METAPOST

РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

John D. Hobby
и команда разработки MetaPost

версия документа: 1.004

Примечания переводчика

В версии 1.004 MetaPost руководство пользователя для пакета `boxes` было вынесено в отдельный документ. В переводе этого разделения не произошло, но добавления к документации были учтены. Создания этого перевода было бы невозможным без Ольги Гагаркиной.

Владимир Лидовский, litwr@yandex.ru

Содержание

1 Введение

MetaPost — это язык программирования, очень похожий на METAFONT[1] [4] Кнута с тем исключением, что он производит PostScript-программы вместо растровых картинок. Заимствования из METAFONT— это базовые средства для создания и манипулирования картинками. Они включают числа, координатные пары, кубические сплайны, аффинные трансформации, текстовые строки и булевы величины. Дополнительные средства делают возможными соединение текста и графики и доступ к специальным возможностям PostScript[2] таким как вырезка, затенение, пунктирные линии. Другое свойство, заимствованное у METAFONT, — это способность решать заданные неявно линейные уравнения, что позволяет писать многие программы в значительной мере в декларативном стиле. Мощь и гибкость MetaPost достигаются построением сложных

[1] METAFONT — это торговая марка компании Addison Wesley Publishing.
[2] PostScript — это торговая марка Adobe Systems.

операций из более простых.

MetaPost особенно хорошо приспособлен для генерации картинок для технических документов, где некоторые свойства рисунка могут контролироваться математическими или геометрическими ограничениями, которые наилучшим образом выражаются в символьной форме. Другими словами, MetaPost не занимает место средств для ручного рисования или даже интерактивных графических редакторов. Это настоящий язык программирования для генерации графики и, особенно, иллюстраций для документов TEX[3] и troff.

Для использования MetaPost вы должны приготовить входной файл с Metapost-кодом и затем вызвать сам MetaPost при помощи, как правило, команды в форме

`mpost` ⟨имя файла⟩

Синтаксис и имя программы являются системо-зависимыми, иногда она зовется `mp`. Входные файлы для MetaPost обычно имеют имена, заканчивающиеся на "`.mp`", и эта часть имени может опускаться при вызове MetaPost. Например, для входного файла `foo.mp`

```
mpost foo
```

вызовет MetaPost и произведет выходные файлы с именами типа `foo.1` и `foo.2`. Все сообщения, появляющиеся на терминале, собираются в файл-дубликат с именем `foo.log`. Туда включаются сообщения об ошибках и все команды MetaPost, введенные интерактивно.[4]

Файл-дубликат начинается с заголовочной строки, идентифицирующей используемую вами версию MetaPost. Вы можете также определить текущую версию из программы MetaPost через строковую константу `mpversion` (это стало возможным с версии 0.9). Например,

```
if known mpversion:
  message "mp = " & mpversion;
  if scantokens(mpversion) < 1: message "Поддержка цветов CMYK недоступна!" fi
fi
```

печатает

```
mp = 1.004
```

Команда `scantokens` описана на с. 19 и может быть употреблена для конвертирования строк в числа. Номер версии также включается в комментарий `Creator` в Postscript-выводе.

Этот документ представляет язык MetaPost, начиная с простейших для использования и наиболее важных для простых приложений свойств. Чтение руководства не требует знания METAFONT или доступа к *The METAFONTbook*, но обе возможности будут полезными. Первые несколько разделов описывают язык таким, каким он кажется пользователю-новичку с ключевыми параметрами, зафиксированными на предопределенных значениях. Некоторые свойства, определяемые в этих разделах, — часть макропакета с именем Plain. Следующие разделы охватывают весь язык и отличают примитивы от макросов из автоматически загружаемого макропакета Plain. Вследствие того, что большая часть языка идентична METAFONT Кнута, приложение дает детальное сравнение таким образом, что опытные пользователи смогут узнать больше о MetaPost, читая *The METAFONTbook* [4].

Документация к MetaPost дополняется "Drawing Boxes with MetaPost" — руководством к пакету `graph`, изначально разработанному Джоном Д. Хобби.

Домашняя страница MetaPost — `http://tug.org/metapost`. Она содержит ссылки на много дополнительной информации, включая множество статей, которые написаны о MetaPost. При поиске подсказки попробуйте рассылку на `metapost@tug.org`; вы можете подписаться туда на `http://tug.org/mailman/listinfo/metapost`.

[3] TEX — это торговая марка American Mathematical Society.

[4] Знак `*` используется для приглашения к интерактивному вводу и знак `**` показывает, что ожидается имя входного файла. Диалога можно избежать вызовом MetaPost с файлом, который заканчивается командой `end`.

Текущая разработка размещена на https://foundry.supelec.fr/projects/metapost/; посетите этот сайт для контактов с членами текущей команды разработчиков, загрузки исходников и многого другого.

Пожалуйста, сообщайте об ошибках и требуемых улучшениях в список на metapost@tug.org или через адреса, приведенные выше. Пожалуйста, не посылайте больше отчеты напрямую Dr. Hobby.

2 Базовые команды для рисования

Простейшие команды рисования — для генерации прямых линий. Таким образом,

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

рисует диагональную линию и

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

рисует ломаную линию, подобную этой:

MetaPost также имеет команду `drawdot` для печати одной точки, например, `drawdot(30,0)`.

Что означается координатами подобными `(30,0)`? MetaPost использует ту же самую типовую систему координат, что и PostScript. Это значит, что `(30,0)` — это 30 единиц вправо от начала координат, где единица — это $\frac{1}{72}$ дюйма. Мы будем ссылаться на эту единицу измерения по-умолчанию как на *PostScript-пункт* для отличения его от стандартного для принтеров пункта, который равен $\frac{1}{72.27}$ дюйма.

MetaPost использует те же имена для единиц измерения, что и TEX и METAFONT. Таким образом, `bp` ссылается на PostScript-пункты ("большие пункты"), а `pt` — на пункты принтера. Другие единицы измерения включают `in` для дюймов, `cm` для сантиметров и `mm` для миллиметров. Например,

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

генерирует больший вариант диаграммы выше. Будет верно сказать `0` вместо `0cm`, потому что `cm` в действительности только множитель преобразования и `0cm` только умножает этот множитель на ноль. (MetaPost понимает конструкции подобные `2cm` как сокращение для `2*cm`).

Удобно ввести свой собственный масштабирующий множитель, скажем u. Затем вы можете определить координаты относительно u и позже решать, хотите ли вы начать с `u=1cm` или `u=0.5cm`. Это даст вам контроль над тем, что масштабируемо и над тем, что нет, т. к. изменение u не повлияет на такие свойства как толщина линий.

Есть много путей изменять вид линии сверх простого изменения ее толщины, однако механизмы управления шириной вводят много общих понятий, которыt нам пока еще не нужны. Соответствующие команды могут странно выглядеть, например, команда

```
pickup pencircle scaled 4pt
```

устанавливает толщину линии в 4 пункта для последующей команды `draw`. (Это примерно в 8 раз больше стандартной толщины линии).

С такой большой толщиной даже линия длины ноль выглядит как большая жирная точка. Мы можем это использовать для создания решетки из жирных точек, имея одну команду

`drawdot` для каждого узла решетки. Такая повторяющаяся последовательность команд `draw` записывается наилучшим образом как пара вложенных циклов:

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

Внешний цикл исполняется для $i = 0, 1, 2$, а внутренний — для $j = 0, 1, 2$. Результат — решетка три на три из жирных точек, как показано на рис. 1. Этот рисунок включает также больший вариант ломаной линии, которую мы видели раньше.

```
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:  drawdot (i*u,j*u);  endfor
endfor
endfig;
```

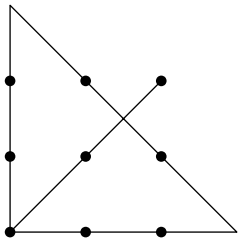


Рис. 1: Команды MetaPost и результирующий вывод

Заметьте, что программа на рис. 1 начинается с `beginfig(2)` и заканчивается с `endfig`. Эти макросы, выполняющие административные функции, гарантируют, что результаты всех команд `draw` собираются вместе и транслируются в PostScript. Входной файл для MetaPost обычно содержит последовательность пар `beginfig` и `endfig` с командой `end` после последней пары. Если этот файл именован `fig.mp`, то вывод от команд `draw` между `beginfig(1)` и следующей `endfig` пишется в файл `fig.1`. Другими словами, числовой аргумент в макросе `beginfig` определяет имя соответствующего выходного файла.

Что делать со всеми этими PostScript-файлами? Они могут быть включены как рисунки в документы TEX или troff, если вы имеете драйвер, который может работать с PostScript-картинками. Они также могут быть предварительно просмотрены до их включения в документ с тысячей страниц. Следующие разделы дают больше информации.

3 Управление выводом MetaPost

Взаимодействие между TEX и MetaPost может быть двояким. С одной стороны, графика MetaPost может импортироваться в документы, набираемые TEX и его друзьями. С другой стороны, MetaPost может поручить набор текстовых элементов TEX, LATEX или *troff*, например, текстовых меток или математических формул в графике. Таким способом графика MetaPost может легко принять стиль документа (шрифт, размер шрифта и т. п.) и соответствовать качеству его набора (использовать кернинг, лигатуры и т. п.). Это делает MetaPost идеальным инструментом для приготовления высококачественной графики для документов TEX или *troff*.

Этот раздел относится к первой стороне взаимодействия TEX–MetaPost: импорту графики MetaPost в TEX и его друзей. Набор текстов в MetaPost обсуждается в разделе 8.

3.1 Предварительный просмотр графики MetaPost

Вывод MetaPost — это вариант PostScript, называемый Encapsulated PostScript (EPSF). Графика MetaPost может, следовательно, быть просмотрена в любом PostScript-просмотрщике, например, GSview.

Ситуация становится только немного сложнее, когда вывод MetaPost содержит текст. Обычно MetaPost не производит самодостаточные EPS-файлы, например, шрифты и таблицы кодировок не помещаются в вывод. Поэтому вывод MetaPost, содержащий текст, может быть показан с неверными шрифтами, неверными символами или вообще без текста в PostScript-просмотрщике. Долгое время, наиболее надежный путь для просмотра был в подготовке тест-документа, включающего все картинки MetaPost, обработке его либо TEX, либо LATEX, затем `dvips` и показу результирующего `ps`-файла в PostScript-просмотрщике.[5]

Однако, с версии 1.000 MetaPost ситуация изменилась. С этой версии MetaPost способен производить самодостаточные файлы EPS, которые могут быть достоверно просмотрены в независимости от того, есть ли в них текст или нет. Новые возможности могут быть включены установкой внутренней переменной MetaPost `prologues` в 3. Смотри раздел 8.1 для большей информации о `prologues`.

3.2 Использование графики MetaPost в TEX, LATEX, pdfLATEX, pdfTEX, ConTEXt и troff

То как рисунки MetaPost могут быть интегрированы с документами, подготовленными в TEX и родственных TEX программах, зависит от формата документов и драйвера вывода. Рис. 2 показывает процесс работы для plain TEX, LATEX и свободно доступной программы `dvips`[6]. Схожая процедура работает с troff: процессор вывода `grops` включает рисунки на PostScript, когда они запрашиваются через команду troff `\X`. С использованием PDF с TEX и LATEX ситуация несколько иная. Следующие абзацы дают краткую информацию по некоторым популярным TEX-форматам и драйверам вывода.

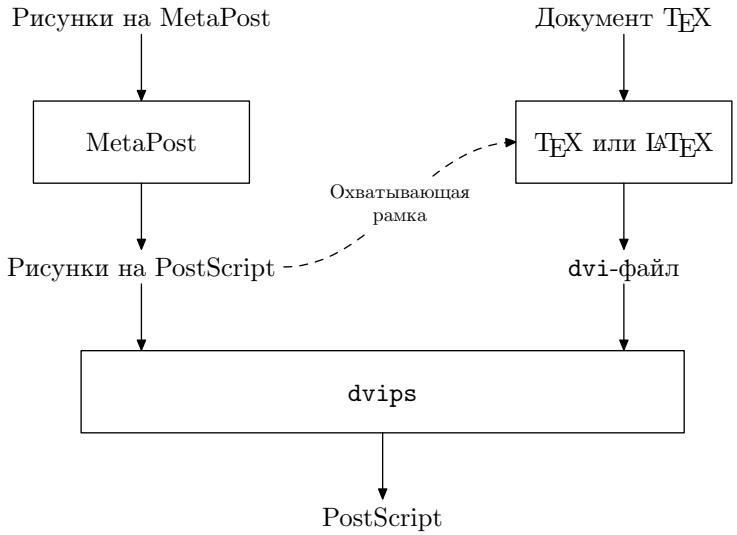


Рис. 2: Диаграмма обработки для TEX-документа с рисунками в MetaPost

TEX Пользователи TEX могут импортировать графику, загрузив сначала пакет `epsf` через `\input epsf` и затем введя команду

`\epsfbox{`⟨имя файла⟩`}`

[5] Хотя есть и альтернативы: `mpstoeps` — это Perl-сценарий, который автоматизирует процесс, обозначенный выше, а `mptopdf` — это другое средство, которое конвертирует MetaPost-файлы в PDF.

[6] Исходники на C для `dvips` находятся вместе с web2c TEX-дистрибутивом. Подобные программы доступны и в других местах.

для загрузки EPS-файла, например, `\epsfbox{fig.1}`.

