných, protože definice na schématu 1.4 na straně 34 připouští, aby ⟨suffix⟩ začínal ⟨subscript⟩-em.

Obrázek 1.53 ukazuje, jak parametry typu `suffix` a `expr` mohou být použity současně. Makro `getmid` požaduje proměnnou typu cesta a vytváří pole bodů a směrů, jejichž jména jsou získána přidáním `mid`, `off` a `dir` k proměnné typu cesta. Makro `joinup` požaduje pole bodů a vytváří cestu délky `n`, která prochází každým `pt[i]` směrem `d[i]` nebo `−d[i]`.

Definice začínající

```
def joinup(suffix pt, d)(expr n) =
```

naznačuje, že volání makra `joinup` by mělo obsahovat dvoje závorky jako

```
joinup(p.mid, p.dir)(36)
```

místo

```
joinup(p.mid, p.dir, 36)
```

Ve skutečnosti jsou přípustné oba tvary. Parametry ve volání makra mohou být odděleny čárkami nebo dvojicí závorek „`)(`“. Jediné omezení je, že parametr typu `text` musí být následován pravou závorkou. Například makro `foo` s jedním parametrem typu `text` a jedním parametrem typu `expr` může být voláno

```
foo(a,b)(c)
```

v tomto případě je „`a,b`" parametr typu `text` a `c` je parametr typu `expr`, ale

```
foo(a,b,c)
```

nastaví parametr `text` na „`a,b,c`" a nechá METAPOSTový překladač hledat ještě parameter typu `expr`.

1.9.4 Makra vardef

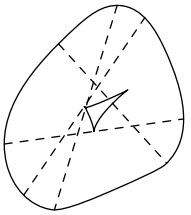
Definice makra může začínat `vardef` místo `def`. Makra definovaná tímto způsobem se nazývají *vardef* makra. Jsou vhodná především tam, kde jsou makra použita jako funkce nebo podprogramy. Hlavní myšlenkou je, že *vardef* makro je jako proměnná typu „macro“.

Místo `def` ⟨symbolic token⟩, *vardef* makro začíná

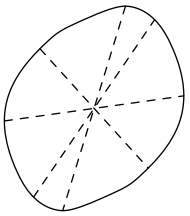
`vardef` ⟨generic variable⟩

kde ⟨generic variable⟩ je název proměnné s číselnými indexy nahrazenými obecným indexovým symbolem `[]`. Jinými slovy název proměnné následovaný `vardef` se řídí

```
def getmid(suffix p) =
  pair p.mid[], p.off[], p.dir[];
  for i=0 upto 36:
    p.dir[i] = dir(5*i);
    p.mid[i]+p.off[i] = directionpoint p.dir[i] of p;
    p.mid[i]-p.off[i] = directionpoint -p.dir[i] of p;
  endfor
enddef;

def joinup(suffix pt, d)(expr n) =
  begingroup
  save res, g; path res;
  res = pt[0]{d[0]};
  for i=1 upto n:
    g:= if (pt[i]-pt[i-1]) dotprod d[i] <0: - fi 1;
    res := res{g*d[i-1]}...{g*d[i]}pt[i];
  endfor
  res
  endgroup
enddef;
```



```
beginfig(53)
path p, q;
p = ((5,2)...(3,4)...(1,3)...(-2,-3)...(0,-5)...(3,-4)
     ...(5,-3)...cycle) scaled .3cm shifted (0,5cm);
getmid(p);
draw p;
draw joinup(p.mid, p.dir, 36)..cycle;
q = joinup(p.off, p.dir, 36);
draw q..(q rotated 180)..cycle;
drawoptions(dashed evenly);
for i=0 upto 3:
  draw p.mid[9i]-p.off[9i]..p.mid[9i]+p.off[9i];
  draw -p.off[9i]..p.off[9i];
endfor
endfig;
```



Obrázek 1.53: Další ukázka použití maker.

přesně stejnou syntaxí jako název daný v deklaraci proměnné. Je to posloupnost tagů a obecných indexových symbolů začínající tagem, kde tag je symbolický token, který není makro nebo primitivní operátor, jak bylo vysvětleno v kapitole 1.6.2.

Nejjednodušší případ je, když název *vardef* makra sestává z jednotlivých tagů. Za těchto okolností `def` a `vardef` mají zhruba stejnou funkčnost. Největší rozdíl je, že `begingroup` a `endgroup` jsou automaticky vloženy na začátek a konec ⟨replacement text⟩-u každého *vardef* makra. To dělá z ⟨replacement text⟩-u skupinu, tedy *vardef* makro se chová jako podprogram nebo funkční procedura.

Další vlastností *vardef* maker je, že připouští více-tokenové názvy maker zahrnující obecné indexy. Když název *vardef* makra obsahuje obecné indexy, číselné hodnoty musí být zadány při volání makra. Po definici makra

`vardef a[]b(expr p) =` ⟨replacement text⟩ `enddef;`

jsou `a2b((1,2))` nebo `a3b((1,2)..(3,4))` jeho platná volání. Jak však může ⟨replacement text⟩ poznat rozdíl mezi `a2b` a `a3b`? Pro tento účel jsou automaticky poskytnuty dva implicitní parametry typu `suffix`. Každé *vardef* makro má parametry typu `suffix #@` a `@`, kde `@` je poslední token z názvu volání makra a `#@` je všechno, co předchází poslednímu tokenu. Tedy `#@` je `a2`, když je jméno zadáno jako `a2b`, a `a3`, když je jméno zadáno jako `a3b`.

Předpokládejme například, že makro `a[]b` bere své argumenty a posouvá je o velikost, která závisí na jménu makra. Makro by mohlo být definováno takto:

```
vardef a[]b(expr p) = p shifted (#@,b) enddef;
```

Pak `a2b((1,2))` znamená `(1,2) shifted (a2,b)` a `a3b((1,2)..(3,4))` znamená `((1,2)..(3,4)) shifted (a3,b)`.

Je-li název makra `a.b[]`, pak `#@` je vždy `a.b` a parametr `@` bude zadán číselným indexem. Tedy `a@` by se odkazovalo na prvek pole `a[]`. Všimněte si, že `@` je parametr typu `suffix` a ne typu `expr`, tedy výrazy jako `@+1` nejsou dovoleny. Jediný způsob, jak získat číselné hodnoty indexů v parametru typu `suffix`, je jejich vyjmutím z řetězce pomocí operátoru `str`. Tento operátor bere příponu a vrací řetězec, který je zobrazený v příponě. Tedy `str @` v `a.b3` je `"3"` a `"3.14"` v `a.b3.14` nebo `a.b[3.14]`. Protože syntaxe pro ⟨suffix⟩ na schématu 1.4 požaduje, aby byly záporné indexy v závorkách, `str @` vrací `"[-3]"` v `a.b[-3]`.

Operátor `str` se obecně používá pouze v mimořádných případech. Je lepší použít parametry typu `suffix` pouze jako názvy proměnných nebo přípon. Nejlepší příklad *vardef* macra, které zahrnuje přípony, je makro `z`, které definuje `z` konvenci. Definice obsahuje speciální token `@#`, který se odkazuje na příponu následovanou názvem makra:

```
vardef z@#=(x@#,y@#) enddef;
```

To znamená, že libovolný název proměnné, jehož první token je `z`, je ekvivalentní dvojici proměnných, jejichž názvy jsou získány výměnou `z` za `x` a `y`. Například `z.a1` volá `z` makro s parametrem typu `suffix @#` nastaveným na `a1`.

Obecně

`vardef` ⟨generic variable⟩`@#`

je alternativní s `vardef` ⟨generic variable⟩, což způsobí, že METAPOSTový překladač hledá příponu následovanou názvem zadaným ve volání makra a udělá ji dostupnou jako parametr typu `suffix @#`.

Shrnutí

Vardef makra poskytují širokou třídu názvů maker, stejně jako názvy maker následované speciálním parametrem typu `suffix`. Kromě toho `begingroup` a `endgroup` jsou automaticky přidány k ⟨replacement text⟩-u *vardef* makra. Tedy užití `vardef` místo `def` v definici makra `joinup` na obrázku 1.53 se lze vyhnout nutnosti zahrnout `begingroup` a `endgroup` explicitně do definice makra.

Ve skutečnosti většina z definic maker uvedených v předchozích příkladech mohou stejně dobře použít `vardef` místo `def`. Obvykle příliš nezáleží na tom, co použijete, ale jestliže zamýšlíte makro použít jako funkci nebo podprogram je dobré pravidlo použít `vardef`. Následující srovnání by mělo pomoci v rozhodování, kdy použít `vardef`.

- *Vardef* makra jsou automaticky obklopena `begingroup` a `endgroup`.
- Název *vardef* makra může obsahovat více než jeden token a může obsahovat indexy.
- *Vardef* makro může mít přístup k příponě, která následuje název makra, při jeho volání.
- Když je symbolický token užit v názvu *vardef* makra, zůstává tagem a může být ještě použit v dalších názvech proměnných. Tedy `p5dir` je oprávněný název proměnné, ačkoliv `dir` je *vardef* makro. Naopak obyčejné makro, jako `...`, nemůže být použito jako název proměnné. (Toto je rozumné, protože `z5...z6` je pokládáno za výraz pro cestu a ne komplikovaný název proměnné).

1.9.5 Definování unárních a binárních maker

Několikrát bylo zmíněno, že některé dosud probrané operátory a příkazy jsou ve skutečnosti předdefinovaná makra. Jedná se o unární operátory jako např. `round` a `unitvector`, příkazy jako `fill` a `draw` a binární operátory jako `intersectionpoint` a `dotprod`. Hlavní rozdíl mezi těmito makry a makry, které už známe, je jejich výchozí syntaxe.

Makra `round` a `unitvector` jsou příklady toho, co je na schématu 1.2 nazváno ⟨unary op⟩. To znamená, že jsou následovány primárním výrazem. Definice makra tohoto typu by měla vypadat takto:

`vardef round primary u =` ⟨replacement text⟩ `enddef;`

Parametr `u` je parametr typu `expr` a může být použit úplně stejně jako parametr typu `expr` definovaný obvyklou syntaxí:

`(expr u).`

Jak příklad `round` naznačuje, makro může být definováno požadováním ⟨secondary⟩, ⟨tertiary⟩ nebo parametru typu ⟨expression⟩. Například předdeklarovaná definice makra `fill` je zhruba

```
def fill expr c = addto currentpicture contour c enddef;
```

Dokonce je možné definovat makro tak, aby hrálo roli ⟨of operator⟩ na schématu 1.2. Například makro `direction of` má definici v tomto tvaru :

`vardef direction expr t of p =` ⟨replacement text⟩ `enddef;`

Makra mohou být také definována tak, že se chovají jako binární operátory. Například definice makra `dotprod` má tvar

`primarydef w dotprod z =` ⟨replacement text⟩ `enddef;`

Makro `dotprod` se stává ⟨primary binop⟩. Podobně `secondarydef` a `tertiarydef` představují ⟨secondary binop⟩ a ⟨tertiary binop⟩ definice. Toto všechno definuje obyčejná makra, ne *vardef* makra; například neexistuje „`primaryvardef`".

Definice maker mohou být tedy uvozeny `def`, `vardef`, `primarydef`, `secondarydef` nebo `tertiarydef`. ⟨replacement text⟩ je seznam tokenů, které jsou souměrné vzhledem k dvojicím `def-enddef`, kde je se všemi pěti tokeny definice makra zacházeno jako s `def` ve smyslu spojování `def-enddef`.

Zbývající část syntaxe pro definice maker je shrnuta na schématu 1.9. Syntaxe obsahuje několik překvapení. Parametry makra mohou mít ⟨delimited part⟩ a ⟨undelimited part⟩. Normálně je jedna z nich prázdná, ale obecně nemusí být:

`def foo(text a) expr b =` ⟨replacement text⟩ `enddef;`

Toto definuje makro `foo`, které požaduje parametr typu `text` v závorkách následovaný výrazem.

Syntaxe také připouští ⟨undelimited part⟩ k určení argumentu typu `suffix` nebo `text`. Příklad makra s neohraničeným parametrem typu `suffix` je předdefinované makro `incr`, které je vlastně definováno jako:

```
vardef incr suffix $ = $:=$+1; $ enddef;
```

To dělá `incr` funkcí, která požaduje proměnnou, zvětší ji a vrátí novou hodnotu. Neohraničené parametry typu `suffix` mohou být v závorkách, tedy `incr a` a `incr(a)` je obojí přípustné, jestliže `a` je číselná proměnná. Existuje také podobné předdefinované makro `decr`, které odečítá 1.

Neohraničené textové parametery běží do konce příkazu. Přesněji, neohraničený parametr typu `text` je seznam tokenů následovaný voláním makra až k prvnímu „`;`" nebo „`endgroup`" nebo „`end`" až na to, že argument obsahující „`begingroup`" bude vždy zahrnovat odpovídající „`endgroup`". Příklad neohraničeného parametru typu `text` lze získat z předdefinovaného makra `cutdraw`, jehož definice je zhruba

```
def cutdraw text t =
  begingroup interim linecap:=butt; draw t; endgroup enddef;
```

To dělá `cutdraw` synonymním s `draw` s výjimkou `linecap` hodnoty. (Toto makro existuje hlavně kvůli kompatibilitě s METAFONTem.)

⟨macro definition⟩ → ⟨macro heading⟩=⟨replacement text⟩ `enddef`
⟨macro heading⟩ → `def` ⟨symbolic token⟩⟨delimited part⟩⟨undelimited part⟩
| `vardef` ⟨generic variable⟩⟨delimited part⟩⟨undelimited part⟩
| `vardef` ⟨generic variable⟩`@#`⟨delimited part⟩⟨undelimited part⟩
| ⟨binary def⟩⟨parameter⟩⟨symbolic token⟩⟨parameter⟩
⟨delimited part⟩ → ⟨empty⟩
| ⟨delimited part⟩`(`⟨parameter type⟩⟨parameter tokens⟩`)`
⟨parameter type⟩ → `expr` | `suffix` | `text`
⟨parameter tokens⟩ → ⟨parameter⟩ | ⟨parameter tokens⟩`,`⟨parameter⟩
⟨parameter⟩ → ⟨symbolic token⟩
⟨undelimited part⟩ → ⟨empty⟩
| ⟨parameter type⟩⟨parameter⟩
| ⟨precedence level⟩⟨parameter⟩
| `expr` ⟨parameter⟩ `of` ⟨parameter⟩
⟨precedence level⟩ → `primary` | `secondary` | `tertiary`
⟨binary def⟩ → `primarydef` | `secondarydef` | `tertiarydef`

Schéma 1.9: Syntaxe definicí maker.