LaTeX Для документов LaTeX процедура похожая: первый пакет `graphicx` должен быть загружен размещением `\usepackage{graphicx}` в преамбулу документа и затем EPS-файлы могут быть загружены через

`\includegraphics{`⟨имя файла⟩`}`,

например, `\includegraphics{fig.1}`.

Как можно заметить на рис. 2 графические файлы никогда не включаются при исполнении TeX или LaTeX. Вместо этого TeX и LaTeX только читают информацию об охватывающих рамках из PostScript-файла, резервируя столько места на странице, сколько занимает графика и записывая ссылку на соответствующий файл в `dvi`-выводе. Графический файл включается только при последующем исполнении драйвера вывода, который может обрабатывать PostScript-файлы, например, `dvips`.

pdfLaTeX Приложение pdfLaTeX, когда исполняется в режиме PDF, является сразу и LaTeX-интерпретатором, и драйвером вывода для документа в PDF-формат. Поэтому графические файлы включаются во время исполнения pdfLaTeX, за один проход. В отличие от `dvips`, pdfLaTeX не может обрабатывать обычные PostScript-файлы — он может работать только с так называемыми очищенными EPS-файлами, которые могут использовать только ограниченное множество возможностей языка PostScript. К счастью, вывод MetaPost — это и *есть* очищенный EPS, так что тут нам повезло. Из того, что `mps` — это типовое расширение pdfLaTeX для очищенных EPS-файлов, а вывод MetaPost обычно имеет расширения-числа, мы должны

- сказать pdfLaTeX обрабатывать занумерованные файлы MetaPost согласно правилам для файлов `mps` или
- изменить расширение файла вывода MetaPost на `mps`.

При первом подходе мы должны добавить строку

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

к преамбуле документа после загрузки пакета `graphicx`. Эта декларация скажет pdfLaTeX загружать *все* файлы с неизвестным расширением как `mps`-файлы. См. документацию по пакетам `graphicx` и `graphics` для дополнительной информации.

С версии MetaPost 1.000 рекомендован второй подход.[7] Примитив MetaPost `filenametemplate` может быть использован для установки расширения файла вывода MetaPost в `mps` (см. раздел 3.3) Поэтому декларация `\DeclareGraphicsRule` здесь не нужна. Более того, расширение может опускаться в команде `\includegraphics`.

LaTeX и pdfLaTeX Если вы хотите сохранить гибкость и возможность компилировать как LaTeX, так и pdfLaTeX, то нужно позаботиться о некоторых вещах. Стандартная декларация `\DeclareGraphicsRule` может быть активирована только, если pdfLaTeX исполняется в PDF-режиме. Поэтому универсальная декларация должна выглядеть подобно этой:

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

[7] Для обработки расширений MetaPost-файлов предположительно более естественно вместо LaTeX-исходников использовать исходники MetaPost. Тем более, что установка расширения MetaPost-файла в `mps` предохраняет от загрязнения избытком расширений — вам будет нужно зарегистрировать только одно расширение для вашего PostScript-просмотрщика — `.mps`, вместо `.0`, `.1`, `.2` и т. д.

Если вы используете метод `filenametemplate`, то расширение файла `mps` не следует опускать в команде `\includegraphics`, т. к. `mps` — это не часть имени LaTeX-файла, заполняемая по-умолчанию. Если расширение `mps` присутствует, то LaTeX обрабатывает эти файлы как `eps`-файлы, что очевидно является корректным. Для дополнительной информации см. описание `\DeclareGraphicsExtensions` и `\DeclareGraphicsRule` в документации пакетов `graphicx` и `graphics`.

pdfTeX Пользователи plain pdfTeX должны ознакомиться с отдельной программой `mptopdf`, которую можно найти в http://context.aanhet.net/mptopdf.htm.

ConTeXt В ConTeXt поддержка MetaPost интегрирована в ядро. Отдельно от встроенной графики (см. руководство по MetaFun) можно также встраивать графику извне командой `\externalfigure`. Занумерованные файлы распознаются автоматически, как графика с `mps`-расширением. Специальные свойства, такие как затенение, прозрачность, включение рисунков, цветовое пространство и подобные обрабатываются автоматически. Практически пользователи ConTeXt будут вероятно определять графику MetaPost в документе-исходнике, который использует некоторые новшества, например, более естественный интерфейс со свойствами документа, поддержка шрифтов и автоматическая обработка. Поддержка включений MetaPost представлена в версиях MkII и MkIV, но используемые методы слегка различаются. Будущие версии MkIV будут поддерживать даже более тесную интеграцию.

troff Также возможно включать вывод MetaPost в GNU *troff*-документ. Макропакет `-mpspic` определяет команду `.PSPIC`, которая включает EPS-файл. Например, команда *troff*

```
.PSPIC fig.1
```

включает `fig.1`, используя заданные в файле охватывающей рамкой естественные высоту и ширину образа.

3.3 Шаблоны имен файлов

MetaPost поддерживает шаблоны для выходных файлов. Эти шаблоны используют стиль `printf` escape-последовательностей и пересчитываются перед тем, как рисунок записывается на диск

Здесь используется команда `filenametemplate`, которая воспринимает строку-аргумент. Ее несложный синтаксис:

```
filenametemplate "%j-%3c.mps";
beginfig(1);
   draw p;
endfig;
```

Если исходный файл сохранялся как `fig.mp`, то будет создан выходной файл `fig-001.mps` вместо `fig.1`. Маленькое множество возможных escape-последовательностей см. в таблице 1.

Примитив `filenametemplate` может быть также полезным для именования графических файлов индивидуально и еще для хранения всех MetaPost-исходников в одном файле. Например,

Escape-последовательность	Смысл
%%	Знак процента
%j	Имя текущей работы
%⟨0-9⟩c	Значение charcode
%⟨0-9⟩y	Текущий год
%⟨0-9⟩m	Номер месяца
%⟨0-9⟩d	День месяца
%⟨0-9⟩H	Час
%⟨0-9⟩M	Минута

Таблица 1: Разрешенные escape-последовательности для filenametemplate

соберем исходники разных диаграмм в один файл fig.mp

```
filenametemplate "fig-quality.mps";
beginfig(1);
  ...
endfig;

filenametemplate "fig-cost-vs-productivity.mps";
beginfig(2);
  ...
endfig;
```

— может оказаться проще вспомнить правильное имя диаграммы в документе TEX, чем нумерованное имя файла. Заметьте, что аргумент beginfig не используется при отсутствии образца %c в строке шаблона имени файла.

Для обеспечения совместимости со старыми файлами начальное значение filenametemplate устанавливается в %j.%c. Если вы присвоите пустую строку, то это будет означать возврат к начальному значению.

4 Кривые

MetaPost совершенно счастлив при рисовании как кривых, так и прямых линий. Команда draw с разделенными .. аргументами-точками рисует плавную кривую через эти точки. Например, посмотрите на результат

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

после определения пяти точек таким образом:

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

Рис. 3 показывает кривую через точки, помеченные от z0 до z4

Есть много других способов нарисовать путь через те же самые пять точек. Для получения гладкой замкнутой кривой соедините z4 с началом добавлением ..cycle к команде draw как показано на рис. 4a. Также возможно в одной команде draw смешивать кривые и прямые линии как показано на рис. 4b. Просто используйте -- там, где вы хотите прямые линии, и .. там, где вы хотите кривые. Таким образом,

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

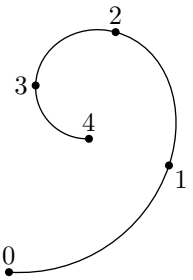


Рис. 3: Результат `draw z0..z1..z2..z3..z4`

произведет кривую через точки 0, 1, 2 и 3, затем ломаную линию из точки 3 в точку 4 и обратно в точку 0. Результат будет точно таким же как после двух команд рисования

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

и

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

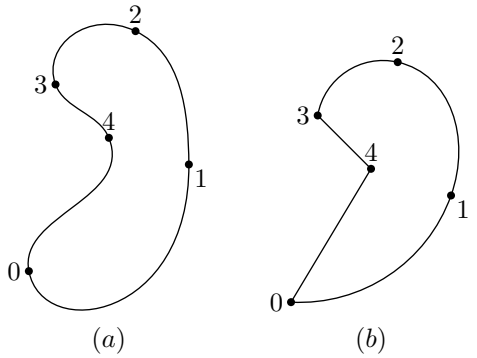


Рис. 4: (a) Результат `draw z0..z1..z2..z3..z4..cycle`; (b) Результат `draw z0..z1..z2..z3--z4--cycle`.

4.1 Кубические кривые Безье

Когда MetaPost просят нарисовать гладкую кривую через последовательность точек, он конструирует кусочную кубическую кривую с непрерывным уклоном и с приблизительно непрерывной кривизной. Это значит, что спецификация пути такая как

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

дает в результате кривую, что может быть определена параметрически как $(X(t),Y(t))$ для $0 \leqslant t \leqslant 5$, где $X(t)$ и $Y(t)$ — кусочные кубические функции. Таким образом существуют различные пары кубических функций для каждого ограниченного целыми числами интервала для t. Если `z0` $= (x_0, y_0)$, `z1` $= (x_1, y_1)$, `z2` $= (x_2, y_2)$, ..., то MetaPost выбирает контрольные точки Безье (x_0^+, y_0^+), (x_1^-, y_1^-), (x_1^+, y_1^+), ..., где

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

для $0 \leq t \leq 1$. Точные правила для выбора контрольных точек Безье приведены в [2] и в *METAFONTbook* [4].

Для того, чтобы путь имел непрерывный уклон в (x_i, y_i) входящее и исходящее направления в $(X(i), Y(i))$ должны соответствовать. Таким образом, вектора

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

должны иметь одинаковое направление, т. е. (x_i, y_i) должна быть на отрезке линии между (x_i^-, y_i^-) и (x_i^+, y_i^+). Эта ситуация иллюстрируется на рис. 5, где контрольные точки Безье, выбираемые MetaPost, соединены пунктирными линиями. Для тех, кто знаком с интересными свойствами такой конструкции, MetaPost позволяет специфицировать контрольные точки напрямую в следующем формате:

```
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);
```

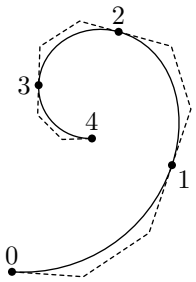


Рис. 5: Результат `draw z0..z1..z2..z3..z4` с автоматически выбираемой управляющей ломаной Безье, иллюстрируемой пунктирными линиями.

4.2 Спецификация направления, напряжения и изгиба

MetaPost обеспечивает много способов управления поведением пути кривой без действительного указания контрольных точек. Например, некоторые точки на пути могут быть выбраны как вертикальный или горизонтальный экстремумы. Если `z1` следует быть горизонтальным экстремумом, а `z2` — вертикальным, то вы можете указать, что $(X(t), Y(t))$ должна идти вверх в `z1` и влево в `z2`:

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

Картинка-результат, показанная на рис. 6, имеет желаемые вертикальное и горизонтальное направления в `z1` и `z2`, но она не выглядит такой плавной, как кривая на рис. 3. Это обусловлено большим разрывом в величине кривизны в `z1`. Если явно не указать направление в `z1`, то MetaPost-интерпретатор выберет направление таким, чтобы кривизна над `z1` была почти такой же как и кривизна под этой точкой.

Как может выбор направлений в данных точках на кривой определять будет ли кривизна непрерывной? Ответ в том, что кривые, используемые в MetaPost, пришли из семейства, где путь определяется своими концами и направлениями там. Рисунки 7 и 8 дают хорошую идею о том, на что похоже это семейство кривых.

Рисунки 7 и 8 иллюстрируют новые возможности MetaPost. Первая — это оператор `dir`, который по углу в градусах генерирует единичный вектор в этом направлении. Таким образом,

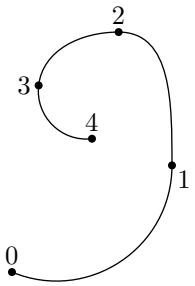
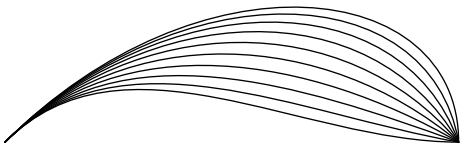
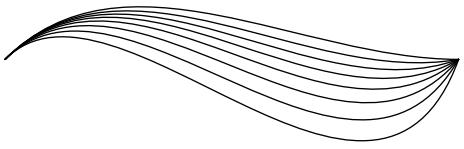


Рис. 6: Результат `draw z0..z1{up}..z2{left}..z3..z4`.



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

Рис. 7: Семейство кривых и инструкции MetaPost для его генерации



```
beginfig(8)
for a=0 upto 7:
  draw (0,0){dir 45}..{dir 10a}(6cm,0);
endfor
endfig;
```

Рис. 8: Другое семейство кривых с соответствующими инструкциями MetaPost

`dir 0` эквивалентен `right` и `dir 90` эквивалентен `up`. Есть также готовые вектора направлений `left` и `down` для `dir 180` и `dir 270`.

Вектора направлений, заданные в `{}`, могут быть любой длины и они могут как входить в точку, так и выходить из нее. Возможно даже в спецификации пути иметь оба направления для одной точки, до и после. Например, участок спецификации пути

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

произведет кривую с углом в $(10, 0)$.