1.9.6 Smyčky

Řada příkladů v předešlých kapitolách používala jednoduché smyčky `for` ve tvaru

`for` ⟨symbolic token⟩ `=` ⟨expression⟩ `upto` ⟨expression⟩ `:` ⟨loop text⟩ `endfor`

Smyčky, které odpočítávají, jednoduše vytvoříme náhradou `downto` za `upto`. Tato kapitola pojednává o složitějších typech posloupností, smyčkách, kde se čítač chová jako parametr typu `suffix`, a o způsobech vystoupení ze smyčky.

První zobecnění je ovlivněno skutečností, že `upto` je předdefinované makro pro

```
step 1 until
```

a `downto` je makro pro `step -1 until`. Smyčka začínající

```
for i=a step b until c
```

postupně načítá hodnoty `i`: `a`, `a` + `b`, `a` + 2`b`, ... a končí před `c`; tj., smyčka prochází hodnoty `i`, kde `i` $\leq$ `c`, je-li `b` > 0 a `i` $\geq$ `c`, je-li `i` < 0.

Užívat tuto funkci je lepší pouze v případě, kdy je velikost kroku celé číslo nebo číslo v aritmetice s pevnou řádovou čárkou (je násobkem zlomku $\frac{1}{65536}$). Jinak se chyba může zvětšovat a index smyčky nemusí dosáhnout požadované konečné hodnoty. Například hodnoty i v průběhu smyčky

```
for i=0 step .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, viz obrázky 1.2 a 1.50. 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
```

Kromě toho mohou být hodnoty iterací zadány explicitně. Libovolná posloupnost výrazů oddělených čárkami může být použita místo `a step b upto c`. Ve skutečnosti výrazy nemusí být stejného typu a nemusí mít známé hodnoty. Tak

```
for t=3.14, 2.78, (a,2a), "hello": show t; endfor
```

znázorňuje seznam čtyř hodnot.

Všimněte si, že tělo smyčky v předešlém příkladě je příkaz následovaný středníkem. Je běžné, že tělo smyčky tvoří jeden nebo více příkazů, ale nemusí to tak být. Tělo smyčky může být prakticky libovolná posloupnost tokenů, pokud dohromady dává smysl. Tedy (absurdní) příkaz

```
draw for p=(3,1),(6,2),(7,5),(4,6),(1,3): p-- endfor cycle;
```

je ekvivalentní s

```
draw (3,1)--(6,2)--(7,5)--(4,6)--(1,3)--cycle;
```

(Viz obrázek 1.19 pro reálnější příklad.)

Pohlížíme-li na smyčku jako na definici makra, pak na index smyčky lze pohlížet jako na parametr typu `expr`, který může nabývat libovolné hodnoty, ale není to proměnná a nemůže být měněn přiřazovacím příkazem. Toto umožňuje smyčka `forsuffixes`. Smyčka `forsuffixes` se podobá smyčce `for`, až na to, že index smyčky se chová jako parameter typu `suffix`. Syntaxe je

`forsuffixes` ⟨symbolic token⟩ `=` ⟨suffix list⟩ `:` ⟨loop text⟩ `endfor`

kde ⟨suffix list⟩ je čárkami oddělený seznam přípon. Jestliže jsou některé z přípon ⟨empty⟩, ⟨loop text⟩ se vykoná s parametrem indexu smyčky nastaveným na prázdnou příponu.

Dobrý příklad smyčky `forsuffixes` je definice makra `dotlabels`:

```
vardef dotlabels@#(text t) =
 forsuffixes $=t: dotlabel@#(str$,z$); endfor  enddef;
```

To by mělo objasnit, proč parametr `dotlabels` musí být čárkami oddělený seznam přípon. Většina maker tak přijímá proměnnou délku seznamů oddělených čárkami, které užívá ve smyčkách `for` nebo `forsuffixes` v tomto tvaru jako hodnoty, přes které se iteruje.

Jestliže neexistují žádné hodnoty, přes které se iteruje, můžete použít smyčku `forever`:

`forever:` ⟨loop text⟩ `endfor`

K ukončení smyčky v okamžiku, kdy booleovská proměnná dosáhne pravdivé hodnoty, se používá výstupní podmínka:

`exitif` ⟨boolean expression⟩`;`

Když METAPOSTový překladač narazí na výstupní podmínku, vyhodnotí ⟨boolean expression⟩ a ukončí danou smyčku, je-li výraz pravdivý. Potřebujeme-li ukončit smyčku po dosažení nepravdivé hodnoty, použijeme předdefinované makro `exitunless`.

Tedy METAPOSTová verze smyčky **while** je

`forever: exitunless` ⟨boolean expression⟩`;` ⟨loop text⟩ `endfor`

Výstupní podmínka může stejně dobře předcházet `endfor` nebo se vyskytovat kdekoli v ⟨loop text⟩u. Ve skutečnosti může kterýkoliv ze smyček `for`, `forever` nebo `forsuffixes` obsahovat libovolný počet výstupních podmínek.

Souhrn syntaxe pro smyčku uvedený na schématu 1.10 se explicitně nezmiňuje o výstupních podmínkách, protože ⟨loop text⟩ může být prakticky libovolná posloupnost tokenů. Jediné omezení je, že ⟨loop text⟩ musí být souměrný vzhledem k `for` a `endfor`. Samozřejmě tato souměrnost je obdobná pro `forsuffixes` a `forever`.

Získávání informací z obrázků

Od verze METAPOSTu 0.6 lze z obrázků zpětně získávat informace o tom, jak byly vykresleny. METAPOSTové obrázky jsou složeny z čar, vyplněných oblastí, textu, ořezávacích cest a *setbounds* cest (*Setbounds* cesta je cesta okolo *bounding boxu* obrázku,

⟨loop⟩ → ⟨loop header⟩ : ⟨loop text⟩`endfor`
⟨loop header⟩ → `for` ⟨symbolic token⟩ = ⟨progression⟩
| `for` ⟨symbolic token⟩ = ⟨for list⟩
| `forsuffixes` ⟨symbolic token⟩ = ⟨suffix list⟩
| `forever`
⟨progression⟩ → ⟨numeric expression⟩ `upto` ⟨numeric expression⟩
| ⟨numeric expression⟩ `downto` ⟨numeric expression⟩
| ⟨numeric expression⟩ `step` ⟨numeric expression⟩ `until` ⟨numeric expression⟩
⟨for list⟩ → ⟨expression⟩ | ⟨for list⟩ `,` ⟨expression⟩
⟨suffix list⟩ → ⟨suffix⟩ | ⟨suffix list⟩ `,` ⟨suffix⟩

Schéma 1.10: Syntaxe pro smyčky.

např. při sazbě textu pomocí `btex...etex`.). Obrázek může obsahovat mnoho komponent různých typů, které lze procházet následující smyčkou

`for` ⟨symbolic token⟩ `within` ⟨picture expression⟩ : ⟨loop text⟩ `endfor`

Počet komponent v obrázku vrací operátor

`length` ⟨picture primary⟩

⟨symbolic token⟩ je řídící proměnná smyčky, která postupně prochází komponenty obrázku ⟨picture expression⟩ v pořadí, v jakém byly nakresleny.

Komponenta pro ořezávací nebo *setbounds* cestu zahrnuje všechno, na co je daná cesta aplikována. Tedy pokud je jedna ořezávací nebo *setbounds* cesta aplikována na všechno v ⟨picture expression⟩, celý obrázek může být chápán jako jedna velká komponenta. Abychom zpřístupnili obsah takového obrázku, ignoruje v tomto případě smyčka `for...within` ořezávací nebo *setbounds* cestu.

Jakmile je komponenta obrázku nalezena smyčkou `for...within`, lze pro její identifikaci použít řady operátorů. Operátor

`stroked` ⟨primary expression⟩

testuje, zda výraz je známý obrázek, jehož první komponenta je čára. Podobně, operátory `filled` a `textual` vrací `true`, pokud první komponenta je vyplněná oblast nebo text. Operátory `clipped` a `bounded` testují, zda argument je známý obrázek, který začíná ořezávací nebo *setbounds* cestou. Vrací `true`, pokud první komponenta je ořezávací nebo *setbounds* cesta nebo zda je celý obrázek v ořezávací cestě nebo *setbounds* cestě obsažen.

Existuje také mnoho operátorů, které testují první komponentu obrázku. Pokud `p` je obrázek a `stroked p` je `true`, `pathpart p` vrací cestu čáry, `penpart p` vrací pero, které bylo použito, `dashpart p` vrací vzor čárkování čáry a barvu vrací

(`redpartp`, `greenpartp`, `bluepartp`)

Není-li čára přerušovaná, operátor `dashpart p` vrací prázdný obrázek.

Stejné operátory lze použít i pro vyplněné oblasti (`filled p` je `true`), kromě `dashpart p`, který v tomto případě nemá smysl. Pro textové komponenty (`textual p` je `true`) `textpart p` vrací řetězec textu, `fontpart p` vrací použitý font a `xpart p`, `ypart p`, `xxpart p`, `xypart p`, `yxpart p` a `yypart p` vrací parametry použité afinní transformace. Operátory `redpart`, `greenpart` a `bluepart` lze použít i pro textové komponenty.

Pro ořezávací a *setbounds* cesty (`clipped p` nebo `bounded p` je `true`) `pathpart p` vrací ořezávací nebo *setbounds* cestu a ostatní operátory nemají význam. Při použití operátorů, které nemají význam, se negeneruje chyba, ale operátory vrací nulovou hodnotu, např. `pathpart` vrací triviální cestu `(0,0)`, `penpart` vrací `nullpen`, `dashpart` prázdný obrázek, `redpart`, `greenpart` a `bluepart` vrací nulu a `textpart` nebo `fontpart` nulový řetězec.

1.10 Ladění

METAPOST odvozuje od METAFONTu mnoho prostředků pro interaktivní ladění, většina z nich zde byla stručně zmíněna. Další informace o chybových hlášeních, ladění a generování sledovaných informací můžete nalézt v *The METAFONTbook* [Knu86a].

Předpokládejme, že ve vašem vstupním souboru je na 17. řádce

```
draw z1--z2;
```

bez předešlého zadání známých hodnot `z1` a `z2`. Následující výpis zobrazuje, co překladač METAPOSTu vytiskne na terminál, když najde chybu. Aktuální chybová hláška je řádek začínající „ `!`"; následujících šest řádek přesně ukazuje, co bylo čteno, když se vyskytla chyba; a „`?`" na posledním řádku je výzva na vaši odpověď. Protože chybové hlášení mluví o nedefinované x-ové souřadnici, tato hodnota je vytištěna na první řádku za „`>>`". V tomto případě x-ová souřadnice proměnné `z1` je právě neznámá proměnná `x1`, tak překladač vytiskne název proměnné `x1` zrovna tak, jako by provedl příkaz „`show x1`".

```
>> x1
! Undefined x coordinate has been replaced by 0.
<to be read again>
                   {
--->{
     curl1}..{curl1}
l.17 draw z1--
             z2;
?
```

Na první pohled se může zdát tento výpis poněkud matoucí, ale těchto pár řádek nám říká, co bylo dosud přečteno. Každý řádek vstupu je vypsán na dvou řádcích takto:

⟨descriptor⟩ Dosud přečtený text

Text k načtení

⟨descriptor⟩ určuje zdroj vstupu. Jedná se buď o číslo řádky jako „`l.17"` pro 17. řádek aktuálního souboru nebo o název makra následovaný „`->"` nebo jde o popis v lomených závorkách. Výpis tedy říká: právě se načetl 17. řádek vstupního souboru až po „`--"`, začalo se provádět makro `--` a byla vložena otevírací složená závorka „{“, což umožňuje uživateli zadat pokyny dříve, než se začne interpretovat tento token.

Možné reakce na výzvu `?` jsou:

x ukončuje interpretr. Můžete opravit vstupní soubor a opět spustit.

h vypíše nápovědu následovanou další výzvou `?`.

⟨**return**⟩ vyzve interpretr, aby pokračoval dál.

? vypíše seznam možných voleb následovaný další výzvou `?`.

Chybová hlášená a odpovědi na příkazy `show` jsou také zapsány do logovacího souboru, jehož název je shodný s názvem hlavního vstupního souboru, až na příponu, která se změní na „`.log"`. Pozor, pokud je vnitřní proměnná `tracingonline` nastavena na svoji implicitní hodnotu, tj. na nulu, vypíšou se výsledky některých příkazů `show` podrobněji pouze do logovacího souboru.

Zatím jsme se setkali pouze s jedním typem příkazu `show`: `show` následovaný seznamem výrazů (oddělených čárkami) vypíše symbolickou reprezentaci výrazů.

Příkaz `showtoken` lze použít pro zobrazení parametrů a ⟨replacement text⟩ makra. Prochází čárkami oddělený seznam tokenů a identifikuje je. Pokud se jedná o základní token jako např. v ”`showtoken +`”, pak se pouze vypíše:

```
> +=+
```

Použijeme-li `showtoken` s proměnnou nebo `vardef` makrem, dostaneme

> ⟨token⟩=variable

Pokud chceme získat o proměnné více informací, použijeme místo `showtoken` příkaz `showvariable`. Argumentem příkazu `showvariable` je čárkami oddělený seznam symbolických tokenů a výsledkem je popis všech proměnných, jejichž jména začínají jménem ze seznamu tokenů. Obdobně i pro `vardef` makra. Např. `showvariable z` vede na

```
z@#=macro:->begingroup(x(SUFFIX2),y(SUFFIX2))endgroup
```

Existuje také příkaz `showdependencies`, který nemá žádné argumenty a vypisuje seznam všech *závislých* proměnných a závislost dosud uvedených lineárních rovnic na ostatních proměnných. Např. po příkazech

```
z2-z1=(5,10); z1+z2=(a,b);
```

`showdependencies` vypíše

```
x2=0.5a+2.5
y2=0.5b+5
x1=0.5a-2.5
y1=0.5b-5
```

Toto s výhodou použijeme, pokud budeme hledat odpověď na otázku: „Co znamená ‘`! Undefined x coordinate`’?“

Pokud vše selže, můžeme použít předdefinované makro `tracingall`, které vypíše podrobný seznam o všem, co se děje. Protože tyto informace jsou často dosti rozsáhlé, může být někdy lepší využít makro `loggingall`, které poskytne stejné informace, ale zapíše je do logovacího souboru. Existuje také makro `tracingnone`, které všechny výpisy vypíná.

Množství a formát odlaďovacích výpisů je řízeno následujícími vnitřními proměnnými. Kladná hodnota proměnné aktivuje daný formát výpisu.

`tracingcapsules` vypíše hodnoty dočasných veličin (capsules).

`tracingchoices` vypíše kontrolní body Bézierovy křivky každé nové cesty.

`tracingcommands` vypíše příkazy těsně před jejich vykonáním. Pokud je hodnota této vnitřní proměnné > 1, vypíše také podmínky `if` a smyčky. Pokud je hodnota proměnné > 2, vypíše i algebraické operace.

`tracingequations` vypíše každou proměnnou.

`tracinglostchars` upozorní na chybějící znaky, neboť se nevyskytují ve fontu použitém v příkazech `label`.

`tracingmacros` vypíše makra.

`tracingoutput` vypíše obrázky ve formátu PostScriptu.

`tracingrestores` vypíše symboly a vnitřní proměnné tak, jak jsou obnoveny na konci skupiny.

`tracingspecs` vypíše hranice vytvořené při kreslení polygonálním perem.

`tracingstats` zapíše do logovacího souboru, kolik bylo použito systémových prostředků METAPOSTu.

1.11 Operace se soubory

Od verze METAPOSTu 0.6 lze využívat i jednoduché příkazy pro práci se soubory. Nový operátor

`readfrom` ⟨file name⟩

čte ze zvoleného souboru jeden řádek a vrací ho jako řetězec. Parametr ⟨file name⟩ může být libovolný primární výraz typu řetězec. Pokud je dosaženo konce souboru nebo ze souboru nelze číst, vrací operátor řetězec obsahující jeden nulový znak. Balík maker Plain zavádí pro tento znak jméno `EOF`. Potom, co operátor `readfrom` vrátí znak `EOF`, následující čtení ze stejného souboru způsobí, že soubor bude čten opět od počátku.

Pro zápis do souboru slouží operátor

`write` ⟨string expression⟩ `to` ⟨file name⟩

Ten zapíše řádku text do zvoleného souboru, který případně nejprve otevře. Všechny tyto soubory jsou automaticky zavřeny při ukončení programu. Explicitně je lze uzavřít zapsáním `EOF` do souboru. Jediný způsob, jak zjistit, že příkaz `write` byl úspěšný, je uzavřít soubor a použít příkaz `readfrom`.

1.12 Zdroje METAPOSTu na internetu

www-cs-faculty.stanford.edu/~knuth
- domovská stránka prof. Knuthe, autora TEXu i METAFONTu.

cm.bell-labs.com/who/hobby/MetaPost.html
- domovská stránka prof. Hobbyho, autora METAPOSTu.

www.tug.org/metapost.html
- stránky o METAPOSTu na webu TEXUsers Group

www.tug.org/docs/html/metapost/
- manuál METAPOSTu v češtině, autor Robert Špalek.

mirka.janik.cz/dp
- stránka diplomové práce Mirky Krátké z Masarykovy univerzity na téma METAPOST.

bulletin.cstug.cz/pdf/bul981.pdf
- český popis možností kreslení METAFONTem.

melusine.eu.org/syracuse/metapost/
- velmi rozsáhlé stránky o METAPOSTu s mnoha obrázky a makry.

encyclopedia.thefreedictionary.com/MetaPost
- články o METAPOSTu v encyklopedii TheFreeDictionary.com.

matagalatlante.org/nobre/hyt/mpost.html
- stránka Luise Nobreho, autora METAPOSTového makra FEATPOST, které slouží k vykreslování 3D grafiky. Na stránce také naleznete pěkný popis převodu rastrových obrázků do METAPOSTu.

www-math.univ-poitiers.fr/~phan/
- stránka Anthony Phana pojednává o problému průhlednosti v METAPOSTu a popisuje makro m3D pro tvoru 3D grafiky.

www.cs.ucc.ie/~dongen/mpost/mpost.html
- stránka Marca van Dongena, kde naleznete mnoho dalších odkazů na zdroje METAPOSTu.

www.math.kth.se/~ekola/cm_arrows.html
- makra na tvorbu METAPOSTových šipek a závorek ve stylu Computer Modern.

2 Referenční manuál

Tabulky 2.1–2.6 shrnují vlastnosti Plain METAPOSTu. Vlastnosti balíku maker Plain jsou označeny symbolem †.

Tabulky v tomto dodatku udávají název dané vlastnosti, číslo stránky, na níž je vysvětlena a její krátký popis. Některé vlastnosti jsou vysvětleny pouze zde a proto u nich není uvedeno číslo stránky. Tyto vlastnosti existují především kvůli kompatibilitě s METAFONTem a jsou zřejmé. Některé další vlastnosti z METAFONTu chybí zcela, protože nejsou z hlediska uživatele METAPOSTu zajímavé a/nebo by vyžadovaly rozsáhlý popis. Všechny tyto vlastnosti jsou podrobně popsány v *The METAFONTbook* [Knu86a].

V tabulce 2.1 jsou vypsány vnitřní proměnné typu `numeric`. V tabulce 2.2 jsou vypsány předdefinované proměnné ostatních typů. Předdefinované konstanty jsou uvedeny v tabulce 2.3. Některé z nich jsou implementovány jako proměnné, jejichž hodnoty by neměly být měněny.

Tabulka 2.4 shrnuje METAPOSTové operátory, seznam možných argumentů a jejich výsledné typy. Symbol „–“ pro levý argument označuje unární operátor; symbol „–“ pro oba argumenty označuje nulární operátor. Operátory, které požadují parametry typu `suffix` nejsou v těchto tabulkách uvedeny, neboť jsou uvedeny v tabulce 2.6.

Poslední dvě tabulky jsou: tabulka 2.5 pro příkazy a tabulka 2.6 pro makra, které se chovají jako funkce nebo procedury. Tato makra vyžadují seznam argumentů v závorkách a/nebo parametry typu `suffix` a vrací buď hodnotu, jejíž typ je uveden v tabulce, nebo nic (v tabulce označeno symbolem „–“). V druhém případě se makra chovají jako procedury.

Schémata v dodatku znázorňují syntaxi METAPOSTu.

Tabulka 2.1: Vnitřní proměnné typu `numeric`.

Název	Stránka	Popis
†`ahangle`	58	úhel hrotu šipky ve stupních (implicitně: 45)
†`ahlength`	58	velikost hrotu šipky (implicitně: 4`bp`)
†`bboxmargin`	40	přípustná mezera pro `bbox` (implicitně 2`bp`)
`charcode`	66	číslo udávající příponu výstupního souboru
`day`	–	aktuální den v měsíci
†`defaultpen`	60	číselný index pro `pickup` k výběru implicitního pera
†`defaultscale`	36	měřítko fontu pro popisy (implicitně 1)
†`labeloffset`	35	vyrovnávací vzdálenost pro popisy (implicitně 3`bp`)
`linecap`	56	nastavení konců čar 0 pro butt, 1 pro round, 2 pro square
`linejoin`	56	napojení čar 0 pro mitered, 1 pro round, 2 pro beveled
`miterlimit`	57	řídí délku pokosu jako v PostScriptu
`month`	–	aktuální měsíc (např. 3 ≡ březen)
`pausing`	–	> 0 zobrazuje řádky na terminálu před jejich čtením
`prologues`	39	> 0 formátuje výstup jako PostScript s vestavěnými fonty
`showstopping`	–	> 0 zastaví po každém příkazu `show`
`time`	–	čas v minutách, kdy byl job spuštěn
`tracingcapsules`	85	> 0 vypíše také hodnoty dočasných veličin
`tracingchoices`	85	> 0 vypíše kontrolní body vybrané křivky
`tracingcommands`	85	> 0 vypíše příkazy a operace, jak jsou vykonávány
`tracingequations`	85	> 0 vypíše každou proměnnou, když se stane známou
`tracinglostchars`	85	> 0 vypíše znaky, které nejsou `infont`
`tracingmacros`	85	> 0 vypíše makra
`tracingonline`	26	> 0 zobrazí výpis na terminálu
`tracingoutput`	85	> 0 vypíše podrobný zápis vzniklého obrázku
`tracingrestores`	85	> 0 vypíše vnitřní proměnné, když jsou obnoveny
`tracingspecs`	85	> 0 vypíše cestu vzniklou pomocí polygonálního pera
`tracingstats`	85	> 0 vypíše množství využité paměti
`tracingtitles`	–	> 0 vypíše nadpisy
`truecorners`	40	> 0 vytvoří a vypíše `llcorner` atd., ignoruje `setbounds`

Tabulka 2.1: Vnitřní proměnné typu `numeric` – pokračování.

Název	Stránka	Popis
`warningcheck`	28	vypíše chybové hlášení při velikých hodnotách proměnných
`year`	–	aktuální rok (např. 2005)

Tabulka 2.2: Ostatní předdefinované proměnné.

Název	Typ	Stránka	Popis
†`background`	color	41	Barva pro `unfill` a `undraw` (obvykle bílá)
†`currentpen`	pen	64	Poslední použité pero (v příkazu `draw`)
†`currentpicture`	picture	64	Spojuje výsledky příkazů `draw` a `fill`
†`cuttings`	path	45	Část cesty odžíznutá pomocí posledního `cutbefore` nebo `cutafter`
†`defaultfont`	string	36	Fonty pro popisy
†`extra_beginfig`	string	–	Řetězec, který se vykoná jako poslední v makru `beginfig`
†`extra_endfig`	string	–	Řetězec, který se vykoná jako první v makru `endfig`

Tabulka 2.3: Předdefinované konstanty.

Název	Typ	Stránka	Popis
†`beveled`	numeric	56	Hodnota parametru `linejoin` pro šikmé spojení [2]
†`black`	color	28	Ekvivalentní s `(0,0,0)`
†`blue`	color	28	Ekvivalentní s `(0,0,1)`
†`bp`	numeric	13	Jeden PostScriptový bod v `bp` jednotkách [1] (1`bp`=1/72 palce)
†`butt`	numeric	56	Hodnota parametru `linecap` pro uříznuté zakončení [0]
†`cc`	numeric	–	Jedna typografická jednotka v `bp` jednotkách [12.79213]
†`cm`	numeric	13	Jeden centimetr v `bp` jednotkách [28.34645]

Tabulka 2.3: Předdefinované konstanty – pokračování.

Název	Typ	Stránka	Popis
†dd	numeric	–	Jeden didotův bod v bp jednotkách [1.06601]
†ditto	string	32	Znak "
†down	pair	18	Směrový vektor - dolů $(0, -1)$
†eps	numeric	–	eps:=.00049; pěkné malé kladné číslo
†epsilon	numeric	–	epsilon:=1/256/256; nejmenší kladné číslo
†evenly	picture	54	Předpis pro rovnoměrné čárkování
false	boolean	28	Booleovská hodnota *nepravda*
†fullcircle	path	41	Kruh o průměru 1 a středem v $(0, 0)$
†green	color	28	Ekvivalentní s (0,1,0)
†halfcircle	path	41	Horní polovina kruhu o průměru 1
†identity	transform	53	Identická transformace
†in	numeric	13	Jeden palec v bp jednotkách [72]
†infinity	numeric	45	Velká kladná hodnota [4095.99998]
†left	pair	18	Směrový vektor - vlevo $(-1, 0)$
†mitered	numeric	57	Hodnota parametru linejoin pro skosené spojení [0]
†mm	numeric	13	Jeden milimetr v bp jednotkách [2.83464]
nullpicture	picture	30	Prázdný obrázek
†origin	pair	–	Souřadnice $(0, 0)$
†pc	numeric	–	Jedna pika v bp jednotkách [11.95517]
pencircle	pen	60	Kruhové pero o průměru 1
†penrazor	pen	60	Čárkové pero
†penspeck	pen	60	Větší čtvercové pero
†pensquare	pen	60	Čtvercové pero 1×1
†pt	numeric	13	Jeden tiskařský bod v bp jednotkách [0.99626]
†quartercircle	path	–	První kvadrant kružnice o průměru 1
†red	color	28	Ekvivalentní s (1,0,0)
†right	pair	18	Směrový vektor - vpravo $(1, 0)$
†rounded	numeric	56	Hodnota parametru linecap a linejoin pro kruhové spojení a zakončení [1]
†squared	numeric	56	Hodnota parametru linecap pro čtvercové zakončení [2]
true	boolean	28	Booleovská hodnota *pravda*

Tabulka 2.3: Předdefinované konstanty – pokračování.

Název	Typ	Stránka	Popis
†`unitsquare`	path	–	Cesta `(0,0)--(1,0)--(1,1)--(0,1)--cycle`
†`up`	pair	18	Směrový vektor - nahoru $(0, 1)$
†`white`	color	28	Ekvivalentní s `(1,1,1)`
†`withdots`	picture	54	Vzor pro tečkovanou čáru

Tabulka 2.4: Operátory.

Název	Typy argumentů a výsledku			Str.	Popis
	Levý	Pravý	Výsledek		
`&`	string path	string path	string path	30	Spojení řetězců nebo cest
`*`	numeric	color numeric pair	color numeric pair	29	Násobení
`*`	color numeric pair	numeric	color numeric pair	29	Násobení
`**`	numeric	numeric	numeric	29	Umocňování
`+`	color numeric pair	color numeric pair	color numeric pair	29	Sčítání
`++`	numeric	numeric	numeric	29	Pythagorické sčítání $\sqrt{l^2 + r^2}$
`+-+`	numeric	numeric	numeric	29	Pythagorické odčítání $\sqrt{l^2 - r^2}$
`-`	color numeric pair	color numeric pair	color numeric pair	29	Odčítání
`-`	–	color numeric pair	color numeric pair	29	Negace
`/`	color numeric pair	numeric	color numeric pair	29	Dělení

Tabulka 2.4: Operátory – pokračování.