Заметьте, что некоторые кривые на рис. 7 имеют точки перегиба. Это необходимо при создании гладкой кривой в ситуации подобной рис. 4a, но это вероятно нежелательно при работе с вертикальными и горизонтальными точками экстремума как показано на рис. 9a. Если `z1` нужно сделать наивысшей точкой на кривой, то это можно получить, используя `...` вместо `..` в спецификации пути как показано на рис. 9b. Значение `...` — это "выбрать свободный от перегибов путь между этими точками, если направления в концовых точках дают такую возможность". На рис. 7 возможно избавиться от перегибов, а на рис. 8 нет.

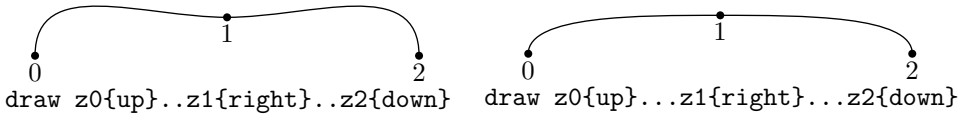


Рис. 9: Две команды `draw` и кривая-результат.

Другой способ управлять неподходящим путем в увеличении параметра "напряжение". Использование `..` в спецификации пути устанавливает параметр напряжения в типовое значение 1. Если это делает некоторую часть пути слишком дикой, то мы можем выборочно увеличить напряжение. Если рис. 10a рассматривается как "слишком дикий", то команда `draw` в следующей форме увеличит напряжение между `z1` и `z2`:

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

Это произведет рис. 10b. Для асимметричного эффекта подобного рис. 10c, команда `draw` получает вид

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

Параметр напряжения может быть меньшим единицы, но он должен быть не менее $\frac{3}{4}$.

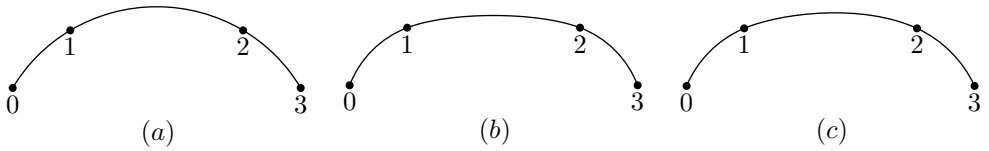


Рис. 10: Результаты `draw z0..z1..tension` α `и` β `..z2..z3` для разных α и β: (a) $\alpha = \beta = 1$; (b) $\alpha = \beta = 1.3$; (c) $\alpha = 1.5$, $\beta = 1$.

Пути MetaPost имеют также параметр, называемый "изгиб", который влияет на концы путей. При отсутствии спецификаций направления первый и последний отрезки нециклического пути — это приблизительно дуги окружности, как в случае $c = 1$ на рис. 11. Для использования другого значения для параметра изгиба укажите `{curl c}` для некоторого значения c. Таким образом,

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

установит параметр изгиба для `z0` и `z2`. Маленькие значения параметра изгиба уменьшают кривизну в указанных концовых точках пути, а большие значения увеличивают кривизну как показано на рис. 11. В частности, значение изгиба ноль делает кривизну нулевой.

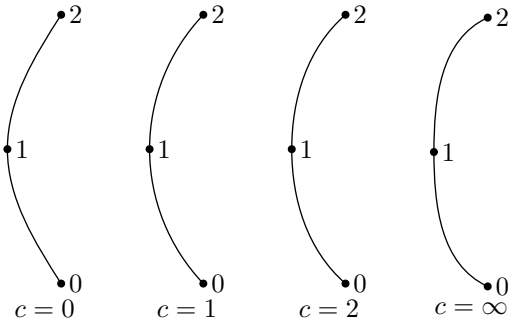


Рис. 11: Результаты `draw z0{curl c}..z1..{curl c}z2` для разных значений параметра изгиба c.

4.3 Полный синтаксис пути

Есть еще несколько других свойств синтаксиса пути MetaPost, но они относительно неважны. Из-за того, что METAFONT использует такой же синтаксис пути, заинтересованным читателям стоит посмотреть [4, раздел 14]. Сводка синтаксиса пути на рис. 12 включает все, обсуждаемое до сих пор, включая конструкции `--` и `...`, которые [4] показывает как макросы, а не примитивы. Несколько комментариев по семантике приведены здесь: если непустой ⟨указатель направления⟩ стоит перед ⟨узлом пути⟩, но не после, или наоборот, то указанное направление (или величина изгиба) прилагается как к входящим, так и к выходящим отрезкам пути. Похожее соглашение применяется, когда спецификатор ⟨управления⟩ дает только одну ⟨первичную пару⟩. Таким образом,

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

эквивалентно

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

⟨выражение-путь⟩ → ⟨подвыражение-путь⟩
| ⟨подвыражение-путь⟩⟨указатель направления⟩
| ⟨подвыражение-путь⟩⟨соединитель пути⟩ `cycle`
⟨подвыражение-путь⟩ → ⟨узел пути⟩
| ⟨выражение-путь⟩⟨соединитель пути⟩⟨узел пути⟩
⟨соединитель пути⟩ → `--`
| ⟨указатель направления⟩⟨базовый соединитель пути⟩⟨указатель направления⟩
⟨указатель направления⟩ → ⟨пусто⟩
| `{curl` ⟨числовое выражение⟩`}`
| `{`⟨выражение-пара⟩`}`
| `{`⟨числовое выражение⟩`,`⟨числовое выражение⟩`}`
⟨базовый соединитель пути⟩ → `..` | `...` | `..`⟨напряжение⟩`..` | `..`⟨управление⟩`..`
⟨напряжение⟩ → `tension`⟨числовая первичность⟩
| `tension`⟨числовая первичность⟩`and`⟨числовая первичность⟩
⟨управление⟩ → `controls`⟨первичная пара⟩
| `controls`⟨первичная пара⟩`and`⟨первичная пара⟩

Рис. 12: Синтаксис конструкции пути

Пара координат, подобная `(30,20)` или переменной `z`, представляющей координатную пару, — это то, что на рис. 12 зовется ⟨первичной парой⟩. Похожим является ⟨узел пути⟩ за исклю-

чением того, что он может приобретать другие формы, такие как выражение пути в скобках. Первичности и выражения различных типов будут обсуждаться в полном объеме в разделе 6.

5 Линейные уравнения

Важным свойством, взятым из METAFONT, является возможность решать линейные уравнения, вследствие этого программы могут писаться в частично декларативной манере. Например, MetaPost-интерпретатор может считать

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

и вывести, что $a = 2$ и $b = 1$. Эти же выражения могут быть записаны слегка более компактно путем соединения их вместе несколькими знаками равенства:

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

Каким бы способом вы не задавали уравнения, вы можете затем дать команду

```
show a,b;
```

для просмотра значений `a` и `b`. MetaPost ответит, напечатав

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

Заметьте, что `=` не является оператором присваивания; он просто объявляет, что левая часть равна правой. Таким образом, `a=a+1` приведет к сообщению об ошибке, жалующемуся на "противоречивое уравнение". Способ увеличения значения `a` — в использовании оператора присваивания `:=` как в примере:

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

Другими словами, `:=` для изменения существующих значений, а `=` для задания линейных уравнений для решения.

Нет ограничений на смешивание уравнений и операций присваивания, например,

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

После первых двух уравнений, устанавливающих `a` и `b` равными 2, операция присваивания изменит `a` на 3 без влияния на `b`. Окончательное значение `c` — 3, т. к. оно приравняется новому значению `a`. В общем, операция присваивания интерпретируется сначала вычислением нового значения и затем уничтожением старого значения из всех существующих уравнений перед собственно присваиванием.

5.1 Уравнения и координатные пары

MetaPost может также решать линейные уравнения, содержащие координатные пары. Мы уже видели много тривиальных примеров этого в форме уравнений, подобных

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

Каждая сторона уравнения должна быть сформирована сложением или вычитанием координатных пар и умножением или делением их на известные числовые количества. Другие способы именования пар значений переменных будут обсуждаться позже, а пока рассмотрим именование вида `z`⟨число⟩, которое весьма удобно, потому что оно — сокращение для

`(x`⟨число⟩`, y`⟨число⟩`)`

Это делает возможным давать значения переменным z заданием уравнений для их координат. Например, точки z1, z2, z3, и z6 на рис. 13 были инициализированы следующими уравнениями:

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

В точности те же самые точки могли быть получены прямой установкой их значений:

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

После чтения уравнений MetaPost-интерпретатор знает значения z1, z2, z3 и z6. Следующий шаг в конструировании рис. 13 — это определение точек z4 и z5, одинаково удаленных от z3 и z6 и лежащих на одной линии с ними. Потому как эта операция появляется часто, MetaPost имеет для нее специальный синтаксис. Усредняющая конструкция

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

означает, что z4 прошла $\frac{1}{3}$ пути от $z3$ до $z6$, т. е.

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

Схожая конструкция

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

устанавливает z5 на $\frac{2}{3}$ пути от $z3$ до $z6$.

```
beginfig(13);
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;
```

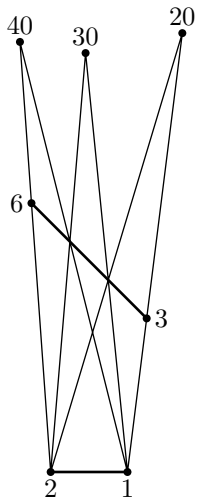


Рис. 13: Команды MetaPost и рисунок-результат. Ярлыки точек добавлены к рисунку для ясности.

Усреднение может быть также использовано для того, чтобы сказать, что некоторая точка находится в неизвестной позиции на прямой между двумя известными точками. Например, мы можем ввести новую переменную aa и записать что-то вроде

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

Это означает, что z20 — это неизвестное отношение aa пути по прямой между z1 и z3. Еще одного такого отношения для другой линии достаточно для фиксации значения z20. Описание

того, что z20 пересечение прямых z1-z3 и z2-z4 вводит еще одну переменную ab и устанавливается

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

Это позволяет MetaPost найти x20, y20, aa и ab.

Несколько сложновато постоянно думать о новых именах, подобных aa и ab. Этого можно избежать, используя специальную возможность, называемую whatever. Этот макрос генерирует новую анонимную переменную каждый раз, когда он появляется. Таким образом, команда

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

устанавливает z20 как и раньше, но она использует whatever для генерации двух *различных* анонимных переменных вместо aa и ab. Рис. 13 показывает как устанавливаются z20, z30 и z40.

5.2 Работа с неизвестными

Уравнения в системе такой, как на рис. 13, могут быть заданы в любом порядке, но все уравнения должны быть линейными и все переменные должно быть возможным вычислить тогда, когда они понадобятся. Это значит, что уравнения

```
z1=-z2=(.2in,0);
x3=-x6=.3in;
x3+y3=x6+y6=1.1in;
z4=1/3[z3,z6];
z5=2/3[z3,z6];
```

достаточны для определения z1 и остальных до z6 и порядок уравнений не важен. С другой стороны

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

будет правильно только в случае, когда известное значение было предварительно указано для разности z3 − z1, потому что это уравнение эквивалентно

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

и требования линейности не позволяют умножать неизвестные компоненты z3 − z1 на анонимный неизвестный результат whatever. Общее правило в том, что вы не можете умножать два неизвестных количества, делить на неизвестное количество или использовать неизвестное количество в команде draw. Из-за того, что разрешены только линейные уравнения, MetaPost-интерпретатор может легко решать уравнения и хранить информацию о том, какие величины известны.

Наиболее естественный способ гарантировать, что MetaPost сможет воспринять выражение типа

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

в гарантии, что z1 и z3 известны. Однако этого в действительности не требуется, т. к. MetaPost сможет вывести значение для z3 − z1, не зная предварительно z1 и z3. Например, MetaPost воспринимает правильными уравнения

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

и при этом не будет способен определить любую из компонент z1, z3 или z20.

Эти уравнения дают частичную информацию о z1, z3 и z20. Хороший способ понять это в рассмотрении другого уравнения

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

Оно произведет сообщение об ошибке "`! Redundant equation`"[8]. MetaPost считает, что вы пытаетесь сообщать ему что-то новое и поэтому он обычно предупреждает при задании избыточного уравнения. Новое уравнение вида

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

произведет сообщение об ошибке[9]

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

Это сообщение об ошибке иллюстрирует ошибку округления в механизме MetaPost для решения линейных уравнений. Ошибка округления — это обычно несерьезная проблема, но она может вызвать затруднение при попытке найти пересечение двух почти параллельных прямых.

6 Выражения

Настало время для более систематического обзора языка MetaPost. Мы видели числовые количества и координатные пары и то, что их можно соединять для указания пути для команд `draw`. Мы также видели, как переменные могут быть использованы в линейных уравнениях, но не обсуждали всех операций и типов данных, что могут быть использованы в уравнениях.

Использование команды

`show` ⟨выражение⟩

для печати символьного представления значения любого выражения делает возможным эксперименты с выражениями любых типов данных, встречающихся далее. Известные числовые значения печатаются в отдельных строках, предваряемые "`>>` ". Другие типы результата расчета печатаются похожим образом, за исключением того, что сложные значения иногда не распечатываются на устройстве вывода. Последнее производит ссылку на файл-дубликат, которая выглядит примерно так[10]

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

Если вы захотите распечатки на терминале полных результатов команды `show`, то назначьте положительное значение внутренней переменной `tracingonline`.

6.1 Типы данных

MetaPost в действительности имеет десять типов данных: числовой, для пар, для путей, трансформации, цвета (rgb-цвета), cmyk-цвета, строковый, логический, для картинок и тип пера. Давайте рассматривать их по-одному, начиная с числового типа.