Název	Typy argumentů a výsledku			Str.	Popis
	Levý	Pravý	Výsledek		
`< = >` `<= >=` `<>`	string numeric pair color transform	string numeric pair color transform	boolean	28	Relační operátory
†`abs`	–	numeric pair	numeric	31	Absolutní hodnota
`and`	boolean	boolean	boolean	28	Logický součin
`angle`	–	pair	numeric	31	2−parametrický arctangent (ve stupních)
`arclength`	–	path	numeric	49	Délka cesty
`arctime of`	numeric	path	numeric	49	Čas, ve kterém délka cesty od začátku dosáhne dané hodnoty
`ASCII`	–	string	numeric	–	ASCII hodnota prvního znaku v řetězci
†`bbox`	–	picture path pen	path	40	Obdélníková cesta pro `bounding box`
`bluepart`	–	color	numeric	32	Extrahuje třetí (modrou) složku barvy
`boolean`	–	any	boolean	31	Je výraz typu boolean?
`bot`	–	numeric pair	numeric pair	60	Dolní okraj aktuálního pera se středem v zadaných souřadnicích
†`ceiling`	–	numeric	numeric	31	Nejmenší celé číslo větší nebo rovno
†`center`	–	picture path pen	pair	40	Střed `bounding` boxu
`char`	–	numeric	string	39	Znak odpovídající danému ASCII kódu
`color`	–	any	boolean	31	Je výraz typu color?
`cosd`	–	numeric	numeric	31	Kosinus úhlu ve stupních
†`counter-clockwise`	–	path	path	49	Cesta proti směru hodinových ručiček

Tabulka 2.4: Operátory – pokračování.

Název	Typy argumentů a výsledku			Str.	Popis
	Levý	Pravý	Výsledek		
†`cutafter`	path	path	path	46	Odřízne část cesty dané levým argumentem za průsečíkem cest
†`cutbefore`	path	path	path	45	Odřízne část cesty dané levým argumentem před průsečíkem cest
`cycle`	–	path	boolean	31	Je cesta uzavřená?
`decimal`	–	numeric	string	31	Převod čísla na řetězec
†`dir`	–	numeric	pair	17	Vrací směr, tj. souřadnice $(\cos\theta, \sin\theta)$, daného úhlu θ ve stupních
†`direction of`	numeric	path	pair	46	Směr cesty v daném 'čase'
†`direction-point of`	pair	path	pair	49	První bod, ve kterém má cesta daný směr
`direction-time of`	pair	path	numeric	46	První 'čas', ve kterém má cesta daný směr
†`div`	numeric	numeric	numeric	–	Celočíselné dělení $\lfloor l/r \rfloor$
†`dotprod`	pair	pair	numeric	29	Skalární součin vektorů
`floor`	–	numeric	numeric	31	Největší celé číslo menší nebo rovno
`fontsize`	–	string	numeric	36	Velikost fontu v bodech
`greenpart`	–	color	numeric	32	Extrahuje druhou (zelenou) složku barvy
`hex`	–	string	numeric	–	Převádí hexadecimální reprezentaci čísla (řetězec) na číslo
`infont`	string	string	picture	39	Obrázek daného řetězce s použitím zvoleného fontu
†`intersectionpoint`	path	path	pair	43	Průsečík cest
`intersectiontimes`	path	path	pair	44	Časy (t_l, t_r) cest l a r v jejich průsečíku
†`inverse`	–	transform	transform	50	Zpětná transformace
`known`	–	any	boolean	31	Má argument známou hodnotu?
`length`	–	path	numeric	45	Počet úseků dané cesty

Tabulka 2.4: Operátory – pokračování.

Název	Typy argumentů a výsledku			Str.	Popis
	Levý	Pravý	Výsledek		
†lft	–	numeric pair	numeric pair	60	Levý okraj aktuálního pera se středem v zadaných souřadnicích
llcorner	–	picture path pen	pair	40	Dolní levý roh bounding boxu
lrcorner	–	picture path pen	pair	40	Dolní pravý roh bounding boxu
makepath	–	pen	path	61	Uzavřená cesta vytvořená z tvaru pera
makepen	–	path	pen	60	Pero vytvořené z konvexní cesty
mexp	–	numeric	numeric	–	Funkce $\exp(x/256)$
mlog	–	numeric	numeric	–	Funkce $256\ln(x)$
†mod	numeric	numeric	numeric	–	Zbytek po celočíselném dělení $l - r\lfloor l/r \rfloor$
normal-deviate	–	–	numeric	–	Náhodné číslo s normálním rozdělením se střední hodnotou 0 a směrodatnou odchylkou 1
not	–	boolean	boolean	28	Logické negace
numeric	–	any	boolean	31	Je výraz typu numeric?
oct	–	string	numeric	–	Převádí oktalovou reprezentaci čísla (řetězec) na číslo
odd	–	numeric	boolean	–	Je nejbližší celé číslo liché?
or	boolean	boolean	boolean	28	Logický součet
pair	–	any	boolean	31	Je výraz typu pair?
path	–	any	boolean	31	Je výraz typu path?
pen	–	any	boolean	31	Je výraz typu pen?
penoffset of	pair	pen	pair	–	Tečný bod obrysu pera, který je daný směrem a je nejvíc vpravo
picture	–	any	boolean	31	Je výraz typu picture?
point of	numeric	path	pair	44	Souřadnice bodu cesty pro daný čas
postcontrol of	numeric	path	pair	–	První Bézierův řídící bod úseku cesty, který začíná v daném čase

Tabulka 2.4: Operátory – pokračování.

Název	Typy argumentů a výsledku			Str.	Popis
	Levý	Pravý	Výsledek		
`precontrol of`	numeric	path	pair	–	Poslední Bézierův řídící bod úseku cesty, který končí v daném čase
`redpart`	–	color	numeric	32	Extrahuje první (červenou) složku barvy
`reverse`	–	path	path	58	'Časové'-obrácení cesty
`rotated`	picture path pair pen transform	numeric	picture path pair pen transform	50	Rotace proti směru hodinových ručiček o daný úhel ve stupních
†`round`	–	numeric pair	numeric pair	31	Zaokrouhlí na nejbližší celé číslo
†`rt`	–	numeric pair	numeric pair	60	Pravý okraj aktuálního pera se středem v zadaných souřadnicích
`scaled`	picture path pair pen transform	numeric	picture path pair pen transform	50	Škáluje objekt danou hodnotou
`shifted`	picture path pair pen transform	pair	picture path pair pen transform	50	Posune objekt daným směrem
`sind`	–	numeric	numeric	31	Sinus úhlu ve stupních
`slanted`	picture path pair pen transform	numeric	picture path pair pen transform	50	Zešikmí objekt
†`softjoin`	path	path	path	46	Hladké navázání cest
`sqrt`	–	numeric	numeric	31	Druhá odmocnina
`str`	–	suffix	string	76	Převede příponu na řetězec
`string`	–	any	boolean	31	Je výraz typu řetězec?
`subpath of`	pair	path	path	45	Část cesty mezi danými časy

Tabulka 2.4: Operátory – pokračování.

Název	Typy argumentů a výsledku			Str.	Popis
	Levý	Pravý	Výsledek		
`substring of`	pair	string	string	30	Podřetězec ohraničený indexy
†`top`	–	numeric pair	numeric pair	60	Horní okraj aktuálního pera se středem v zadaných souřadnicích
`transform`	–	any	boolean	31	Je argument typu transform?
`transformed`	picture path pair pen transform	transform	picture path pair pen transform	53	Aplikuje transformaci na daný objekt
`turning-number`	–	path	numeric	49	Zjistí smysl otáčení cesty
`ulcorner`	–	picture path pen	pair	40	Horní levý roh `bounding boxu`
`uniform-deviate`	–	numeric	numeric	–	Náhodné číslo s rovnoměrným rozdělením mezi 0 a danou hodnotou
†`unitvector`	–	pair	pair	31	Přepočte vektor na jednotkovou délku
`unknown`	–	any	boolean	31	Je hodnota neznámá?
`urcorner`	–	picture path pen	pair	40	Horní pravý roh `bounding boxu`
†`whatever`	–	–	numeric	24	Vytvoří novou anonymní proměnnou
`xpart`	–	pair transform	number	32 53	x nebo t_x složka
`xscaled`	picture path pair pen transform	numeric	picture path pair pen transform	50	Přenásobí všechny x-ové souřadnice danou hodnotou
`xxpart`	–	transform	number	53	t_{xx} vstup transformační matice
`xypart`	–	transform	number	53	t_{xy} vstup transformační matice
`ypart`	–	pair transform	number	32 53	y nebo t_y složka

Tabulka 2.4: Operátory – pokračování.

Název	Typy argumentů a výsledku			Str.	Popis
	Levý	Pravý	Výsledek		
`yscaled`	picture path pair pen transform	numeric	picture path pair pen transform	50	Přenásobí všechny y-ové souřadnice danou hodnotou
`yxpart`	–	transform	number	53	t_{yx} vstup transformační matice
`yypart`	–	transform	number	53	t_{yy} vstup transformační matice
`zscaled`	picture path pair pen transform	pair	picture path pair pen transform	50	Otočí a vynásobí všechny souřadnice tak, že $(1, 0)$ je mapováno do daných souřadnic; tj. provádí násobení komplexním číslem

Tabulka 2.5: Příkazy.

Název	Stránka	Popis
`addto`	64	Nízkoúrovňový příkaz pro kreslení a vybarvování
`clip`	66	Ořízne obrázek podle cesty
†`cutdraw`	78	Kreslí čáry s uříznutými konci
†`draw`	15	Kreslí čáry a obrázky
†`drawarrow`	57	Kreslí čáry se šipkou na konci
†`drawdblarrow`	58	Kreslí čáry se šipkami na koncích
†`fill`	41	Vybarvuje vnitřek uzevřené cesty
†`filldraw`	59	Kreslí uzevřenou cestu a vybarví její vnitřek
`interim`	69	Lokální změna vnitřní proměnné
`let`	–	Přiřadí jednomu symbolickému tokenu význam jiného
†`loggingall`	85	Zapne ukládání všech odlaďovacích informací do log-souboru
`newinternal`	34	Deklaruje nové vnitřní proměnné
†`pickup`	28	Volba nového pera pro kreslení čar
`save`	68	Uložení proměnné
`setbounds`	40	Přiměje obrázek 'lhát' o svém bounding boxu
`shipout`	66	Nízkoúrovňový příkaz pro výstup obrázku
`show`	26	Vypíše výraz symbolicky
`showdependencies`	84	Vypíše všechny nevyřešené rovnice
`showtoken`	84	Vypíše, o jaký token se jedná
`showvariable`	84	Vypíše proměnné symbolicky
`special`	–	Vypíše řetězec přímo do PostScriptového souboru
†`tracingall`	85	Zapne výpis všech odlaďovacích informací
†`tracingnone`	85	Vypne výpis všech odlaďovacích informací
†`undraw`	59	Maže čáru nebo obrázek
†`unfill`	41	Maže vnitřek uzavřené cesty
†`unfilldraw`	59	Maže uzavřenou cestu a její vnitřek

Tabulka 2.6: Makra chovající se jako funkce.

Název	Argumenty	Výsledek	Stránka	Popis
†`buildcycle`	list of paths	path	42	Vytvoří uzavřenou cestu
†`dashpattern`	on/off distances	picture	55	Vytvoří vzor přerušované čáry
†`decr`	numeric	numeric	78	Odečítá jedničku
†`dotlabel`	suffix, picture, pair	–	35	Vyznačí bod a poblíž umístí obrázek
†`dotlabel`	suffix, string, pair	–	35	Vyznačí bod a poblíž umístí text
†`dotlabels`	suffix, point numbers	–	36	Vyznačí body `z` a jejich pořadová čísla
†`drawoptions`	drawing options	–	59	Nastaví volby pro kreslicí příkazy
†`flex`	list of pairs	path	61	Speciální makro
†`incr`	numeric	numeric	78	Přičítá jedničku
†`interpath`	numeric, path, path	path	63	Speciální makro pro *morfing* dvou cest
†`label`	suffix, picture, pair	–	35	Umístí obrázek vedle daného bodu
†`label`	suffix, string, pair	–	35	Umístí text vedle daného bodu
†`labels`	suffix, point numbers	–	36	Vykreslí pořadová čísla bodů `z`; ne body
†`max`	list of numerics	numeric	–	Najde maximum
†`max`	list of strings	string	–	Najde lexikograficky poslední řetězec
†`min`	list of numerics	numeric	–	Najde minimum
†`min`	list of strings	string	–	Najde lexikograficky první řetězec
†`superellipse`	pair, pair, pair, pair, numeric	path	63	Vytvoří ovál
†`thelabel`	suffix, picture, pair	picture	35	Jako label, ale vrací obrázek
†`thelabel`	suffix, string, pair	picture	35	Jako label, ale vrací obrázek
†`z`	suffix	pair	33	Souřadnice `x`⟨suffix⟩, `y`⟨suffix⟩)

⟨atom⟩ → ⟨variable⟩ | ⟨argument⟩
| ⟨number or fraction⟩
| ⟨internal variable⟩
| `(`⟨expression⟩`)`
| `begingroup`⟨statement list⟩⟨expression⟩`endgroup`
| ⟨nullary op⟩
| `btex`⟨typesetting commands⟩`etex`
| ⟨pseudo function⟩
⟨primary⟩ → ⟨atom⟩
| `(`⟨numeric expression⟩`,`⟨numeric expression⟩`)`
| `(`⟨numeric expression⟩`,`⟨numeric expression⟩`,`⟨numeric expression⟩`)`
| ⟨of operator⟩⟨expression⟩`of`⟨primary⟩
| ⟨unary op⟩⟨primary⟩
| `str`⟨suffix⟩
| `z`⟨suffix⟩
| ⟨numeric atom⟩`[`⟨expression⟩`,`⟨expression⟩`]`
| ⟨scalar multiplication op⟩⟨primary⟩
⟨secondary⟩ → ⟨primary⟩
| ⟨secondary⟩⟨primary binop⟩⟨primary⟩
| ⟨secondary⟩⟨transformer⟩
⟨tertiary⟩ → ⟨secondary⟩
| ⟨tertiary⟩⟨secondary binop⟩⟨secondary⟩
⟨subexpression⟩ → ⟨tertiary⟩
| ⟨path expression⟩⟨path join⟩⟨path knot⟩
⟨expression⟩ → ⟨subexpression⟩
| ⟨expression⟩⟨tertiary binop⟩⟨tertiary⟩
| ⟨path subexpression⟩⟨direction specifier⟩
| ⟨path subexpression⟩⟨path join⟩`cycle`

⟨path knot⟩ → ⟨tertiary⟩
⟨path join⟩ → −−
| ⟨direction specifier⟩⟨basic path join⟩⟨direction specifier⟩
⟨direction specifier⟩ → ⟨empty⟩
| {`curl`⟨numeric expression⟩}
| {⟨pair expression⟩}
| {⟨numeric expression⟩`,`⟨numeric expression⟩}
⟨basic path join⟩ → .. | ... | `---` | ..⟨tension⟩.. | ..⟨controls⟩..
⟨tension⟩ → `tension`⟨numeric primary⟩
| `tension`⟨numeric primary⟩`and`⟨numeric primary⟩
⟨controls⟩ → `controls`⟨pair primary⟩
| `controls`⟨pair primary⟩`and`⟨pair primary⟩

⟨argument⟩ → ⟨symbolic token⟩
⟨number or fraction⟩ → ⟨number⟩`/`⟨number⟩
| ⟨number not followed by '`/`⟨number⟩'⟩
⟨scalar multiplication op⟩ → + | −
| ⟨'⟨number or fraction⟩' not followed by '⟨add op⟩⟨number⟩'⟩

Schéma 2.1: Syntaxe pro výrazy - část 1.