Числовые (numeric) количества представляются в MetaPost с фиксированной десятичной точкой как целые, умноженные на $\frac{1}{65536}$. Они должны обычно иметь модуль, меньший 4096, но промежуточные результаты могут быть в восемь раз больше. Это не должно быть проблемой для расстояний или значений координат, т. к. 4096 PostScript-пунктов составляют более 1.4 метра. Если вам нужно работать с числами размера 4096 и более, то установка внутренней

[8] Избыточное уравнение

[9] Противоречивое уравнение (отклонение на 71.99979).

[10] картинка (см. файл-дубликат)

переменной `warningcheck` в ноль подавит предупреждающие сообщения о больших числовых количествах.

Тип pair (пары) представляется как пара числовых количеств. Мы видели, что пары используются для задания координат в команде `draw`. Пары можно складывать, вычитать, использовать в выражениях усреднения, умножать и делить на числа.

Тип путей (path) уже обсуждался в контексте команды `draw`, но это обсуждение обошло стороной то, что пути важные отдельные объекты, которые можно сохранять и изменять. Путь представляет прямую линию или кривую, определяемые параметрически.

Другой тип данных представляет произвольную аффинную трансформацию (transform). *Трансформация* может быть любой комбинацией вращений, масштабирований, наклонов и сдвигов. Если `p` $= (p_x, p_y)$ — это пара и `T` — это трансформация, то

```
p transformed T
```

— это пара вида

$$(t_x + t_{xx}p_x + t_{xy}p_y, t_y + t_{yx}p_x + t_{yy}p_y),$$

где 6 числовых количеств $(t_x, t_y, t_{xx}, t_{xy}, t_{yx}, t_{yy})$ определяют `T`. Трансформации могут быть применимы к путям, рисункам, перьям и трансформациям.

Тип цвета (color) подобен типу пары, но он имеет три компоненты вместо двух и каждая компонента обычно находится в диапазоне от 0 до 1. Подобно парам, цвета могут складываться, вычитаться, использоваться в выражения усреднения, умножаться и делиться на числа. Цвета могут задаваться при помощи констант `black`, `white`, `red`, `green`, `blue` или явно заданными красной, зеленой и синей компонентами. Черный — это `(0,0,0)` и белый — это `(1,1,1)`. Уровень серого, такой как `(.4,.4,.4)`, можно также задать как `0.4white`. Хотя цветовой переменной может быть любая упорядоченная тройка, при добавлении объекта к картинке MetaPost преобразует ее цвета обрезкой каждой цветовой компоненты до диапазона от 0 до 1. Например, MetaPost будет выводить цвет (1,2,3) как (1,1,1). MetaPost решает линейные уравнения с цветами таким же образом как и с парами. Тип 'rgbcolor' (rgb-цвет) — это синоним типа 'color' (цвет).

Тип cmykcolor (cmyk-цвета) подобен типу color, но имеет четыре компоненты вместо трех. Этот тип используется для задания цветов их зеленоголубой (cyan), пурпурнокрасной (magenta), желтой и черной компонентами. Из-за того, что cmyk-цвет использует краски вместо световых лучей, белый цвет будет выражаться как `(0,0,0,0)` и черный как `(0,0,0,1)`. Теоретически цвета `(`c`,`m`,`y`,1)` и `(1,1,1,`k`)` должны давать черный для любых значений c, m, y и k. На практике этого избегают, т. к. это тратит цветные чернила и может приводить к неудовлетворительным результатам.

Строки (string) представляют последовательность символов. Строковые константы задаются в двойных кавычках `"подобно этой"`. Строковые константы не могут содержать двойных кавычек или переходов на новую строку, но есть способ конструировать строки, содержащие любую последовательность восьмибитных знаков.

Преобразование строк в другие типы, обычно числовые, возможно примитивом `scantokens`:

```
n := scantokens(строка);
```

Более абстрактно, `scantokens` разбирает строку на последовательность знаков, также как MetaPost считывал бы их при вводе.

Логический тип (boolean) имеет константы `true` и `false` и операторы `and`, `or`, `not`. Отношения `=` и `<>` проверяют объекты любых типов на равенство и неравенство. Отношения сравнения `<`, `<=`, `>` и `>=` определяются словарно для строк и обычным способом для чисел. Отношения порядка определены также и для логических величин, пар, цветов и трансформаций, но правила их сравнения не стоит обсуждать здесь.

Тип данных picture (рисунок) — это в точности то, что подразумевается его именем. Все, что можно нарисовать в MetaPost, можно сохранить в переменной-картинке. Фактически команда `draw` сохраняет свои результаты в особой переменной-картинке, называемой `currentpicture`. Картинки могут быть добавлены к другим картинкам и их можно трансформировать.

Наконец, тип данных, называемый pen (перо). Главная функция перьев в MetaPost в определении толщины линии, но их также можно использовать для достижения каллиграфических эффектов. Команда

`pickup` ⟨выражение-перо⟩

обусловит использование заданного пера в последующих командах `draw`.

Обычно выражение-перо имеет форму

`pencircle scaled` ⟨числовая первичность⟩.

Оно определяет круговое перо, производящее линии постоянной толщины. Если желательны каллиграфические эффекты, то выражение-перо может быть приспособлено для задания эллиптического или многоугольного пера.

6.2 Операторы

Есть много разных способов сделать выражения десяти базовых типов, но большинство операций можно сопоставить сравнительно простому синтаксису с четырьмя уровнями приоритета как показано на рис. 14. Есть первичности, вторичности, третичности и выражения каждого из базовых типов, поэтому синтаксические правила могут уточняться для работы с такими сущностями как ⟨числовая первичность⟩, ⟨логическая третичность⟩ и т. д. Это позволяет типу результата операции зависеть от выбора оператора и типов операндов. Например, отношение `<` — это ⟨третичная бинарность⟩, которую можно применить к ⟨числовому выражению⟩ и к ⟨числовой третичности⟩ для получения ⟨логического выражения⟩. Этот же оператор может допускать другие типы операндов такие как ⟨строковое выражение⟩ и ⟨строковая третичность⟩, но результатом в случае несовпадения типов операндов будет сообщение об ошибке.

⟨первичность⟩ → ⟨переменная⟩
| `(`⟨выражение⟩`)`
| ⟨оператор 0-уровня⟩
| ⟨of-оператор⟩⟨выражение⟩`of`⟨первичность⟩
| ⟨унарный оператор⟩⟨первичность⟩

⟨вторичность⟩ → ⟨первичность⟩
| ⟨вторичность⟩⟨первичный бин. оп-р⟩⟨первичность⟩

⟨третичность⟩ → ⟨вторичность⟩
| ⟨третичность⟩⟨вторичный бин. оп-р⟩⟨вторичность⟩

⟨выражение⟩ → ⟨третичность⟩
| ⟨выражение⟩⟨третичный бин. оп-р⟩⟨третичность⟩

Рис. 14: Общие синтаксические правила для выражений

Операторы умножения и деления, `*` и `/`, — примеры того, что на рис. 14 зовется ⟨первичным бинарным оператором⟩. Каждый из них может допускать два числовых операнда или один числовой операнд и один типа пара или цвет. Оператор возведения в степень `**` — это ⟨первичный бинарный оператор⟩, который требует два числовых операнда. Размещение его на том же уровне приоритета, что и умножение и деление, имеет неприятное последствие в том, что `3*a**2` значит $(3a)^2$, а не $3(a^2)$. Из-за того, что унарный минус относится к первичному уровню, он также приводит к неудобочитаемости типа `-a**2`, означающей $(-a)^2$. К счастью, вычитание имеет меньший приоритет и `a-b**2` означает $a-(b^2)$ вместо $(a-b)^2$

Другим ⟨первичным бинарным оператором⟩ является оператор `dotprod`, вычисляющий скалярное произведение двух пар. Например, `z1 dotprod z2` эквивалентно `x1*x2 + y1*y2`.

Аддитивные операторы `-` и `+` — ⟨вторичные бинарные операторы⟩, применимые к числам, парам или цветам и производящие результаты того же типа. Другие операторы, что попадают в эту категорию — это "Пифагорово сложение" `++` и "Пифагорово вычитание" `+-+`: `a++b` значит $\sqrt{a^2+b^2}$ и `a+-+b` значит $\sqrt{a^2-b^2}$. Есть еще слишком много других операторов для перечисления здесь, но одни из самых важных — это логические операторы `and` и `or`. Оператор `and` — это ⟨первичный бинарный оператор⟩ и оператор `or` — это ⟨вторичный бинарный оператор⟩.

Базовые операции со строками — это склейка, выделение подстроки и вычисление длины строки. Склейка реализуется ⟨третичным бинарным оператором⟩ `&`, например,

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

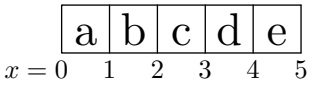
производит строку `"abcde"`. Оператор `length` возвращает число символов в строке, если аргументом является ⟨строковая первичность⟩, например,

```
length "abcde"
```

возвращает `5`. Другое применение оператора `length` обсуждается на стр. 38. Для выделения подстроки ⟨of-оператор⟩ `substring` используется таким образом:

`substring` ⟨выражение-пара⟩ `of` ⟨строковая первичность⟩

Часть строки для выделение определяется ⟨выражением-парой⟩. Позиции в строке нумеруются так, что целые позиции попадают *между* символами. Представим строку, написанную на кусочке бумаге в клетку так, что первый символ займет x-координаты между нулем и единицей, а следующий символ покроет координаты в диапазоне $1 \leqslant x \leqslant 2$, и т. д. Поэтому строку `"abcde"` следует представлять в виде



и `substring (2,4) of "abcde"` будет `"cd"`. Это выглядит несколько усложнено, но имеет целью избежать надоедающих ошибок "на единицу".

Некоторые операторы не берут аргументов вообще. Пример того, что на рис. 14 зовется ⟨оператором 0-уровня⟩, — это `nullpicture`, который возвращает совершенно пустую картинку.

Базовый синтаксис на рис. 14 покрывает только те аспекты синтаксиса выражений, которые являются независимыми от типа. Например, непростой синтаксис пути на рис. 12, дает альтернативные правила для конструирования ⟨выражения-пути⟩. Дополнительное правило

⟨узел пути⟩ → ⟨третичная пара⟩ | ⟨третичный путь⟩

объясняет значение ⟨узла пути⟩ на рис. 12. Таким образом, выражение-путь

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

не нуждается в скобках вокруг `z1+(1,1)`.

6.3 Дроби, усреднения и унарные операторы

Выражения усреднения отсутствуют в синтаксисе базового выражения на рис. 14. Выражения усреднения разбираются на ⟨первичном⟩ уровне, так что общее правило для их конструирования следующее

⟨первичность⟩ → ⟨числовой атом⟩`[`⟨выражение⟩`,`⟨выражение⟩`]`,

где каждое ⟨выражение⟩ может быть типа число, пара или цвет. ⟨Числовой атом⟩ в выражении усреднения имеет очень простой тип ⟨числовой первичности⟩, как показано на рис. 15. Значением всего этого является то, что первый параметр в выражении усреднения требует заключения в скобки, если он в точности не переменная, не положительное число или не положительная дробь. Например,

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

очень различны: первое — это $-b$, т. к. оно эквивалентно `-(1[a,b])`; второе — это $a-(b-a)$ или $2a-b$.

⟨числовая первичность⟩ → ⟨числовой атом⟩
| ⟨числовой атом⟩`[`⟨числовое выражение⟩`,`⟨числовое выражение⟩`]`
| ⟨of-оператор⟩⟨выражение⟩`of`⟨первичность⟩
| ⟨унарный оператор⟩⟨первичность⟩
⟨числовой атом⟩ → ⟨числовая переменная⟩
| ⟨число или дробь⟩
| `(`⟨числовое выражение⟩`)`
| ⟨числовой оператор 0-уровня⟩
⟨число или дробь⟩ → ⟨число⟩`/`⟨число⟩
| ⟨число, за которым нет '`/`⟨числа⟩'⟩

Рис. 15: Синтаксические правила для числовых первичностей

Заметным свойством синтаксических правил на рис. 15 является то, что оператор `/` связывает более крепко, когда его операнды являются числами. Таким образом, `2/3` — это ⟨числовой атом⟩, а `(1+1)/3` — это только ⟨числовая вторичность⟩. Применение ⟨унарного оператора⟩, такого как `sqrt`, делает разницу очевидной:

```
sqrt 2/3
```

значит $\sqrt{\frac{2}{3}}$, а

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

значит $\sqrt{2}/3$. Операторы, такие как `sqrt`, могут быть записаны в стандартной функциональной нотации, но часто нет нужды брать аргумент в скобки. Это верно для любой функции, что разбирается как ⟨унарная операция⟩. Например, и `abs(x)`, и `abs x` вычисляют модуль `x`. Это же верно для функций `round`, `floor`, `ceiling`, `sind` и `cosd`. Две последние из них вычисляют тригонометрические функции от угла в градусах.

Не все унарные операторы берут числовые аргументы и возвращают числовые результаты. Например, оператор `abs` можно применять к паре для вычисления длины вектора. Применение оператора `unitvector` к паре производит опять пару, задающую вектор с тем же направлением и длиной 1. Оператор `decimal` берет число и возвращает его строковое представление. Оператор `angle` берет пару и вычисляет арктангенс отношения ее компонент, т. е. `angle` — это оператор, обратный `dir`, что обсуждается в разделе 4.2. Есть также оператор `cycle`, что берет ⟨первичный путь⟩ и возвращает логический результат, показывающий является ли этот путь замкнутой кривой.