⟨transformer⟩ → `rotated`⟨numeric primary⟩
| `scaled`⟨numeric primary⟩
| `shifted`⟨pair primary⟩
| `slanted`⟨numeric primary⟩
| `transformed`⟨transform primary⟩
| `xscaled`⟨numeric primary⟩
| `yscaled`⟨numeric primary⟩
| `zscaled`⟨pair primary⟩
| `reflectedabout(`⟨pair expression⟩`,`⟨pair expression⟩`)`
| `rotatedaround(`⟨pair expression⟩`,`⟨numeric expression⟩`)`

⟨nullary op⟩ → `false` | `normaldeviate` | `nullpicture` | `pencircle`
| `true` | `whatever`
⟨unary op⟩ → ⟨type⟩
| `abs` | `angle` | `arclength` | `ASCII` | `bbox` | `bluepart` | `bot` | `ceiling`
| `center` | `char` | `cosd` | `cycle` | `decimal` | `dir` | `floor` | `fontsize`
| `greenpart` | `hex` | `inverse` | `known` | `length` | `lft` | `llcorner`
| `lrcorner` | `makepath` | `makepen` | `mexp` | `mlog` | `not` | `oct` | `odd`
| `redpart` | `reverse` | `round` | `rt` | `sind` | `sqrt` | `top` | `ulcorner`
| `uniformdeviate` | `unitvector` | `unknown` | `urcorner` | `xpart` | `xxpart`
| `xypart` | `ypart` | `yxpart` | `yypart`
⟨type⟩ → `boolean` | `color` | `numeric` | `pair`
| `path` | `pen` | `picture` | `string` | `transform`
⟨primary binop⟩ → `*` | `/` | `**` | `and`
| `dotprod` | `div` | `infont` | `mod`
⟨secondary binop⟩ → + | − | ++ | +−+ | `or`
| `intersectionpoint` | `intersectiontimes`
⟨tertiary binop⟩ → `&` | `<` | `<=` | `<>` | `=` | `>` | `>=`
| `cutafter` | `cutbefore`
⟨of operator⟩ → `arctime` | `direction` | `directiontime` | `directionpoint`
| `penoffset` | `point` | `postcontrol` | `precontrol` | `subpath`
| `substring`

⟨variable⟩ → ⟨tag⟩⟨suffix⟩
⟨suffix⟩ → ⟨empty⟩ | ⟨suffix⟩⟨subscript⟩ | ⟨suffix⟩⟨tag⟩
| ⟨suffix parameter⟩
⟨subscript⟩ → ⟨number⟩ | `[`⟨numeric expression⟩`]`

⟨internal variable⟩ → `ahangle` | `ahlength` | `bboxmargin`
| `charcode` | `day` | `defaultpen` | `defaultscale` | `labeloffset`
| `linecap` | `linejoin` | `miterlimit` | `month` | `pausing`
| `prologues` | `showstopping` | `time` | `tracingoutput`
| `tracingcapsules` | `tracingchoices` | `tracingcommands`
| `tracingequations` | `tracinglostchars` | `tracingmacros`
| `tracingonline` | `tracingrestores` | `tracingspecs`
| `tracingstats` | `tracingtitles` | `truecorners`
| `warningcheck` | `year`
| ⟨symbolic token defined by `newinternal`⟩

Schéma 2.2: Syntaxe pro výrazy - část 2.

⟨pseudo function⟩ → `min(`⟨expression list⟩`)`
| `max(`⟨expression list⟩`)`
| `incr(`⟨numeric variable⟩`)`
| `decr(`⟨numeric variable⟩`)`
| `dashpattern(`⟨on/off list⟩`)`
| `interpath(`⟨numeric expression⟩`,`⟨path expression⟩`,`⟨path expression⟩`)`
| `buildcycle(`⟨path expression list⟩`)`
| `thelabel`⟨label suffix⟩`(`⟨expression⟩`,`⟨pair expression⟩`)`
⟨path expression list⟩ → ⟨path expression⟩
| ⟨path expression list⟩`,`⟨path expression⟩
⟨on/off list⟩ → ⟨on/off list⟩⟨on/off clause⟩ | ⟨on/off clause⟩
⟨on/off clause⟩ → `on`⟨numeric tertiary⟩ | `off`⟨numeric tertiary⟩

Schéma 2.3: Syntaxe pro makra chovající se jako funkce.

⟨boolean expression⟩ → ⟨expression⟩
⟨color expression⟩ → ⟨expression⟩
⟨numeric atom⟩ → ⟨atom⟩
⟨numeric expression⟩ → ⟨expression⟩
⟨numeric primary⟩ → ⟨primary⟩
⟨numeric tertiary⟩ → ⟨tertiary⟩
⟨numeric variable⟩ → ⟨variable⟩ | ⟨internal variable⟩
⟨pair expression⟩ → ⟨expression⟩
⟨pair primary⟩ → ⟨primary⟩
⟨path expression⟩ → ⟨expression⟩
⟨path subexpression⟩ → ⟨subexpression⟩
⟨pen expression⟩ → ⟨expression⟩
⟨picture expression⟩ → ⟨expression⟩
⟨picture variable⟩ → ⟨variable⟩
⟨string expression⟩ → ⟨expression⟩
⟨suffix parameter⟩ → ⟨parameter⟩
⟨transform primary⟩ → ⟨primary⟩

Schéma 2.4: Dodatečná syntaxe nutná pro kompletaci BNF-schémat (Backus Normal Form)

⟨program⟩ → ⟨statement list⟩`end`
⟨statement list⟩ → ⟨empty⟩ | ⟨statement list⟩`;`⟨statement⟩
⟨statement⟩ → ⟨empty⟩
| ⟨equation⟩ | ⟨assignment⟩
| ⟨declaration⟩ | ⟨macro definition⟩
| ⟨compound⟩ | ⟨pseudo procedure⟩
| ⟨command⟩
⟨compound⟩ → `begingroup`⟨statement list⟩`endgroup`
| `beginfig(`⟨numeric expression⟩`);`⟨statement list⟩`;endfig`

⟨equation⟩ → ⟨expression⟩`=`⟨right-hand side⟩
⟨assignment⟩ → ⟨variable⟩`:=`⟨right-hand side⟩
| ⟨internal variable⟩`:=`⟨right-hand side⟩
⟨right-hand side⟩ → ⟨expression⟩ | ⟨equation⟩ | ⟨assignment⟩

⟨declaration⟩ → ⟨type⟩⟨declaration list⟩
⟨declaration list⟩ → ⟨generic variable⟩
| ⟨declaration list⟩`,`⟨generic variable⟩
⟨generic variable⟩ → ⟨symbolic token⟩⟨generic suffix⟩
⟨generic suffix⟩ → ⟨empty⟩ | ⟨generic suffix⟩⟨tag⟩
| ⟨generic suffix⟩`[]`

⟨macro definition⟩ → ⟨macro heading⟩`=`⟨replacement text⟩`enddef`
⟨macro heading⟩ → `def`⟨symbolic token⟩⟨delimited part⟩⟨undelimited part⟩
| `vardef`⟨generic variable⟩⟨delimited part⟩⟨undelimited part⟩
| `vardef`⟨generic variable⟩`@#`⟨delimited part⟩⟨undelimited part⟩
| ⟨binary def⟩⟨parameter⟩⟨symbolic token⟩⟨parameter⟩
⟨delimited part⟩ → ⟨empty⟩
| ⟨delimited part⟩`(`⟨parameter type⟩⟨parameter tokens⟩`)`
⟨parameter type⟩ → `expr` | `suffix` | `text`
⟨parameter tokens⟩ → ⟨parameter⟩ | ⟨parameter tokens⟩`,`⟨parameter⟩
⟨parameter⟩ → ⟨symbolic token⟩
⟨undelimited part⟩ → ⟨empty⟩
| ⟨parameter type⟩⟨parameter⟩
| ⟨precedence level⟩⟨parameter⟩
| `expr`⟨parameter⟩`of`⟨parameter⟩
⟨precedence level⟩ → `primary` | `secondary` | `tertiary`
⟨binary def⟩ → `primarydef` | `secondarydef` | `tertiarydef`

⟨pseudo procedure⟩ → `drawoptions(`⟨option list⟩`)`
| `label`⟨label suffix⟩`(`⟨expression⟩`,`⟨pair expression⟩`)`
| `dotlabel`⟨label suffix⟩`(`⟨expression⟩`,`⟨pair expression⟩`)`
| `labels`⟨label suffix⟩`(`⟨point number list⟩`)`
| `dotlabels`⟨label suffix⟩`(`⟨point number list⟩`)`
⟨point number list⟩ → ⟨suffix⟩ | ⟨point number list⟩`,`⟨suffix⟩
⟨label suffix⟩ → ⟨empty⟩ | `lft` | `rt` | `top` | `bot` | `ulft` | `urt` | `llft` | `lrt`

Schéma 2.5: Souhrnná syntaxe pro METAPOSTové programy.

⟨command⟩ → `clip`⟨picture variable⟩`to`⟨path expression⟩
| `interim`⟨internal variable⟩`:=`⟨right-hand side⟩
| `let`⟨symbolic token⟩`=`⟨symbolic token⟩
| `newinternal`⟨symbolic token list⟩
| `pickup`⟨expression⟩
| `randomseed:=`⟨numeric expression⟩
| `save`⟨symbolic token list⟩
| `setbounds`⟨picture variable⟩`to`⟨path expression⟩
| `shipout`⟨picture expression⟩
| `special`⟨string expression⟩
| ⟨addto command⟩
| ⟨drawing command⟩
| ⟨font metric command⟩
| ⟨show command⟩
| ⟨tracing command⟩

⟨show command⟩ → `show`⟨expression list⟩
| `showvariable`⟨symbolic token list⟩
| `showtoken`⟨symbolic token list⟩
| `showdependencies`

⟨symbolic token list⟩ → ⟨symbolic token⟩
| ⟨symbolic token⟩`,`⟨symbolic token list⟩
⟨expression list⟩ → ⟨expression⟩ | ⟨expression list⟩`,`⟨expression⟩

⟨addto command⟩ →
`addto`⟨picture variable⟩`also`⟨picture expression⟩⟨option list⟩
| `addto`⟨picture variable⟩`contour`⟨path expression⟩⟨option list⟩
| `addto`⟨picture variable⟩`doublepath`⟨path expression⟩⟨option list⟩
⟨option list⟩ → ⟨empty⟩ | ⟨drawing option⟩⟨option list⟩
⟨drawing option⟩ → `withcolor`⟨color expression⟩
| `withpen`⟨pen expression⟩ | `dashed`⟨picture expression⟩

⟨drawing command⟩ → `draw`⟨picture expression⟩⟨option list⟩
| ⟨fill type⟩⟨path expression⟩⟨option list⟩
⟨fill type⟩ → `fill` | `draw` | `filldraw` | `unfill` | `undraw` | `unfilldraw`
| `drawarrow` | `drawdblarrow` | `cutdraw`

⟨tracing command⟩ → `tracingall` | `loggingall` | `tracingnone`

Schéma 2.6: Syntaxe pro příkazy.

⟨if test⟩ → `if`⟨boolean expression⟩`:`⟨balanced tokens⟩⟨alternatives⟩`fi`
⟨alternatives⟩ → ⟨empty⟩
| `else:`⟨balanced tokens⟩
| `elseif`⟨boolean expression⟩`:`⟨balanced tokens⟩⟨alternatives⟩

⟨loop⟩ → ⟨loop header⟩`:`⟨loop text⟩`endfor`
⟨loop header⟩ → `for`⟨symbolic token⟩`=`⟨progression⟩
| `for`⟨symbolic token⟩`=`⟨for list⟩
| `forsuffixes`⟨symbolic token⟩`=`⟨suffix list⟩
| `forever`
⟨progression⟩ → ⟨numeric expression⟩`upto`⟨numeric expression⟩
| ⟨numeric expression⟩`downto`⟨numeric expression⟩
| ⟨numeric expression⟩`step`⟨numeric expression⟩`until`⟨numeric expression⟩
⟨for list⟩ → ⟨expression⟩ | ⟨for list⟩`,`⟨expression⟩
⟨suffix list⟩ → ⟨suffix⟩ | ⟨suffix list⟩`,`⟨suffix⟩

Schéma 2.7: Syntaxe pro podmínky a smyčky.