Существует целый класс других операторов для классификации выражений с логическим результатом. Имя типа, такое как `pair`, может применяться к любому типу ⟨первичности⟩ и возвращать логический результат, показывающий является ли аргумент парой. Аналогично, каждое имя из следующих далее можно использовать как унарный оператор: `numeric`, `boolean`, `cmykcolor`, `color`, `string`, `transform`, `path`, `pen`, `picture` и `rgbcolor`. Кроме проверки типа ⟨первичности⟩, вы можете использовать операторы `known` и `unknown` для проверки, имеет ли она конкретное значение.

Даже числа могут вести себя как оператор в некоторых контекстах. Это ссылка на трюк, что позволяет `3x` и `3cm` как альтернативы для `3*x` и `3*cm`. Правило в том, что ⟨число или дробь⟩, за которыми нет `+`, `-` или другого ⟨числа или дроби⟩, может служить как ⟨первичный бинарный оператор⟩. Таким образом, `2/3x` — это две трети от `x`, но `(2)/3x` — это $\frac{2}{3x}$, а `3 3` — это ошибка.

Есть также операторы для извлечения числовых полей из пар, цветов, cmyk-цветов и даже трансформаций. Если `p` — это ⟨первичная пара⟩, то `xpart p` и `ypart p` извлекают ее компоненты так, что

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

эквивалентно `p`, даже если `p` — неизвестная пара, используемая в линейном уравнении. Аналогично, цвет `c` эквивалентен

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

Для cmyk-цвета `c` его эквивалент

```
(cyanpart c, magentapart c, yellowpart c, blackpart c),
```

а для оттенка серого `c` есть только один компонент

```
greypart c.
```

Все операторы компонент цвета обсуждаются более подробно в разделе 9.10. Спецификаторы частей трансформаций обсуждаются в разделе 9.3.

7 Переменные

MetaPost позволяет составные имена переменных, такие как `z.a`, `x2r`, `y2r` и `z2r`, где `z2r` означает `(x2r,y2r)`, а `z.a` — `(x.a,y.a)`. Фактически существует широкий класс суффиксов, например, `z`⟨суффикс⟩, означающий

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

Из-за того, что ⟨суффикс⟩ составляется из знаков, будет наилучшим начать с нескольких слов о знаках.

7.1 Знаки

Входной файл MetaPost рассматривается как последовательность чисел, строковых констант и символьных знаков. Число состоит из последовательности цифр и может содержать десятичную точку. Технически знак минус вначале отрицательного числа — это отдельный знак. Из-за того, что MetaPost использует арифметику с фиксированной точкой, он не понимает экспоненциальной нотации, такой как `6.02E23`. MetaPost будет интерпретировать это как число 6.02, за которым следует символьный знак `E`, за которым идет число 23.

Все между парой двойных кавычек, `"`, является строковой константой. Строковой константе нельзя начинаться на одной строке и заканчиваться на другой. Строковая константа не может также содержать двойных кавычек, `"`, и чего-нибудь еще, отличного от печатных символов ASCII.

Все в строке ввода, отличное от чисел и символьных констант, разбивается на символьные знаки. Символьный знак — это последовательность одного или более схожих символов, где символы “схожи”, если они встречаются на одной строке таблицы 2.

Таким образом, `A_alpha` и `+-+` — это отдельные символические знаки, `!=` интерпретируется как два знака, а `x34` — это символический знак, за которым следует число. Вследствие того, что

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

Таблица 2: Классы символов для разбиения на знаки

квадратные скобки приведены на отдельных строках, символическими знаками, включающими их, являются только `[`, `[[`, `[[[`, ... и `]`, `]]`, ...

Некоторые символы не приведены в таблице 2, потому что они требуют специального обращения. Четыре символа `,;()` являются “одиночками”: запятая, точка с запятой или скобка — это отдельный знак, даже если одинаковые из них идут подряд. Таким образом, `(())` — это четыре знака, а не один или два. Знак процента является весьма специальным, потому что он вводит комментарии. Знак процента и все после него до конца строки игнорируется.

Другим специальным символом является точка. Две и более точек вместе формируют символьный знак, но отдельная точка игнорируется, а точка, перед которой или за которой идут цифры, является частью числа. Таким образом, `..` и `...` — это символьные знаки, а `a.b` — это просто два знака `a` и `b`. Принято использовать точку таким образом для разделения знаков, когда имя переменной имеет длину более одного знака.

7.2 Декларации переменных

Имя переменной — это либо символьный знак, либо последовательность символьных знаков. Большинство символьных знаков являются правильными именами переменных, но что угодно с предопределенным значением подобно `draw`, `+` или `..` недопустимо, например, имена переменных не могут быть макросами или примитивами MetaPost. Это второстепенное ограничение допускает широкий класс имен переменных: `alpha`, `==>`, `@&#$&` и `~~` — все они легитимные имена переменных. Символьные знаки без специального значения называются этикетками (*tags*).

Имя переменной может быть последовательностью этикеток, подобной `f.bot` или `f.top`. Эта идея служит для создания некоторых возможностей записей Паскаля или структур Си. Также возможно симулировать массивы использованием имен переменных, содержащих числа и символьные знаки. Например, имя переменной `x2r` состоит из этикетки `x`, числа 2 и этикетки `r`. Могут быть также переменные, именованные `x3r` и даже `x3.14r`. Эти переменные можно рассматривать как массив через конструкции, подобные `x[i]r`, где `i` имеет подходящее числовое значение. Суммарный обзор синтаксиса для имен переменных показан на рис. 16.

⟨переменная⟩ → ⟨этикетка⟩⟨суффикс⟩
⟨суффикс⟩ → ⟨пусто⟩ | ⟨суффикс⟩⟨индекс⟩ | ⟨суффикс⟩⟨этикетка⟩
⟨индекс⟩ → ⟨число⟩ | `[`⟨числовое выражение⟩`]`

Рис. 16: Синтаксис имен переменных.

Переменные, подобные `x2` и `y2`, обычно имеют числовое значение, поэтому мы можем ис-

пользовать факт, что `z`⟨суффикс⟩ — это сокращение для

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

для генерации пар-значений, когда нужно. С другой стороны, макрос `beginfig` уничтожает все существующие до его исполнения переменные, начинающиеся с этикеток `x` или `y`, так что блоки `beginfig` ... `endfig` не взаимодействуют друг с другом при использовании такой схемы именования. Другими словами, переменные, начинающиеся с `x`, `y`, `z`, локальны в той картинке, где они используются. Общий механизм для создания локальных переменных будет обсуждаться в разделе 10.1.

Объявления типа делает возможным использование почти любой схемы именования при удалении всех предшествующих значений, что могут вызвать взаимодействие. Например, декларация

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

делает `pp` и `a.b` неизвестными парами. Такая декларация не является строго локальной, т. к. `pp` и `a.b` не восстанавливают автоматически свои предшествующие значения в конце текущего рисунка. Они опять становятся неизвестными парами при повторении этой декларации.

Декларации работают одинаковым образом для любого другого из оставшихся девяти типов: числового, путевого, трансформационного, цветового, cmyk-цветового, строкового, логического, рисуночного и перьевого. Единственное ограничение в том, что вы не можете задать точный числовой индекс в декларации переменной. Не пишите ошибочных деклараций типа

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

используйте обобщенный символ индекса `[]` вместо чисел для объявления всего массива:

```
numeric q[];
```

Вы можете также определить “многоразмерные” массивы. После декларации

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

`p2q3` и `pq1.4 5` — это два пути.

Внутренние переменные, подобные `tracingonline`, не могут быть объявлены нормальным образом. Все внутренние переменные, обсуждаемые в этом руководстве имеют значения изначально и никак не могут быть декларированы снова, но есть способ объявить, что новая переменная должна вести себя подобно внутренней. Это декларация `newinternal`, за которой следует список символических знаков. Например,

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

обусловит поведение `a`, `b` и `c`, как и у внутренних переменных. Такие переменные всегда имеют известные числовые значения и эти значения могут быть изменены только использованием оператора присваивания `:=`. Внутренние переменные инициализируются нулем и, кроме того, макропакет Plain дает некоторым из них ненулевые значения. (Макросы Plain обычно загружаются автоматически в начале работы как описано в разделе 1.)

8 Интеграция текста и графики

MetaPost имеет несколько возможностей для включения меток и прочего текста в генерируемые им рисунки. Простейший способ сделать это в использовании команды `label`

`label`⟨суффикс метки⟩`(`⟨выражение-строка или картинка⟩`,` ⟨выражение-пара⟩`);`

⟨Выражение-строка или картинка⟩ задает метку, а ⟨выражение-пара⟩ позицию для нее. ⟨Суффикс метки⟩ может быть ⟨пустым⟩, что будет означать центрировать метку на заданных координатах. Если вы размечаете некоторые участки диаграммы, то вам вероятно понадобится слегка сместить метку, чтобы избежать накладки. Это иллюстрируется на рис. 17, где метка `"a"` размещается над серединой указываемой линии, а метка `"b"` — слева от середины своей линии. Это достигается использованием `label.top` для метки `"a"` и `label.lft` для метки `"b"`, как показано на рисунке. ⟨Суффикс метки⟩ указывает позицию метки относительно заданных координат. Полное множество возможностей — это

⟨суффикс метки⟩ → ⟨пусто⟩ | `lft` | `rt` | `top` | `bot` | `ulft` | `urt` | `llft` | `lrt`,

где `lft` и `rt` означают влево и вправо, а `llft`, `ulft` и т. п. значат вниз и влево, вверх и влево и т. п. Действительное расстояние, на которое будет смещена метка в заданном направлении, определяется внутренней переменной `labeloffset`.

```
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;
```

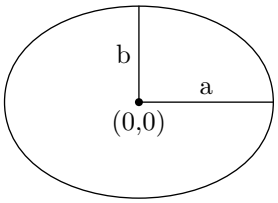


Рис. 17: Код MetaPost и результат вывода

Рис.17 также иллюстрирует команду `dotlabel`. Она в точности такая же как команда `label`, за которой следует команда рисования точки в заданных координатах. Например,

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

помещает точку в `z0` и затем размещает метку “(0,0)” точно под точкой.

Другой альтернативой является макрос `thelabel`. Он имеет такой же синтаксис, что и команды `label` и `dotlabel`, но он возвращает результат как ⟨первичный рисунок⟩ вместо его действительного изображения. Таким образом,

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

эквивалентно

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

Для простых случаев размеченных рисунков, вам может обычно быть достаточно `label` и `dotlabel`. Дополнительно вы можете использовать короткую форму команды `dotlabel`, что сэкономит много времени, когда вы имеете много точек `z0`, `z1`, `z.a`, `z.b`, и т. п. и хотите использовать суффиксы `z` как метки. Команда

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

эквивалентна

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

Таким образом, аргумент `dotlabels` — это список суффиксов для переменных `z`, а ⟨суффикс метки⟩, задаваемый с `dotlabels`, используется для позиционирования всех меток.

Есть еще команда `labels`, аналогичная `dotlabels`, но ее использование не рекомендуется, т. к. она создает проблемы совместимости с METAFONT. Некоторые версии стандартного макропакета Plain определяют `labels` как синоним `dotlabels`.

Для команд разметки, таких как `label` и `dotlabel`, использующих строковые выражения для текста меток, строки печатаются в стандартном шрифте, определяемом строковой переменной `defaultfont`. Начальное значение `defaultfont` — это обычно `"cmr10"`, но оно может быть изменено на другое имя шрифта присваиванием, например,

```
defaultfont:="ptmr8r",
```

`ptmr8r` — это типичный способ сослаться на шрифт Times-Roman в TEX.

Есть еще числовое количество, называемое `defaultscale`, определяющее размер шрифта. Пока `defaultscale` равно 1, вы получаете "нормальный размер", который обычно равен 10 пунктам, но это можно изменить. Например,

```
defaultscale := 1.2
```

делает метки на двадцать процентов больше. Если вам неизвестен нормальный размер и вы хотите быть уверенными в конкретном размере шрифта, скажем 12 пунктов, вы можете использовать оператор `fontsize` для определения нормального размера, например,

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

Когда вы меняете `defaultfont`, то имя нового шрифта должно быть чем-то, что TEX сможет понять, т. к. MetaPost получает информацию о высоте и ширине чтением `tfm`-файла. (Это объясняется в *The TEXbook* [5].) Должно быть возможно использовать встроенные шрифты PostScript, но их имена зависят от системы. Некоторые типичные имена шрифтов — это `ptmr8r` для Times-Roman, `pplr8r` для Palatino и `phvr` для Helvetica. Документ Fontname, доступный в http://tug.org/fontname, содержит много информации об именах шрифтов и TEX. TEX-шрифт, такой как `cmr10`, является немного опасным, потому что он не имеет символа пробел и некоторых других символов ASCII.

MetaPost не использует информацию о лигатурах и кернингах, что содержится в шрифтах TEX. Более того, сам MetaPost не может интерпретировать виртуальные шрифты.

8.1 Набор ваших меток

TEX может быть использован для форматирования сложных меток. Если вы напишите

`btex` ⟨команды печати⟩ `etex`

во входном файле MetaPost, то ⟨команды печати⟩ будут обработаны TEX и транслированы в выражение-картинку, точнее в ⟨первичный рисунок⟩, что сможет использоваться в команде `label` или `dotlabel`. Пробелы после `btex` или перед `etex` игнорируются. Например, команда

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

на рис. 18 поместит метку $\sqrt{x}$ снизу и вправо от точки `(3,sqrt 3)*u`.

Рис. 19 иллюстрирует некоторые более сложные вещи, что можно сделать с метками. Вследствие того, что результатом `btex` ... `etex` является картинка, им можно оперировать как картинкой. В частности, к картинкам возможно применять трансформации. Мы пока не обсуждали синтаксис для этого, но ⟨рисунок-вторичность⟩ может быть

⟨рисунок-вторичность⟩ `rotated` ⟨числовая первичность⟩

```
beginfig(18);
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;
```

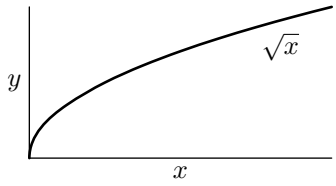


Рис. 18: Произвольный TEX в качестве метки

```
beginfig(19);
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;
```

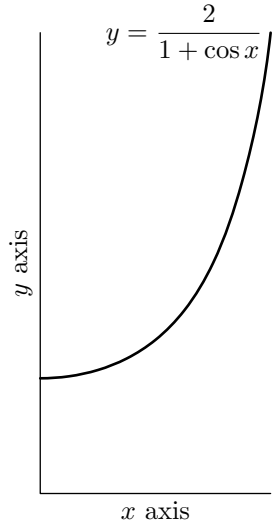


Рис. 19: Математические метки TEX и метки, вращаемые MetaPost

Это используется на рис. 19 для вращения метки "y axis" так, что она располагается повертикали.

Другой сложностью на рис. 19 является использование уравнения

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

как метки. Будет более естественно закодировать эту метку как

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

но это не сработает, потому что TEX набирает метки в "горизонтальном режиме".

Для печати *переменного* текста, как метки, используйте полезное средство `TEX`, описанное на стр. 74.

Далее о том, как TEX-текст транслируется в форму, понятную MetaPost: процессор MetaPost пропускает блок `btex` ... `etex`, полагаясь на препроцессор, который должен перевести этот блок в команды низкого уровня MetaPost. Если `fig.mp` — это главный файл, то транслированный TEX-текст помещается в файл с именем `fig.mpx`. Это обычно делается незаметно для пользователя, но вызывает ошибку, если один из блоков `btex` ... `etex` содержит ошибочную команду TEX. После ошибки, ошибочный текст TEX сохраняется в файле `mpxerr.tex` и сообщения об ошибках появляются в `mpxerr.log`.

Препроцессор для меток TEX *понимает* виртуальные шрифты, т. е. вы можете использовать команды вашего обычного TEX для переключения шрифтов внутри метки.

Определения макросов TEX или любые другие вспомогательные команды TEX могут заключаться в блок `verbatimtex` ... `etex`. Разница между `btex` и `verbatimtex` в том, что первый генерирует выражение-рисунок, а второй только добавляет данные для обработки TEX. Например, если вы хотите, используя TEX, напечатать метки, используя макросы из `mymac.tex`, то ваш входной файл для MetaPost будет выглядеть подобно чему-то такому:

```
verbatimtex \input mymac etex
beginfig(1);
    ...
label(btex ⟨TEX-текст, использующий mymac.tex⟩ etex, ⟨некоторые координаты⟩);
    ...
```

Для Unix[11] и других основанных на Web2C систем опция MetaPost `-troff` скажет препроцессору, что блоки `btex` ... `etex` и `verbatimtex` ... `etex` представлены в troff вместо TEX. Когда используется эта опция, MetaPost устанавливает внутреннюю переменную `troffmode` в 1.

Установка `prologues` может быть полезна также и с TEX, а не только для troff. Далее приводятся некоторые разъяснения:

- Когда `prologues` равно 0, что устанавливается по-умолчанию, выходные файлы MetaPost не содержат используемых шрифтов. Шрифты в результате-выводе будут вероятно Courier или Times-Roman.
- Когда `prologues` равно 1, вывод MetaPost объявляется "структурированным PostScript" (EPSF), но это не вполне верно. Этот вариант поддерживается для обратной совместимости со старыми troff-документами, но его использование как устаревшего не рекомендуется. Из исторических соображений, MetaPost устанавливает `prologues` в 1, когда опция `-troff` приводится в командной строке.

[11] Unix — это зарегистрированная торговая марка Unix Systems Laboratories.

- Когда `prologues` равно 2, вывод MetaPost — это EPSF, в котором предполагается, что текст набран PostScript-шрифтами, предоставляемыми "средой", такой как просмотрщик документа или встроенное приложение, использующие этот вывод. MetaPost будет пытаться установить кодировку шрифта правильно, основываясь на командах `fontmapfile` и `fontmapline`.
- Когда `prologues` равно 3, вывод MetaPost будет EPSF, содержащий шрифты PostScript (или подмножества шрифтов), используемые на основе команд `fontmapfile` и `fontmapline`. Это значение полезно для генерации самодостаточной PostScript-графики.

Стоит отметить, что стандартное значение `prologues:=0` достаточно для графики, включаемой в документы TEX. Переменная `prologues` также не нужна при обработке MetaPost-файлов через утилиту `mptopdf` (из дистрибутива ConTEXt), потому что PDF-файлы естественно самодостаточны. Более того, значение `prologues` не имеет эффекта на шрифты METAFONT в ваших MetaPost-файлах, т. е. MetaPost никогда не встраивает такие шрифты в свой вывод. Только драйверы вывода, например, `dvips` или pdfLATEX, могут встроить такие шрифты.

Детали того, как включать рисунки PostScript в документ, сделанный в TEX или troff, системо-зависимы. Они могут обычно быть найдены в страницах руководства (man pages) или в другой сетевой документации, но посмотрите сначала в раздел 3.2 этого руководства для кратких инструкций, которые во многих случаях окажутся достаточными. Руководство для широко используемой программы Dvips находится в файле `dvips.texi`, включенном в большинство стандартных дистрибутивов и доступном в сети в http://tug.org/texinfohtml/dvips.html и в других местах, а также в других форматах.

В системах, основанных на Web2C, препроцессор называется `makempx` — он вызывает программу `mpto`; документация по Web2C описывает их более подробно. Однако, упомянем здесь одно свойство: если переменная среды `MPTEXPRE` содержит имя существующего файла, то `makempx` будет помещать его в начало при выводе. Вы можете это использовать, например, для включения преамбул LATEX. Макрос `TEX`, описанный на стр. 74, обеспечивает другой способ такого включения.

8.2 Файлы-карты шрифтов

Если `prologues` установлено в 2, то любые используемые в выводе шрифты автоматически перекодируются согласно таблице, указанной в отдельной записи шрифтового файла-карты и включаемой в файл вывода. Если `prologues` установлено в 3, то MetaPost будет также пытаться включить используемые PostScript шрифты или их подмножества. Чтобы это работало, нужно получение информации из шрифтового файла-карты.

Код, основанный на шрифтовой библиотеке, используется pdfTEX. Следуя за pdfTEX, обнаруживаем два новых связанных с темой примитива: `fontmapfile` и `fontmapline`. Далее следует простой пример, указывающий файл-карту для шрифтов Latin Modern в кодировке YandY (LATEX LY1):

```
prologues:=2;
fontmapfile "texnansi-lm.map";
beginfig(1);
   draw "Helló, világ" infont "texnansi-lmr10";
endfig;
```

Используя `fontmapline`, можно указать информацию о шрифте внутри рисунка:

```
prologues:=2;
fontmapline "pplbo8r URWPalladioL-Bold  "&ditto&
            ".167 SlantFont"&ditto&" <8r.enc <uplb8a.pfb";
beginfig(1);
   draw "Hello, world" infont "pplbo8r";
endfig;
```

Это будет попытка перекодировать PostScript-шрифт URWPalladioL-Bold, чей tfm-файл — это pplbo8r.tfm. Кодировка определяется в файле 8r.enc и будет включатся в файл вывода.

Если этот же пример запускать с `prologues:=3`, то MetaPost будет включать подмножество шрифта, что расположено в uplb8a.pfb, в вывод. В этом случае подмножество шрифта перестраивается так, что оно будет верно закодировано на внутреннем уровне, поэтому 8r.enc не будет включаться.

Аргумент к обеим командам имеет символ опционального флага в самом начале. Этот опциональный флаг имеет то же самое значение как и в pdfTEX:

Опция	Значение
+	расширить список шрифта, игнорируя повторения
=	расширить список шрифта, замещая повторения
−	удалить все подходящие шрифты из списка шрифта

Без опций текущий список будет полностью замещен.

Если `prologues` установлен в два или три и команды `fontmapfile` отсутствуют, то MetaPost будет пытаться найти типовой файл-карту, начиная с `mpost.map`. Если это не приведет к успеху, то он будет также пробовать `troff.map` или `pdftex.map`, в зависимости от присутствия установки режима troff. Если `prologues` установлено в 1, то MetaPost пытается читать файл с именем `psfonts.map`, игнорируя любую команду `fontmapfile`. Повторим, это только для обратной совместимости.

8.3 Оператор infont

Как с TEX, так и с troff всю реальную работу по добавлению текста к картинке делает примитивный оператор MetaPost с именем `infont`. Он — ⟨первичный бинарный оператор⟩, берущий левым аргументом ⟨строковую вторичность⟩ и правым — ⟨строковую первичность⟩. Левый аргумент — это текст, а правый — имя шрифта. Результат операции — это ⟨рисунок-вторичность⟩, который можно трансформировать многими способами. Одна из возможностей — это увеличение в заданное число раз через синтаксис

⟨рисунок-вторичность⟩ `scaled` ⟨числовая первичность⟩

Таким образом, `label("text",z0)` эквивалентно

```
label("текст" infont defaultfont scaled defaultscale, z0)
```

Если использовать строковую константу для левого аргумента `infont` окажется неудобным, то можно использовать

`char` ⟨числовая первичность⟩

для выбора символа по его числовой позиции в шрифте. Таким образом,

```
char(n+64) infont "ptmr8r"
```

— это картинка, содержащая символ `n+64` шрифта `ptmr8r`, который обычно используется TEX для ссылки на Times-Roman. См. стр. 27 для дальнейшего обсуждения.

Сам MetaPost не перекодирует свой ввод, т. е. когда вы используете строку `infont` для меток (вместо `btex ... etex`), строка должна быть задана в кодировке шрифта.

8.4 Измерение текста

MetaPost делает доступными физические размеры картинок, генерируемых оператором `infont`. Унарные операторы `llcorner`, `lrcorner`, `urcorner`, `ulcorner` и `center` с аргументом ⟨рисунок-первичность⟩ возвращают углы своей "охватывающей рамки", как показано на рис. 20. Оператор `center` также допускает операнды ⟨путь-первичность⟩ и ⟨перо-первичность⟩. MetaPost версии 0.30 и новее допускает для `llcorner`, `lrcorner`, ... все три типа аргументов.

Ограничения на тип аргумента для corner-операторов не очень важны, потому что их главное назначение позволить командам `label` и `dotlabel` центрировать свой текст правильно. Заранее определенный макрос

`bbox` ⟨рисунок-первичность⟩

находит прямоугольный путь, представляющий охватывающую рамку для данного рисунка. Если `p` — это картинка, то `bbox p` эквивалентно

```
(llcorner p-{}-lrcorner p-{}-urcorner p-{}-ulcorner p-{}-cycle),
```

за исключением того, что первое допускает небольшой промежуток вокруг `p` как указано внутренней переменной `bboxmargin`.

Рис. 20: Охватывающая рамка и ее угловые точки.

Заметьте, что MetaPost вычисляет охватывающую рамку рисунка `btex ... etex` тем же способом, что и TEX. Это вполне естественно, но вовлекает в рассмотрение факта того, что TEX имеет свойства типа `\strut` и `\rlap`, что позволяют пользователям TEX лгать о размерах рамки.

Когда команды TEX, лгущие о размерах рамки, транслируются в низкоуровневый код MetaPost, команда `setbounds` лжет:

`setbounds` ⟨переменная-картинка⟩ `to` ⟨выражение-путь⟩

делает ⟨переменную-картинку⟩ такой, как если бы ее охватывающая рамка была такой же как заданный путь. Этот путь должен быть циклическим, т. е. замкнутым. Для получения настоящей охватывающей рамки такой картинки присвойте положительное значение внутренней переменной `truecorners`:[12], т. е.

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

производит "`>> (4.9813,6.8078)`", а

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

производит "`>> (15.7742,6.8078)`".

[12]Свойства `setbounds` и `truecorners` присутствуют только в MetaPost версии 0.30 и новее.

9 Продвинутая графика

Все примеры предыдущих разделов были простым рисованием линий с добавлением меток. Этот раздел описывает затенение и средства для генерации не столь простых линий. Затенение делается командой `fill`. В своей простейшей форме команда `fill` требует ⟨выражение-путь⟩, задающее границу региона для заполнения. В синтаксисе

`fill` ⟨выражение-путь⟩

аргумент должен быть циклическим путем, т. е. путем, который описывается замкнутой кривой через `..cycle` или `--cycle`. Например, команда `fill` на рис. 21 строит замкнутый путь продолжением приблизительно полукругового пути `p`. Этот путь имеет ориентацию против часовой стрелки, но это не имеет значение, потому что команда `fill` использует правило “ненулевого вертящегося числа” (non-zero winding number) PostScript [1].

```
beginfig(21);
path p;
p = (-1cm,0)..(0,-1cm)..(1cm,0);
fill p{up}..(0,0){-1,-2}..{up}cycle;
draw p..(0,1cm)..cycle;
endfig;
```

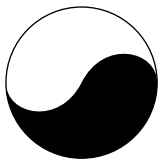


Рис. 21: MetaPost код и соответствующий вывод.