3 Toolbox mmp

Toolbox MATLABu mmp je určen pro výstup grafického okna MATLABu do souboru ve formátu Multi-METAPOSTu. Soubory v tomto formátu lze s výhodou použít při tvorbě prezentace v LaTeXu pomocí balíků FoilTeX s PPower4, BEAMER atd. V současné době toolbox mmp zvládá převod grafických objektů `axes`, `line`, `text`, `patch`, `surface`, `image` a `rectangle`. S toolboxem lze vytvářet působivé názorné dynamické prezentace. V následujících kapitolách postupně probereme práci s toolboxem, seznámíme se s jeho strukturou a ukážeme si začlenění Multi-METAPOSTových výstupů do prezentací. Na závěr si povíme něco o případných záludnostech.

3.1 Postup práce s toolboxem

Nejprve si uvedeme postup tvorby jednoduché prezentace ve formátu pdf pomocí balíku FoilTeX s PPower4, kdy vykreslíme matematickou funkci a označíme její maximum a minimum. Při prezentaci se nejprve vykreslí matematická funkce (v našem případě sinus) a poté na stisknutí tlačítka PageDown nebo tlačítka myši se zobrazí text 'max' a na další stisknutí text 'min'.

Jednotlivé kroky tvorby prezentace:

1. Tvorba a následné spuštění programu pro vytvoření obrázku.

   Vytvořte m–soubor podle následujícího výpisu (zatím o něm moc nepřemýšlejte) a pojmenujte ho `gen_fig.m`.

```
x=linspace(0,2*pi,5);
y=sin(x);
hl=plot(x,y);                                          % line handle
ht(1)=text(pi/2+0.2,1, 'max','Color',[1 0 0]);         % text1 handle
ht(2)=text(3/2*pi+0.2,-1, 'min','Color',[1 1 0]);      % text2 handle
setappdata(hl,'mmp_layer',0);
setappdata(ht(1),'mmp_layer',10);
setappdata(ht(2),'mmp_layer',20);
axis off;
```

   Po jeho spuštění se vytvoří grafické okno MATLABu s požadovaným obrázkem.

2. Tvorba a následné přeložení METAPOSTového souboru.

   Spusťte funkci `print_mmp('fig')`.

   Dojde k vytvoření souboru fig.mmp a jeho přeložení překladačem METAPOSTu. Tím vzniknou soubory fig.0, fig.1 a fig.2.

3. Začlenění METAPOSTového souboru do textu prezentace v LaTeXu.

   Ve vašem oblíbeném editoru LaTeXu vytvořte soubor dle následujícího výpisu a pojmenujte ho `show_fig.tex`.

```
\documentclass[landscape]{foils}
\usepackage{czech}
\usepackage{pause,background,pp4slide,mpmulti}
\usepackage{graphicx}
\usepackage{hyperref}
\hypersetup{
  pdftex={true},
  pdfpagemode={FullScreen},
}

\DeclareGraphicsRule{*}{mps}{*}{}

% background definition {background.sty}
\definecolor{bgblue}{rgb}{0.04,0.39,0.53}
\vpagecolor{bgblue}

% numbering off {FoilTeX}
\rightheader{}

\begin{document}

\begin{center}
    \multiinclude[graphics={height=.9\textheight}]{fig}
\end{center}

\end{document}
```

4. Přeložení prezentace.

   Přeložte soubor show_fig.tex překladačem pdfLaTeXu a spusťte Ppower4 pro konečnou verzi prezentace.

Nyní si jednotlivé kroky podrobně vysvětlíme.

1. Před tvorbou prezentace je třeba mít rozmyšleno, v jakém pořadí se budou jednotlivé grafické prvky při prezentaci zobrazovat. Pořadí řídí uživatel přiřazením čísla vrstvy (celé číslo větší nebo rovno nule) danému grafické prvku. Pokud grafickému prvku není přiřazena žádná vrstva, je prvek zařazen do nulté vrstvy. Vrstvy budou vykreslovány od nejnižší a to tak, že po sobě jdoucí vrstvy, tj. skupina vrstev, budou vykresleny najednou. Pauza bude tedy jen tam, kde je v posloupnosti vrstev vynecháno alespoň jedno číslo. Například, budeme-li mít vrstvy $0, 1, 2, 10, 11, 20, 21, 22$, zobrazí se v prezentaci nejprve skupina $0, 1, 2$, potom $10, 11$ a na závěr $20, 21, 22$. Při zobrazování dané skupiny se jednotlivé prvky ve skupině vykreslují v pořadí daném číslem vrstvy.

   Grafický prvek se umístí do vrstvy funkcí

   `setappdata(h,'mmp_layer',i)`,

   kde `h` je identifikátor (handel) daného objektu a `i` je číslo vrstvy.

S touto funkcí jste se poprvé setkali v souboru gen_fig.m (str. 107) na řádcích 6 až 8. Na řádce 6 byla přiřazena vrstva 0 grafickému prvku s identifikátorem `hl` (sinusovka). Obdobně řádek 7 resp. 8 přiřadil vrstvu 10 resp. 20 grafickému objektu s identifikátorem `ht(1)` (text 'max') resp. `ht(2)` (text 'min').

2. Dalším krokem bylo volání funkce `print_mmp`, která vytváří METAPOSTový soubor a překládá ho. Její syntaxe je

   `print_mmp(mmp_filename)`,

   nebo

   `print_mmp(mmp_filename,options)`,

   kde `mmp_filename` je název souboru, do kterého se METAPOST zapíše. Volitelný parametr `options` je typu struktura a určuje další možné volby. Položka `options.font` určuje font textu ('hv' - Helvetica, 'cm' - Computer Modern (nastaveno implicitně)). Položku `options.preview` zadáme v případě, kdy chceme vytvořit náhled METAPOSTového obrázku ve formátu pdf ('beamer' - obrázek je zobrazen v dokumentu typu beamer, 'foils' - obrázek je zobrazen v dokumentu typu foils). Pokud chceme mít možnost změnit libovolný textový řetězec v obrázku, nastavíme `options.change_text` na hodnotu 'true'.

   My jsme použili volání `print_mmp('fig')`, které způsobí použití fontu Computer Modern, nevyvolá vytvoření náhledu a neptá se na změnu textů v obrázku.

   Jak již bylo řečeno, po volbě `options.change_text`, je možné v průběhu vytváření METAPOSTového souboru změnit veškeré textové řetězce, např. popis os, nadpis grafu a ostatní texty. Matematické výrazy v textech je nutno umístit mezi `$ $`. Ve víceřádkovém textu se jednotlivé řádky oddělují pomocí \\. V našem ukázkovém příkladu by nám funkce `print_mmp` umožnila dodatečně změnit text 'max' a 'min'.

   Vznikl následující METAPOSTový soubor (fig.mmp):[1]

```
001 % +MP-ADDITIONAL-HEADER
002 verbatimtex
003 %&latex
004 \documentclass[12pt]{article}
005 \usepackage[IL2]{fontenc}
006 \usepackage[cp1250]{inputenc}
007 %\usepackage{czech}
008
009 % Computer Modern
010
011 \newcommand{\bm}[1]{\mbox{\boldmath$#1$}}
012
013 \begin{document}
014 etex
015
016 % LaTeX font size given in \documentclass
017 latex_fontsize:=12pt;
018
```

[1] Z důvodu dalších odkazů na jednotlivé řádky souboru jsou řádky očíslovány.

```
019 % -MP-ADDITIONAL-HEADER
020
021 % -MATLAB LINE STYLE DEFINITIONS
022 numeric sc_dashed, sc_dotted, sc_dashdot;
023 sc_dashed := 1;
024 sc_dotted := 1;
025 sc_dashdot := 1;
026 picture pat_dashed;
027 pat_dashed := dashpattern(on 6bp off 6bp);
028 picture pat_dotted;
029 pat_dotted := dashpattern(on 0.5bp off 4bp);
030 picture pat_dashdot;
031 pat_dashdot := dashpattern(on 0.5bp off 4bp on 6bp off 4bp);
032
033 % -MATLAB MARKER STYLE DEFINITIONS
034 path unittriangle_up, unittriangle_down, unittriangle_left, unittriangle_right;
035 path unitdiamond, unitpentagram, unithexagram;
036
037 unittriangle_up := for i=0 step 1 until 3-1:
038 0.5*up rotated (i*360/3) --
039 endfor cycle;
040
041 unittriangle_down := for i=0 step 1 until 3-1:
042 0.5*down rotated (i*360/3) --
043 endfor cycle;
044
045 unittriangle_left := for i=0 step 1 until 3-1:
046 0.5*left rotated (i*360/3) --
047 endfor cycle;
048
049 unittriangle_right := for i=0 step 1 until 3-1:
050 0.5*right rotated (i*360/3) --
051 endfor cycle;
052
053 unitdiamond := (0,1/2) -- (-1/6,0) -- (0,-1/2) -- (1/6,0) -- cycle;
054
055 pair A,B,C,D,E;
056 A:=(0,0.5); B:=A rotated 72; C:=B rotated 72; D:=C rotated 72; E:=D rotated 72;
057 unitpentagram := A--C--E--B--D--cycle;
058
059 % AXIS LABELOFFSETS
060 xticklabel_offset:=7;
061 yticklabel_offset:=3;
062 xlabel_offset:=3;
063 ylabel_offset:=7;
064 title_offset:=0;
065
066
067 input rboxes;
068
069 % ##########################################################################
```

```
070
071 path allbounds;
072 allbounds = (0,0)--(420.00,0)--(420.00,315.00)--(0,315.00)--cycle;
073
074 % Now draw group (0) of layers (0 to 0)
075
076 beginfig(0)
077 %   MATLAB - figure - backgroud
078 fill allbounds withcolor ( 0.80,  0.80,  0.80);
079
080 %  --- layer: 0
081
082 %   MATLAB - line ()
083 % Begin polyline object
084 linecap:=0;
085 linejoin:=0;
086 path p;
087 p = (54.60,163.01)
088   --(127.64,291.37)
089   --(200.68,163.01)
090   --(273.73,34.65)
091   --(346.77,163.01);
092 picture pic;
093 pic:=nullpicture;
094 addto pic doublepath p withpen pencircle scaled 0.5 withcolor (0, 0, 1);
095 clip pic to (54.6,34.6)--(380.1,34.6)--(380.1,291.4)--(54.6,291.4)--cycle;
096 draw pic;
097 % End polyline object
098
099 setbounds currentpicture to allbounds;
100 endfig;
101
102 % Now draw group (1) of layers (10 to 10)
103
104 beginfig(1)
105 %  --- layer: 10
106
107 %   MATLAB - text ()
108 % Begin text object
109 picture pa, pb, pc;
110 pa := thelabel.rt(btex max etex scaled (10.00pt/latex_fontsize),(0,0));
111 pb = pa rotated  0.00;
112 path pd;
113 pd := bbox pb;
114 pair cb;
115 cb := 1/2[llcorner pd, urcorner pd];
116 pc = thelabel(pb,(136.94,291.37)+cb);
117 picture pic;
118 pic:=nullpicture;
119 path p;
120 p = bbox pa rotated  0.00 shifted (136.94,291.37);
```

```
121 addto pic also pc withcolor ( 1.00,  0.00,  0.00);
122 clip pic to (0.0,0.0)--(420.0,0.0)--(420.0,315.0)--(0.0,315.0)--cycle;
123 draw pic;
124 % End text object
125
126 setbounds currentpicture to allbounds;
127 endfig;
128
129 % Now draw group (2) of layers (20 to 20)
130
131 beginfig(2)
132 %  --- layer: 20
133
134 %   MATLAB - text ()
135 % Begin text object
136 picture pa, pb, pc;
137 pa := thelabel.rt(btex min etex scaled (10.00pt/latex_fontsize),(0,0));
138 pb = pa rotated  0.00;
139 path pd;
140 pd := bbox pb;
141 pair cb;
142 cb := 1/2[llcorner pd, urcorner pd];
143 pc = thelabel(pb,(283.03,34.65)+cb);
144 picture pic;
145 pic:=nullpicture;
146 path p;
147 p = bbox pa rotated  0.00 shifted (283.03,34.65);
148 addto pic also pc withcolor ( 1.00,  1.00,  0.00);
149 clip pic to (0.0,0.0)--(420.0,0.0)--(420.0,315.0)--(0.0,315.0)--cycle;
150 draw pic;
151 % End text object
152
153 setbounds currentpicture to allbounds;
154 endfig;
155
156 end
```

Poté funkce `print_mmp` spustí na METAPOSTový soubor (v našem případě soubor fig.mmp) METAPOSTový překladač, čímž vznikne řada souborů s příponami 0, 1, 2, atd. V našem případě vznikly soubory fig.0 (sinusovka), fig.1 (text 'max') a fig.2 (text 'min').

Pokud bychom ve funkci `print_mmp` zadali parametr `options.preview`, automaticky by se vytvořil náhled ve formátu pdf následujícím způsobem: vytvořil by se LaTeXovský soubor preview.tex s vloženým vygenerovaným obrázkem (fig.mmp), pak by se soubor preview.tex přeložil překladačem pdfLaTeXu. Při volbě 'foils' by se na vzniklý soubor preview.pdf aplikoval balík Ppower4 a vzniklý soubor preview.pp4.pdf by byl spuštěn prohlížečem pdf–souborů. Při volbě 'beamer' by byl vzniklý soubor preview.pdf spuštěn prohlížečem pdf–souborů.

3. V souborech preview_foils.tex nebo preview_beamer.tex, ze kterých vzniká soubor preview.tex, si povšimněte především záhlaví a příkazu `\multiinclude`.

3.2 Instalace toolboxu

Instalace toolboxu se provádí klasicky. Soubory včetně podadresářů stačí nakopírovat do vybraného adresáře a tento adresář přidat do cesty MATLABu.

Toolbox obsahuje následující soubory a adresáře:

```
print_mmp.m
print_mmp.cfg
mtlb_axes.m
mtlb_axes3.m
mtlb_image.m
mtlb_image3.m
mtlb_line.m
mtlb_line3.m
mtlb_patch.m
mtlb_patch3.m
mtlb_rectangle.m
mtlb_rectangle3.m
mtlb_surface.m
mtlb_surface3.m
mtlb_text.m
mtlb_text3.m
mp_annotation.m
mp_face.m
mp_legend.m
mp_polydot.m
mp_polyline.m
mp_rectangle.m
mp_text.m
mp_header.txt
mp_latex_header_cm.txt
mp_latex_header_hv.txt
preview_beamer.tex
preview_foils.tex

tools/tree.m
tools/onlycoreobjects.m
tools/onlycoreobjects2.m
tools/mmp_test.m
tools/create_mypause.m
```

```
private/axespos2points.m
private/data2points.m
private/data3d_to_data2d.m
private/get_bold_italic.m
private/get_labelsuffix.m
private/getaxeslim2d.m
private/repeat_label.m
private/ui_puttext.m
private/whatsee.m

demo/mp_demo1.m
demo/mp_demo2.m
demo/mp_demo3.m
demo/mp_demo4.m
demo/mp_demo5.m
demo/mp_demo6.m
demo/mp_demo7.m
demo/mp_demo8.m
demo/mp_demo9.m
demo/mp_demo10.m
demo/mp_demo11.m
demo/mp_demo12.m
demo/mp_demo13.m
demo/mp_demo14.m
demo/mp_demo15.m
demo/mp_demo16.m
demo/mp_demo17.m
demo/mp_demo18.m
demo/mp_demo19.m
demo/mp_demo20.m
demo/mp_demo21.m
demo/mp_demo22.m
```

Po instalaci je nutné v souboru print_mmp.cfg nakonfigurovat příkazy pro spuštění METAPOSTu, pdfLaTeXu a balíku PPower4. Každý příkaz je nutno uvést na samostatný řádek.

Pro bezproblémovou funkci toolboxu je nezbytná:

- funkčnost METAPOSTu,
- funkčnost pdfLaTeXu,
- správná konfigurace balíku FoilTeX s PPower4 nebo BEAMER,
- asociace přípony pdf s prohlížečem souborů pdf.

3.3 Vnitřek toolboxu

V této kapitole si probereme funkčnost toolboxu. Celý toolbox je v podstatě vystavěn na jediné funkci (`print_mmp`), která volá řadu dalších, jak je zřejmé ze schématu 3.1.

```
print_mmp.m
    mtlb_axes.m
    mtlb_axes3.m
    mtlb_image.m
    mtlb_image3.m
    mtlb_line.m
    mtlb_line3.m
    mtlb_patch.m
    mtlb_patch3.m
    mtlb_rectangle.m
    mtlb_rectangle3.m
    mtlb_surface.m
    mtlb_surface3.m
    mtlb_text.m
    mtlb_text3.m
    mp_legend.m

mtlb_axes.m
    mp_polyline.m
    mp_text.m

mtlb_axes3.m
    mp_polyline.m
    mp_text.m
```

```
mtlb_image.m
    mp_face.m

mtlb_image3.m
    mp_face.m

mtlb_line.m
    mp_polyline.m
    mp_polydot.m

mtlb_line3.m
    mp_polyline.m
    mp_polydot.m

mtlb_patch.m
    mp_face.m
    mp_polyline.m
    mp_polydot.m

mtlb_patch3.m
    mp_face.m
    mp_polyline.m
    mp_polydot.m
```

```
mtlb_rectangle.m
    mp_rectangle.m

mtlb_rectangle3.m
    mp_rectangle.m

mtlb_surface.m
    mp_face.m
    mp_polyline.m
    mp_polydot.m

mtlb_surface3.m
    mp_face.m
    mp_polyline.m
    mp_polydot.m

mtlb_text.m
    mp_text.m
    mp_annotation.m

mtlb_text3.m
    mp_text.m
```

Schéma 3.1: Struktura toolboxu mmp

Činnost funkce `print_mmp`:

1. Vyhledá v aktuálním grafickém okně legendy grafu. Pokud je najde, spustí funkci `mp_legend`, která převede legendu na klasický objekt `axes`.
2. Vyhledá v aktuálním grafickém okně všechny objekty `axes`, `image`, `line`, `patch`, `rectangle`, `surface`, a `text`. V těchto objektech vyhledává aplikační proměnnou `mmp_layer`, která udává číslo vrstvy přiřazené grafickému objektu.
3. Vytvoří skupiny spojitých vrstev podle principu uvedeného v kapitole 3.1.
4. Otevře požadovaný soubor (mmp_filename) pro zápis METAPOSTových příkazů.
5. Do souboru zkopíruje obsah souboru mp_latex_header_hv.txt nebo mp_latex_header_cm.txt, podle fontu určeném parametrem fontname, viz kapitola 3.3.1.
6. Do souboru zkopíruje obsah souboru mp_header.txt, který obsahuje společné předdefinované proměnné, jako např. typy přerušovaných čar, značky čar, vzdálenosti popisů, atd. viz kapitola 3.3.1.

7. Nastaví ořezávací cestu podle velikosti aktuálního grafického okna.
8. Prochází cyklem pro skupiny spojitých vrstev.
   - Pro každou skupinu je založen nový obrázek (beginfig).
   - Vykreslení pozadí obrázku, pokud existuje.
   - Prochází cyklem pro jednotlivé vrstvy ve skupině a vykresluje příslušné objekty.
   - Oříznutí obrázku podle ořezávací cesty.
   - Ukončení obrázku (endfig).
9. Zavře soubor mmp_filename.
10. Přeloží soubor mmp_filename METAPOSTovým překladačem.
11. Pokud je požadován náhled (je aktivováno `options.preview`):
    - vytvoří soubor LaTeXu s vloženým vygenerovaným obrázkem, viz kapitola 3.3.2,
    - vytvořený soubor přeloží překladačem pdfLaTeXu,
    - pokud je hodnota `options.preview` rovna 'foils' aplikuje se na vzniklý pdf–soubor balík Ppower4,
    - vzniklý pdf–soubor zobrazí.

3.3.1 Hlavičky v METAPOSTovém souboru

Každý METAPOSTový soubor vytvořený toolboxem `mmp` začíná dvěma hlavičkami. První hlavička obsahuje preambuli LaTeXu a závisí na volbě fontu. Druhá hlavička obsahuje společné předdefinované proměnné.

Hlavička pro font Helvetica je uvedena na schématu 3.2. Hlavička pro font Computer Modern je na řádcích 1 až 18 ve výpisu programu na straně 109. Obdobně hlavička pro předdefinované proměnné je na řádcích 19 až 70.

Pozor, v Helvetice nelze užít velká řecká písmena, malá musí být umístěna v `$\bm{  }$`.

3.3.2 Šablona LaTeXovského souboru použitá `print_mmp`

V případě, že jste aktivovali v příkazu `print_mmp` volbu `options.preview`, `print_mmp` vytvoří soubor LaTeXu, do kterého vloží odkaz na METAPOSTový obrázek. LaTeXovský soubor je vytvořen použitím odpovídající šablony. Při `options.preview='beamer'` použije šablonu `preview_beamer.tex` (schéma 3.3) a `options.preview='foils'` šablonu `preview_foils.tex` (3.4). V šabloně dojde k náhradě textu 'fig_preview' v příkazu `\multiinclude` názvem METAPOSTového souboru bez přípony.

```
% +MP-ADDITIONAL-HEADER
verbatimtex
%&latex
\documentclass[12pt]{article}
\usepackage[IL2]{fontenc}
\usepackage[cp1250]{inputenc}
%\usepackage{czech}

% Helvetica in text strings, except mathematical mode
\renewcommand\sfdefault{phv}
\renewcommand\familydefault{\sfdefault}

% Helvetica in mathematical mode
\DeclareSymbolFont{operators}   {OT1}{phv} {m}{n}
\DeclareSymbolFont{letters}     {OT1}{phv} {m}{it}
\DeclareSymbolFont{symbols}     {OMS}{phv}{m}{n}
\DeclareSymbolFont{largesymbols}{OMX}{phv}{m}{n}
\SetSymbolFont{operators}{bold}{OT1}{phv} {bx}{n}
\SetSymbolFont{letters}  {bold}{OML}{phv} {b}{it}
\SetSymbolFont{symbols}  {bold}{OMS}{phv}{b}{n}
\DeclareSymbolFontAlphabet{\mathrm}    {operators}
\DeclareSymbolFontAlphabet{\mathnormal}{letters}
\DeclareSymbolFontAlphabet{\mathcal}   {symbols}
\DeclareMathAlphabet      {\mathbf}{OT1}{phv}{bx}{n}
\DeclareMathAlphabet      {\mathsf}{OT1}{phv}{m}{n}
\DeclareMathAlphabet      {\mathit}{OT1}{phv}{m}{it}
\DeclareMathAlphabet      {\mathtt}{OT1}{phv}{m}{n}
\SetMathAlphabet\mathsf{bold}{OT1}{phv}{bx}{n}
\SetMathAlphabet\mathit{bold}{OT1}{phv}{bx}{it}

\newcommand{\bm}[1]{\mbox{\boldmath$#1$}}

\begin{document}
etex

% LaTeX font size given in \documentclass
latex_fontsize:=12pt;
```

Schéma 3.2: Hlavička METAPOSTu při použití fontu Helvetica.

```
\documentclass{beamer}
\usepackage{xmpmulti}

\DeclareGraphicsRule{*}{mps}{*}{}

\usetheme{default}

\begin{document}

\begin{frame}

\begin{center}
  \multiinclude[graphics={height=.6\textheight}]{fig_preview}
\end{center}

\end{frame}

\end{document}
```

Schéma 3.3: Šablona založená na LaTeXovské třídě beamer.

```
\documentclass[landscape]{foils}
\usepackage{czech}
\usepackage{pause,background,pp4slide,mpmulti}
\usepackage{graphicx}
\usepackage{hyperref}
\hypersetup{pdftex={true}, pdfpagemode={FullScreen},}

\DeclareGraphicsRule{*}{mps}{*}{}

% background definition {background.sty}
\definecolor{bgblue}{rgb}{0.04,0.39,0.53}
\vpagecolor{bgblue}

% numbering off {FoilTeX}
\rightheader{}

\begin{document}

\begin{center}
    \multiinclude[graphics={height=.9\textheight}]{fig_preview}
\end{center}

\end{document}
```

Schéma 3.4: Šablona založená na LaTeXovské třídě foils.

3.4 Pomocné programy

3.4.1 `mmp_test.m`

Po vytvoření obrázku v MATLABu si můžete vyzkoušet, zda vámi přidělená čísla vrstev jednotlivým grafickým prvkům odpovídají vašim představám. Pro toto otestování slouží skript `mmp_test.m`, který pracuje s aktuálním grafickým oknem MATLABu. Po jeho spuštění jsou všechny grafické prvky zneviditelněny a po stisknutí libovolné klávesy se postupně prvky zobrazují.

3.4.2 `onlycoreobjects.m`

Je-li obrázek vygenerován v Matlabu ve vyšší verzi než 6.5.1, kde se již mohou vyskytnout grafické objekty `hggroup` nebo `hgtransform`[2], je nutno před použitím funkce `print_mmp` aplikovat funkci `onlycoreobjects.m`. Tato funkce zruší všechny objekty `hdgroup` a `hgtransform`, čímž zajistí, že rodičem každého nízkoúrovňového grafického prvku (core objects) bude prvek `axes`, což je pro funkci toolboxu nezbytné. Funkce `onlycoreobjects.m` volá v případě legendy funkci `onlycoreobjects2.m`

3.5 Demonstrační příklady

Demo	Popis	Demo	Popis
`demo1`	Objekty typu `line` a `text`	`demo12`	Funkce `stairs`
`demo2`	Objekty typu `line` s legendou	`demo13`	Funkce `compass`
`demo3`	Objekty typu `patch` s legendou	`demo14`	Funkce `feather`
`demo4`	Objekty typu `rectangles`	`demo15`	Funkce `quiver`
`demo5`	Objekty typu `text`	`demo16`	Funkce `contour`
`demo6`	Vícenásobné osy	`demo17`	Funkce `contourf`
`demo7`	Funkce `area`	`demo18`	Funkce `contour` s `clabel`
`demo8`	Funkce `pie`	`demo19`	`mmp_note`
`demo9`	Funkce `hist`	`demo20`	Funkce `surface` a `image`
`demo10`	Funkce `rose`	`demo21`	`mmp_pairs`
`demo11`	Funkce `stem`	`demo22`	`mmp_pairs` ve 3D

Pro každý demonstrační příklad jsou v adresáři **demo** vždy dva soubory. Soubor s příponou 'm' je vlastní zdrojový kód příkladu a soubor s příponou 'pdf' je výsledná prezentace.

Například po spuštění demonstračního příkladu demo1.m se vytvoří soubor preview.pp4.pdf s prezentací, který odpovídá souboru demo1.pp4.pdf. Pouze v případě souborů demo21.m a demo22.m se soubor preview.pp4.pdf nevytvoří. Soubor demo21.mmp je třeba ještě doplnit následujícím kusem kódu, který se umístí před poslední end, přeložit METAPOSTovým překladačem a začlenit do vhodného souboru LaTeXu. Konečná prezentace by měla odpovídat prezentaci v souboru demo21.pp4.pdf.