Общая команда `fill`

`fill` ⟨выражение-путь⟩ `withcolor` ⟨выражение-цвет⟩

указывает уровень серого или (если у вас есть цветной принтер) некоторый цвет радуги. ⟨Выражение-цвет⟩ может иметь пять возможных значений, переводимых к четырем возможным цветовым моделям:

Действительный ввод	Переводимое значение
`withcolor` ⟨rgb-цвет⟩c	withrgbcolor c
`withcolor` ⟨cmyk-цвет⟩c	withcmykcolor c
`withcolor` ⟨число⟩c	withgreyscale c
`withcolor` ⟨ложь⟩	withoutcolor
`withcolor` ⟨истина⟩	⟨текущая типовая модель цвета⟩

Для указанных моделей цвета есть также

`fill` ⟨выражение-путь⟩ `withrgbcolor` ⟨выражение-rgb-цвет⟩

`fill` ⟨выражение-путь⟩ `withcmykcolor` ⟨выражение-cmyk-цвет⟩

`fill` ⟨выражение-путь⟩ `withgreyscale` ⟨число⟩

`fill` ⟨выражение-путь⟩ `withoutcolor`

Объект-изображение не может иметь более одной цветовой модели, последнее указание `withcolor`, `withrgbcolor`, `withcmykcolor`, `withgreyscale` или `withoutcolor` устанавливает модель цвета для любого отдельного объекта.

Модель `withoutcolor` требует небольших разъяснений: выбор этой модели означает, что MetaPost не будет писать команду выбора цвета в выходной файл PostScript для этого объекта.

‘Текущая типовая’ модель цвета может быть установлена использованием внутренней переменной `defaultcolormodel`. Таблица 3 перечисляет ее допустимые значения.

Значение	Модель цвета
1	нет модели
3	оттенки серого
5	rgb (по-умолчанию)
7	cmyk

Таблица 3: Поддерживаемые модели цвета.

Рис. 22 иллюстрирует несколько применений команды `fill` для заполнения областей оттенками серого. Пути включают пересечения кругов `a` и `b` и путь `ab`, охватывающий область внутри обоих кругов. Круги `a` и `b` происходят от предопределенного пути `fullcircle`, приблизительно соответствующего кругу с единичным диаметром и с центром в начале координат. Есть также предопределенный путь `halfcircle` — половина `fullcircle` над осью x. Путь `ab` затем инициализируется, используя предопределенный макрос `buildcycle`, который будет обсуждаться вскоре.

```
beginfig(22);
path a, b, aa, ab;
a = fullcircle scaled 2cm;
b = a shifted (0,1cm);
aa = halfcircle scaled 2cm;
ab = buildcycle(aa, 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;
```

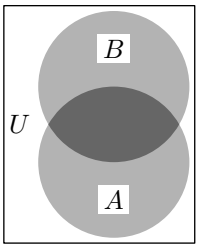


Рис. 22: MetaPost код и соответствующий вывод.

Заполнение круга `a` светлым серым цветом `.7white` и затем такое же заполнение круга `b` дважды заполняет область, где круги пересекаются. Есть правило, что каждая команда `fill` присваивает данный цвет всем точкам покрываемого региона, уничтожая все, что там было, включая линии, текст и заполненные области. Таким образом, важно задавать команды `fill` в правильном порядке. В примере выше перекрываемая область получает одинаковый цвет дважды, оставаясь светлосерой после первых двух команд `fill`. Третья команда `fill` присваивает более темный цвет `.4white` перекрываемой области.

После этого круги и их пересечение получают свои окончательные цвета, но в них нет вырезок для меток. Вырезки получаются командами `unfill`, которые быстро уничтожают области, охватывающии `bbox pa` и `bbox pb`. Более точно, `unfill` — это сокращение заполнения с `withcolor background`, где `background` обычно равен `white`, что подходит при печати на белой бумаге. Если необходимо, то вы можете присвоить новый цвет `background`.

Метки должны быть помещены в картинки `pa` и `pb` для возможности измерения их охватывающих рамок до их рисования. Макрос `thelabel` создает такие картинки и сдвигает их в

позиции, где они готовы для рисования. Использование итоговых картинок в команде `draw` в форме

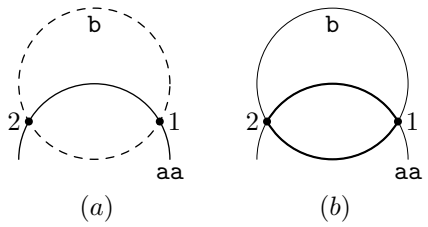
`draw` ⟨выражение-рисунок⟩

добавляет их к текущей картинке `currentpicture` так, что они перезаписывают часть того, что уже нарисовано. На рис. 22 перезаписываются сами белые прямоугольники, созданные `unfill`.

9.1 Построение циклов

Команда `buildcycle` конструирует пути для использования с макросами `fill` или `unfill`. Когда задаются два или более путей, таких как `aa` и `b`, макрос `buildcycle` пытается соединить их части вместе, формируя циклический путь. В рассмотренном случае путь `aa` является полукругом, начинающимся справа от пересечения с путем `b`, затем проходящим через `b` и заканчивающимся снаружи круга слева, как показано на рис. 23a.

Рис. 23b показывает как `buildcycle` формирует замкнутый цикл из кусков путей `aa` и `b`. Макрос `buildcycle` находит два пересечения, помеченные 1 и 2 на рис. 23b. Затем он конструирует циклический путь, показанный выделенным на рисунке, двигаясь вдоль пути `aa` от пересечения 1 к пересечению 2 и затем против часовой стрелки по пути `b` обратно к пересечению 1. Кажется очевидным, что `buildcycle(a,b)` будет производить такой же результат, но основания для этого несколько путанные.



(a) (b)

Рис. 23: (a) Полукруговой путь `aa` с пунктирной линией, отмечающей путь `b`; (b) пути `aa` и `b` с частями, выделяемыми `buildcycle` и показанными жирными линиями.

Проще всего использовать макрос `buildcycle` в ситуациях, подобных рис. 24, где есть более двух аргументов-путей и каждая пара последовательных путей имеет уникальное пересечение. Например, прямая `q0.5` и кривая `p2` пересекаются только в точке P; кривая `p2` и прямая `q1.5` — только в точке Q. Фактически каждая из точек P, Q, R, S является уникальным пересечением и результат команды

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

берет `q0.5` от S до P, затем `p2` от P до Q, затем `q1.5` от Q до R и, наконец, `p4` от R обратно до S. Исследование кода MetaPost для рис. 24 открывает, что вы должны идти назад вдоль `p2` на переходе от P до Q. Все работает вполне совершенно до тех пор, пока точки пересечения определяются уникально, но может обусловить неожиданные результаты, когда пары путей пересекаются более одного раза.

Общее правило для макроса `buildcycle`:

`buildcycle(`p_1`,`p_2`,`p_3`, ... ,`p_k`)`

выбирает пересечение между каждым p_i и p_{i+1} так, чтобы это было как можно дальше на p_i и как можно ближе на p_{i+1}[13]. Нет простого правила для разрешения конфликтов между этими двумя целями, так что вам следует избегать случаев, когда одна точка пересечения случается дальше на p_i и другая точка пересечения случается ближе на p_{i+1}.

[13]Первым находится пересечение между p_k и p_1, затем p_1 и p_2, ... (прим. перев.)

```
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;
```

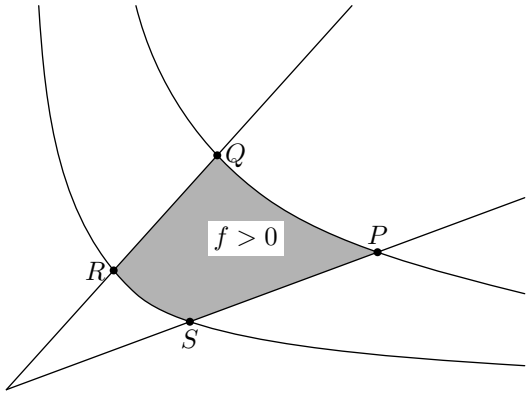


Рис. 24: MetaPost-код и соответствующий вывод.

Установка на самые дальние пересечения для p_i и самые ближние для p_{i+1} ведет к устранения неясности предпочтением идущих впереди подпутей. Для циклических путей, как путь `b` на рис. 23, “близко” и “далеко” относительны по отношению к начальной/конечной точке, которая расположена там, куда вы попадаете обратно, сказав “`..cycle`”. Для пути `b` эта точка устанавливается на самую правую точку на круге.

Более прямой путь для работы с путевыми пересечениями в использовании ⟨вторичного бинарного оператора⟩ `intersectionpoint`, находящего точки P, Q, R и S на рис. 24. Этот макрос находит точку, где два данных пути пересекаются. Если существует более одной точки пересечения, то он выбирает одну; если точек пересечения нет, то макрос генерирует сообщение об ошибке.

9.2 Параметрическая работа с путями

Макрос `intersectionpoint` основан на примитивной операции с именем `intersectiontimes`. Этот ⟨вторичный бинарный оператор⟩ — один из нескольких операторов, работающих с путями параметрически. Он находит пересечение между двумя путями, заданием параметра “время” на каждом из путей. Это ссылка на схему параметризации из раздела 4, определяющего пути как кусочные кубические кривые $\big(X(t), Y(t)\big)$, где диапазон t от нуля до числа отрезков кривой. Другими словами, путь задается, как проходящий через последовательность точек, где $t = 0$ в первой точке, $t = 1$ в следующей, $t = 2$ в следующей и т. д. Результатом

```
a intersectiontimes b
```

будет $(-1, -1)$, если пересечения нет; в противном случае, вы получите пару (t_a, t_b), где t_a — это время на пути `a`, когда он пересекает путь `b`, и t_b — это соответствующее время на пути `b`. Например, предположим, что путь `a` обозначен тонкой линией на рис. 25 и путь `b` обозначен более толстой линией. Если метки показывают значения времени на путях, то пара значений времени, вычисленная в

```
a intersectiontimes b
```

должна быть одной из

$$(0.25, 1.77),\ (0.75, 1.40)\ (2.58, 0.24)$$

и какая из трех будет выбрана зависит от интерпретатора MetaPost. Точное правило выбора из многих точек пересечения несколько сложноватое, но в данном примере вы получите $(0.25, 1.77)$. Меньшие значения времени предпочтительнее больших, так что (t_a, t_b) предпочтительнее, чем (t'_a, t'_b) пока $t_a < t'_a$ и $t_b < t'_b$. Когда нет простого способа минимизировать обе компоненты t_a и t_b, t_a получает больший приоритет, но правила становятся более сложными, когда между t_a и t'_a нет целых чисел. (Больше деталей см. в *The METAFONTbook* [4, Chapter 14]).

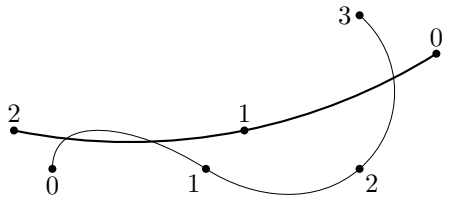


Рис. 25: Два пересекающихся пути с отметками значений времени на каждом.

Оператор `intersectiontimes` более гибкий, чем `intersectionpoint`, потому что существует много всего, что можно сделать со значениями времени на пути. Одно из самых важных — это задать вопрос: “Где проходит путь `p` во время `t`?” Конструкция

`point` ⟨числовое выражение⟩ `of` ⟨путь-первичность⟩

отвечает на этот вопрос. Если ⟨числовое выражение⟩ меньше нуля или больше значения времени, присвоенного последней точке на пути, то конструкция `point of` обычно возвращает последнюю точку пути. Поэтому принято использовать предопределенную константу `infinity` (равную 4095.99998) как ⟨числовое выражение⟩ в конструкции `point of`, когда имеем дело с концом пути.

Такое "бесконечное" значение времени не работает для циклического пути, т. к. значения времени, выходящие за нормальный диапазон, могут в этом случае обрабатываться модульной арифметикой, т. е. для циклического пути `p` через точки z_0, z_1, z_2, …, z_{n-1} с обычным диапазоном параметров $0 \leqslant t < n$,

`point t of p`

может быть вычислено для любого t предварительным взятием t по модулю n. Если модуль n недоступен, то

`length` ⟨путь-первичность⟩

дает целое значение верхнего предела нормального диапазона параметра-времени для заданного пути.

MetaPost использует такие же соответствия между значениями времени и точками на пути при вычислении оператора `subpath`. Синтаксис этого оператора

`subpath` ⟨выражение-пара⟩ `of` ⟨путь-первичность⟩

Если значение ⟨выражения-пары⟩ будет (t_1, t_2) и ⟨путь-первичность⟩ — это p, то результат будет путем, что следует как и p от `point` t_1 `of` p до `point` t_2 `of` p. Если $t_2 < t_1$, то подпуть идет обратно вдоль p.