[2]Přehled a strukturu použitých grafických prvků můžete snadno získat použitím funkce `tree.m`.

```
beginfig(2)
fill fullcircle scaled 3 shifted mp_origin withcolor (0.1, 0.1, 0.1);
drawarrow mp_origin{dir-20}..{dir225}mp_a withcolor 0.2(1, 0, 0);
setbounds currentpicture to allbounds;
endfig;

beginfig(3)
drawarrow mp_origin{dir-20}..{down}mp_b withcolor 0.4(1, 0, 0);
setbounds currentpicture to allbounds;
endfig;

beginfig(4)
drawarrow mp_origin{dir-20}..{dir-45}mp_c withcolor 0.6(1, 0, 0);
setbounds currentpicture to allbounds;
endfig;

beginfig(5)
drawarrow mp_origin{dir-20}..{up}mp_d withcolor 0.8(1, 0, 0);
setbounds currentpicture to allbounds;
endfig;

beginfig(6)
drawarrow mp_origin{dir-20}..{dir45}mp_e withcolor (1, 0, 0);
setbounds currentpicture to allbounds;
endfig;
```

Obdobně je třeba soubor demo22.mmp doplnit následujícím kusem kódu, který se umístí před poslední end, přeložit METAPOSTovým překladačem a začlenit do vhodného souboru LaTeXu. Konečná prezentace by měla odpovídat prezentaci v souboru demo22.pp4.pdf.

```
beginfig(1)
drawarrow (mp_max + (0,7mm))--mp_max withcolor (1, 0, 0);
setbounds currentpicture to allbounds;
endfig;

beginfig(2)
drawarrow (mp_min + (0,-7mm))--mp_min withcolor (0, 0, 1);
setbounds currentpicture to allbounds;
endfig;
```

3.6 Začlenění Multi–METAPOSTových obrázků

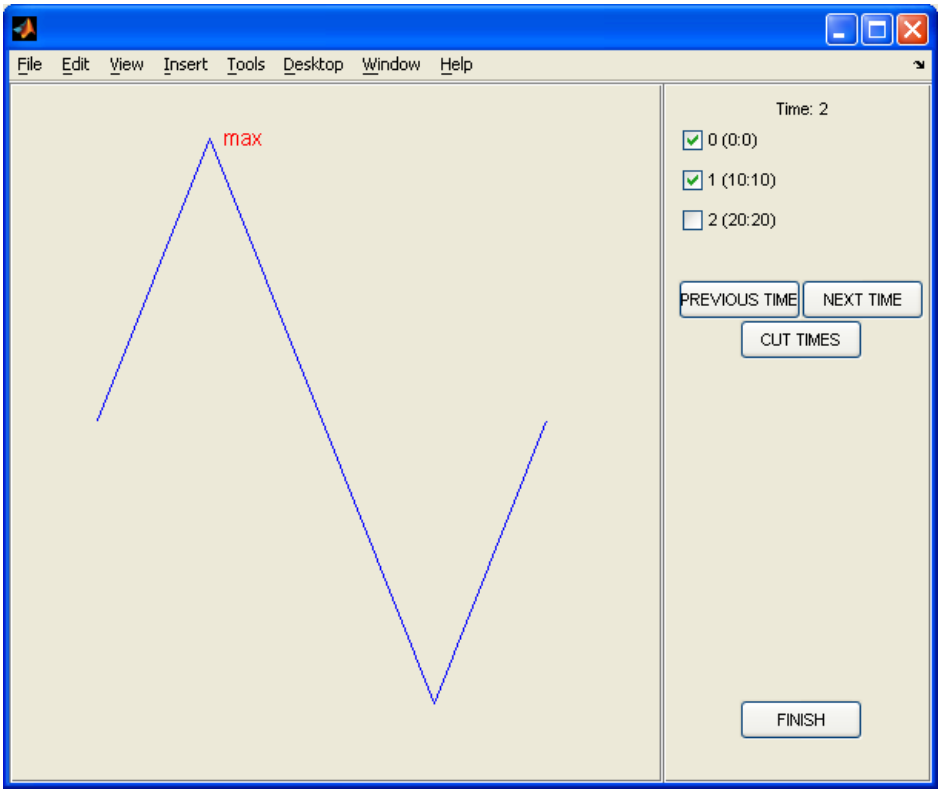
Způsob začlenění Multi–METAPOSTových obrázků do prezentace [3] založené na balíku FoilTeX a PPower4 je patrný ze schématu 3.4 na straně 117.

Pořadí vykreslování jednotlivých vrstev lze dodatečně změnit, případně umožnit mizení vrstev, pomocí volitelného parametru `pause` příkazu `multiinclude`, jak je uvedeno v [Gun02]. Pro zjednodušení tvorby LaTeXovského kódu parametru `pause` byla vytvořena funkce

`create_mypause.m`

[3] Začlenění obyčejného METAPOSTového obrázku bylo popsáno na straně 14.

Tato funkce pracuje s aktuálním grafickým oknem. Po jejím spuštění se k obsahu aktuálního grafického okna přidá po pravé straně ovládací panel funkce, viz obrázek 3.1. Funkce umožňuje zadat, jaké skupinu vrstev budou v daných časových krocích zobrazeny. Funkce na závěr vytvoří soubor, jehož obsah si uživatel může zkopírovat do své prezentace (podrobnosti viz manuál k PPower4 [Gun02]).



Obrázek 3.1: Grafické okno programu `create_pause`.

3.7 Tipy pro toolbox

- Průhledné obrázky

  Chceme-li mít obrázky průhledné, stačí v kódu MATLABu použít následující nastavení:

  ```
  set(gcf, 'Color', 'none')
  set(gca, 'Color', 'none')
  ```

- Černá ve FoilTEXu

  Vzhledem k tomu, že FoilTEX v kombinaci s balíkem PPower4 mění černou na bílou ne zcela důsledně, doporučujeme v obrázcích MATLABu použít místo černé barvy barvu *skoro* černou, např. [0.01, 0.01, 0.01], která se nezmění, a grafickým objektům, které chceme vykreslit bíle, doporučujeme přiřadit rovnou bílou barvu.

- Odstupy popisu

  V souboru `mp_header.txt` lze nastavit odstupy popisů os, názvů os a nadpisu grafu. Implicitně jsou v souboru nastaveny takto:

```
% AXIS LABELOFFSETS
xticklabel_offset:=5;
yticklabel_offset:=5;
xlabel_offset:=7;
ylabel_offset:=7;
zlabel_offset:=7;
title_offset:=0;
```

- Preambule LaTeXu

  Do souboru `mp_latex_header_hv.txt` resp. `mp_latex_header_cm.txt` lze přidat libovolnou preambuli LaTeXu, např. `usepackage`.

- Dokonalejší textové popisy

  Chceme-li do obrázku zahrnout textový popis, ze kterého vychází jedna nebo více šipek, můžeme použít skupinu aplikačních dat `mp_note`. Tato aplikační data se mohou vázat k libovolnému grafickému objektu `text`.

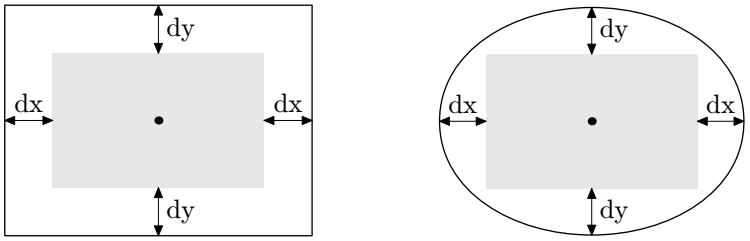
  Následující výpis části programu demonstruje jejich použití:

```
h=text(5,0,{'některé' 'zajímavé' 'body'},'FontSize',12,...
         'HorizontalAlignment','center',...
         'VerticalAlignment','middle',...
         'FontAngle','italic','rotation',0);

setappdata(h,'mmp_layer',60)
setappdata(h,'mp_note_type','rbox')
setappdata(h,'mp_note_dx2dy',0.1)
setappdata(h,'mp_note_FaceColor',[1 0 0])
setappdata(h,'mp_note_EdgeColor',[0 0 1])
setappdata(h,'mp_note_visible',2)
setappdata(h,'mp_note_points',[pi/4,3; 9/4*pi,3])
setappdata(h,'mp_note_begin_dir',[180 0])
setappdata(h,'mp_note_end_dir',[90])
setappdata(h,'mp_note_width',[2])
setappdata(h,'mp_note_color',[1 0 0; 0 0 1])
```

  Význam jednotlivých aplikačních proměnných:

Proměnná	Popis
`mp_note_type`	Typ rámečku: 'box' - obdélník, 'circle' - kruh, 'rbox' - obdélník se zakulacenými rohy
`mp_note_dx2dy`	Poměr dx/dy dle obrázku 3.2.
`mp_note_FaceColor`	Barva výplně rámečku.
`mp_note_EdgeColor`	Barva rámečku.
`mp_note_visible`	Viditelnost: 0-nic, 1-rámeček, 2-text, 3-text a rámeček
`mp_note_points`	Souřadnice konců šipek.
`mp_note_begin_dir`	Úhly, pod kterými vycházejí šipky z rámečku.
`mp_note_end_dir`	Úhly, pod kterými dopadají šipky do koncových bodů.
`mp_note_width`	Tloušťky šipek.
`mp_note_color`	Barva šipek.

V případě, že je počet parametrů proměnných `mp_note_begin_dir`, `mp_note_end_dir`, `mp_note_width` a `mp_note_color` menší než počet šipek, začnou se parametry cyklicky opakovat.



Obrázek 3.2: Význam `dx` a `dy` v rámečcích.

- Naše malé tajemství

  Pokud dopředu víte, že budete chtít modifikovat METAPOSTový soubor, může se vám hodit následující možnost toolboxu, která je založena na aplikační proměnné `mp_pairs`. V této aplikační proměnné si můžete nadefinovat souřadnice bodů a pojmenovat je. Ve funkci `print_mmp` dojde k jejich přepočtu na PostScriptové body. Později se na tyto body můžete v METAPOSTu odvolat jejich jmény. Pro správné fungování je potřeba nastavit tuto aplikační proměnnou na grafický objekt `text`, jehož vlastnost String je prázdná, viz první řádek následujícího výpisu. Na tomto výpisu je část programu demonstrující použití proměnné `mp_pairs`.

```
h=text(0,0,'');
xlim=get(gca,'XLim');
ylim=get(gca,'YLim');
setappdata(h,'mp_pairs',...
           struct('mp_a_ll',[xlim(1) ylim(1)],...
                  'mp_a_ru',[xlim(2) ylim(2)],...
                  'mp_top', [pi/2, 1]));
```

  Do souboru METAPOSTu pak můžeme přidat např. tento kód, který vykreslí dvě šipky:

```
beginfig(5)
drawarrow mp_a_ll{right}..mp_top;
drawarrow mp_a_ru{down}..mp_top;
endfig;
```

  Lze zadat i z-ovou souřadnici bodu. Ukázky použití této vlastnosti lze nalézt v ukázkových příkladech mp_demo21.m a mp_demo22.m

4 Tvorba prezentace

4.1 Nastavení pozadí

Jednobarevné pozadí

- `\usepackage{pause,background,pp4slide,mpmulti}`
- lze definovat barvu např.:
  `\definecolor{bgblue}{rgb}{0.04,0.39,0.53}`
- příkaz `\vpagecolor{bgblue}` (`\hpagecolor{bgblue}` resp. `\pagecolor{bgblue}`) nastaví zadané pozadí až do stránky, kde bude stejným příkazem barva přenastavena. `\vpagecolor`, barvu pozadí vertikálně zesvětluje, `\hpagecolor` zesvětluje horizontálně.

Obrázek na pozadí (.jpg, .pgn, .pdf)

- `\usepackage[bgadd]{background}`
- musí být nainstalovány soubory `eso-pic` a `everyshi`.
- Příkaz `\bgaddcenter{\includegraphics [width=\paperwidth,height=\paperheight]{obrazek.png}}` nastaví zadané pozadí až do stránky, kde bude stejným příkazem pozadí přenastaveno. Obrázek na pozadí lze zrušit také příkazem `\bgclear`, pak je pozadí modré.

4.2 Převod rastrového obrázku do METAPOSTu

V některých případech je výhodné převést obrázek v rastrovém formátu, většinou získaný skenováním, na obrázek ve formátu vektorovém. Pro tento převod lze úspěšně použít program `autotrace`[1], kterým převedeme rastrový formát do PostScriptu. Například příkazem

```
autotrace -background-color=FFFFFF filename.tif > filename.ps
```

převedeme soubor filename.tif do souboru filename.ps, přičemž program při konverzi nepřevádí bílé pozadí.

Pro následný převod PostScriptového souboru do METAPOSTu použijeme program `pstoedit`[2].

[1] http://autotrace.sourceforge.net
[2] http://www.pstoedit.net

4.3 Změna velikosti PostScriptového souboru

V případě, že potřebujete změnit velikost PostScriptového souboru, můžete použít program `epsffit` z balíku `PSutils` [3].

[3] http://freshmeat.net/projects/psutils

Literatura

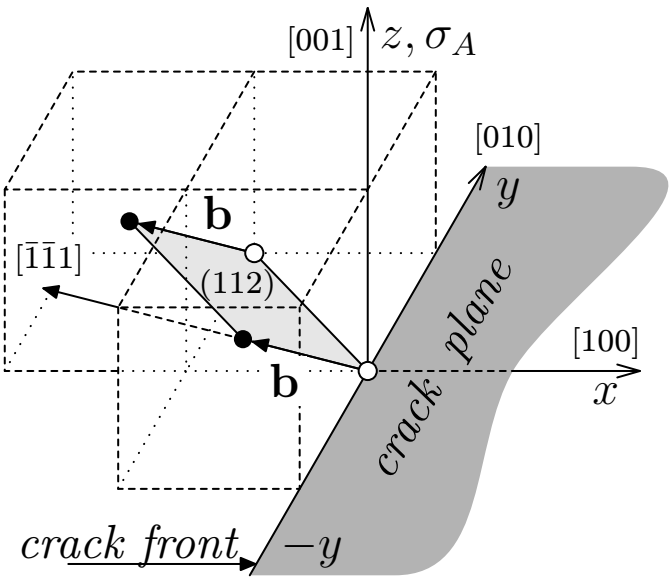
[Ado86] Adobe Systems Inc., *PostScript language reference manual*, Addison Wesley, Reading, Massachusetts, 1986.

[Gun02] K. Guntermann, *Ppower4 manual*, TU Darmstadt, www-sp.iti.informatik.tu-darmstadt.de/software/ppower4, 2002, Version 0.4.

[Hob86] J. D. Hobby, *Smooth, easy to compute interpolating splines*, Discrete and Computational Geometry **1** (1986), no. 2.

[Knu86a] D. E. Knuth, *The METAFONTbook*, Addison Wesley, Reading, Massachusetts, 1986, Volume C of *Computers and Typesetting*.

[Knu86b] ———, *The TEXbook*, Addison Wesley, Reading, Massachusetts, 1986, Volume A of *Computers and Typesetting*.



Rejstřík