Важный оператор, основанный на операторе `subpath`, — это ⟨третичный бинарный оператор⟩ `cutbefore`. Для пересекающихся путей p_1 и p_2,

p_1 `cutbefore` p_2

эквивалентно

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

с тем исключением, что он также устанавливает переменную-путь `cuttings` в часть p_1, что отбрасывается. Другими словами, `cutbefore` возвращает свой первый аргумент без части до пересечения. При множественных пересечениях он пытается отбросить наименьшую часть. Если пути не пересекаются, то `cutbefore` возвращает свой первый аргумент.

Есть также аналогичный ⟨третичный бинарный оператор⟩, называемый `cutafter`, который работает применением `cutbefore` с инвертированным временем вдоль своего первого аргумента. Таким образом,

p_1 `cutafter` p_2

пытается отрезать часть p_1 после последнего пересечения с p_2.

Другой оператор

`direction` ⟨числовое выражение⟩ `of` ⟨путь-первичность⟩

находит вектор в направлении ⟨пути-первичности⟩. Он определяется для любого времени, подобно конструкции `point of`. Вектор-результат имеет правильное направление и несколько произвольной размер. Соединение конструкций `point of` и `direction of` дает уравнение для линии тангенса, как показано на рис. 26.

Если вы знаете уклон и вы хотите найти точку на кривой с тангенсом, равным этому уклону, то нужен оператор `directiontime` — обратный к `direction of`. С данными вектором-уклоном и путем

`directiontime` ⟨выражение-пара⟩ `of` ⟨путь-первичность⟩

```
beginfig(26);
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;
```

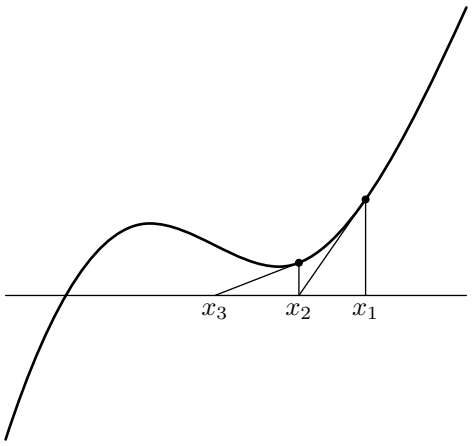


Рис. 26: Код MetaPost и рисунок-результат

возвращает числовое значение, что дает первое время t, когда путь имеет заданное направление. (Если такого времени нет, то результат будет -1.) Например, если `a` — это путь, нарисованный тонкой кривой на рис. 25, то `directiontime (1,1) of a` возвращает 0.2084.

Есть еще предопределенный макрос

`directionpoint` ⟨выражение-пара⟩ `of` ⟨путь-первичность⟩,

который находит первую точку на пути, где есть данное направление. Макрос `directionpoint` производит сообщение об ошибке, если искомое направление отсутствует на пути.

Операторы `arclength` и `arctime of` относятся к "времени" на пути более знакомым образом, используя понятие длины дуги.[14] Выражение

`arclength` ⟨путь-первичность⟩

выдает длину дуги пути. Если `p` — это путь и `a` — число между 0 и `arclength p`, то

`arctime a of p`

дает время `t`, такое что

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

9.3 Аффинные трансформации

Заметьте, что путь `fun` на рис. 26 сначала конструируется как

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

и затем используются операторы `yscaled` и `scaled` для настройки формы и размера пути. Как подсказывает название, выражение с "`yscaled 10`" умножает координаты y на десять так, что каждая точка (x, y) исходного пути будет соответствовать точке $(x, 10y)$ на трансформированном пути.

Вместе с `scaled` и `yscaled` существует семь операторов трансформации с аргументом числом или парой :

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

Большинство этих операторов объясняют сами себя, за исключением `zscaled`, о котором можно думать как о произведении комплексных чисел. Эффект `zscaled` (a, b) — это поворот и масштабирование так, что $(1, 0)$ переходит в (a, b). Эффект от `rotated` θ во вращении на θ градусов против часовой стрелки.

Любая комбинация сдвигов, вращений, наклонов и т. п. — это аффинная трансформация, совокупный эффект которой в трансформации любой пары (x, y) в

$$(t_x + t_{xx}x + t_{xy}y,\ t_y + t_{yx}x + t_{yy}y),$$

[14] Операторы `arclength` и `arctime` доступны только в MetaPost версии 0.50 и новее.

для некоторой шестерки $(t_x, t_y, t_{xx}, t_{xy}, t_{yx}, t_{yy})$. Последняя информация может быть сохранена в переменной типа трансформация, так что `transformed T` может быть эквивалентно

```
xscaled -1 rotated 90 shifted (1,1),
```

если `T` — это переменная-трансформация. Трансформация `T` могла быть инициализирована выражением типа трансформация, например,

```
transform T;
T = identity xscaled -1 rotated 90 shifted (1,1);
```

Этот пример показывает, что выражения-трансформации могут строиться применением трансформационных операторов к другим трансформациям. Стандартная трансформация `identity` — это полезная отправная точка для этого процесса. Все можно проиллюстрировать переводом уравнения выше на естественный язык: "`T` следует быть трансформацией, такой же как `identity`, затем масштабированной по координатам x в -1 раз, вращенной на $90°$ и сдвинутой на $(1, 1)$." Это работает, потому что `identity` — тождественная трансформация, которая ничего не делает, т. е. `transformed identity` — это пустой оператор.

Синтаксис для выражений-трансформаций и операторов трансформации дается на рис. 27. Он включает две дополнительные возможности для ⟨трансформации⟩:

```
reflectedabout(p, q)
```

отражает относительно прямой, определяемой точками p и q и

```
rotatedaround(p, θ)
```

вращает на θ градусов против часовой стрелки вокруг точки p. Например, уравнением для инициализации трансформации `T` может быть

```
T = identity reflectedabout((2,0), (0,2)).
```

⟨вторичная пара⟩ → ⟨вторичная пара⟩⟨трансформация⟩
⟨путь-вторичность⟩ → ⟨путь-вторичность⟩⟨трансформация⟩
⟨рисунок-вторичность⟩ → ⟨рисунок-вторичность⟩⟨трансформация⟩
⟨перо-вторичность⟩ → ⟨перо-вторичность⟩⟨трансформация⟩
⟨трансформация-вторичность⟩ → ⟨трансформация-вторичность⟩⟨трансформация⟩

⟨трансформация⟩ → `rotated`⟨числовая первичность⟩
| `scaled`⟨числовая первичность⟩
| `shifted`⟨первичная пара⟩
| `slanted`⟨числовая первичность⟩
| `transformed`⟨трансформация-первичность⟩
| `xscaled`⟨числовая первичность⟩
| `yscaled`⟨числовая первичность⟩
| `zscaled`⟨первичная пара⟩
| `reflectedabout(`⟨выражение-пара⟩`,`⟨выражение-пара⟩`)`
| `rotatedaround(`⟨выражение-пара⟩`,`⟨числовое выражение⟩`)`

Рис. 27: Синтаксис для трансформаций и родственных операторов

Есть еще унарный оператор с одним аргументом `inverse`, находящий обратную трансформацию, отменяющую эффект трансформации-аргумента. Таким образом, если

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

то

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

Нельзя брать `inverse` для неизвестной трансформации, но мы уже видели, что можно

$$\texttt{T = } \langle\text{выражение-трансформация}\rangle,$$

когда `T` еще не имеет значения. Также возможно применять неизвестную трансформацию к известной паре или трансформации и использовать результат в линейном уравнении. Три таких уравнения достаточны для определения трансформации. Таким образом, уравнения

```
(0,1) transformed T' = (3,4);
(1,1) transformed T' = (7,1);
(1,0) transformed T' = (4,-3);
```

позволяют MetaPost определить, что трансформация `T'` — это комбинация вращения и масштабирования с

$$\begin{aligned} t_{xx} &= 4, & t_{yx} &= -3, \\ t_{yx} &= 3, & t_{yy} &= 4, \\ t_x &= 0, & t_y &= 0. \end{aligned}$$

Уравнения с неизвестными трансформациями рассматриваются как линейные уравнения с шестью параметрами, определяющими трансформацию. Эти шесть параметров могут также именоваться напрямую как

$$\texttt{xpart T},\ \texttt{ypart T},\ \texttt{xxpart T},\ \texttt{xypart T},\ \texttt{yxpart T},\ \texttt{yypart T},$$

где `T` — это трансформация. Например, рис. 28 использует уравнения

```
xxpart T=yypart T; yxpart T=-xypart T
```

для указания, что `T` сохраняет форму, т. е. является комбинацией вращения, сдвига и сохраняющего форму масштабирования.

9.4 Пунктирные линии

Язык MetaPost предоставляет много способов изменять линии помимо простого изменения их толщины. Один такой способ в использовании пунктирных линий, как сделано на рисунках 5 и 23. Синтаксис для этого —

$$\texttt{draw } \langle\text{выражение-путь}\rangle \texttt{ dashed } \langle\text{образец пунктира}\rangle,$$

где ⟨образец пунктира⟩ — это в действительности специальный тип ⟨выражения-картинки⟩. Есть предопределенный ⟨образец пунктира⟩, называемый `evenly`, делающий тире длиной 3 PostScript пункта, разделенные промежутками такого же размера. Другой предопределенный образец, `withdots`, производит точечную линию с точкой на каждые 5 PostScript пунктов.[15] Для более разряженных точек и разряженных и удлиненных тире ⟨образец пунктира⟩ может быть масштабирован как показано на рис. 29.

Другой способ изменить образец линии в изменении его фазы горизонтальным сдвигом. Сдвиг вправо двигает образец вперед вдоль пути, а сдвиг вправо — назад. Рис. 30 иллюстрирует этот эффект. Образец пунктира можно представлять как бесконечно повторяющийся образец, распределенный вдоль горизонтальной линии, где участок линии справа от оси y размещается на пути, где нужны разрывы.

[15] `withdots` есть только в MetaPost версии 0.50 и новее.

```
beginfig(28);
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;
```

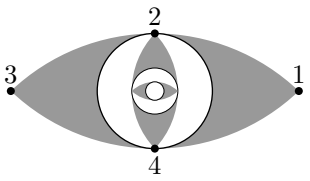


Рис. 28: Код MetaPost и “фрактальная” картинка-результат

dashed withdots scaled 2
dashed withdots
dashed evenly scaled 4
dashed evenly scaled 2
dashed evenly

Рис. 29: Пунктирные линии, помеченные ⟨образцом пунктира⟩, используемым для их создания.

6 draw z6..z7 dashed e4 shifted (18bp,0)
4 draw z4..z5 dashed e4 shifted (12bp,0)
2 draw z2..z3 dashed e4 shifted (6bp,0)
0 draw z0..z1 dashed e4

Рис. 30: Пунктирные линии и команды MetaPost для их рисования, где `e4` ссылается на образец пунктира `evenly scaled 4`.

Когда вы сдвигаете образец пунктира, так что ось y пересекает середину, первое тире урезается. Таким образом, линия с образцами пунктира `e4` начинается с тире длиной 12bp, за которым идет разрыв 12bp, за которым — другое 12bp тире и т. д.; `e4 shifted (-6bp,0)` производит 6bp тире, 12 bp промежуток, затем 12bp тире и т. д. Этот образец пунктира может быть указан более прямо через функцию `dashpattern`:

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

Пример означает "нарисуй первые 6bp линии, затем пропусти следующие 12bp, затем рисуй 6bp и повторяй". Если эта линия с разрывами имеет длину более 30bp, то последние 6bp первой копии образца линии будут соединяться с первыми 6bp следующей копии для формирования тире длиной 12bp. Общий синтаксис для функции `dashpattern` показан на рис. 31.

⟨образец пунктира⟩ → `dashpattern(`⟨список есть/нет⟩`)`
⟨список есть/нет⟩ → ⟨список есть/нет⟩⟨пункт есть/нет⟩ | ⟨пункт есть/нет⟩
⟨пункт есть/нет⟩ → `on`⟨числовая третичность⟩ | `off`⟨числовая третичность⟩

Рис. 31: Синтаксис функции `dashpattern`

Из-за того, что образец пунктира в действительности является особым типом рисунка, функция `dashpattern` возвращает рисунок. Нет необходимости знать структуру такого рисунка, поэтому случайный читатель будет вероятно хотеть перейти к разделу 9.6. Для тех, кто хочет знать, маленький эксперимент показывает, что если `d` — это

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

то `llcorner d` — это $(0, 24)$ и `urcorner d` — $(24, 24)$. Прямое изображение `d` без использования его как образца произведет два тонких горизонтальных отрезка прямой, подобные таким:

— —

Линии в этом примере описывались, как имеющие толщину ноль, но это не имеет значения, потому что толщина линии игнорируется, когда рисунок используется как образец линии.

Общее правило интерпретации рисунка `d`, как образца линии, в том, что отрезки в `d` проектируются на ось x и итоговый образец повторяется бесконечно в обоих направлениях помещением вплотную копий образца. Настоящие длины получаются, начиная с $x = 0$, движением в положительном направлении x.

Чтобы сделать идею "повторять бесконечно" более точной, установим $P(\mathtt{d})$ проекцией `d` на ось x и пусть $\mathrm{shift}(P(\mathtt{d}), x)$ — это результат сдвига `d` на x. Образец, получающийся в результате бесконечного повторения, — это

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

где $\ell(d)$ измеряет длину $P(d)$. Наиболее ограничивающее определение этой длины равно $d_{\max} - d_{\min}$, где $[d_{\min}, d_{\max}]$ — это диапазон координат x в $P(d)$. Фактически MetaPost использует

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

где $y_0(\mathtt{d})$ — это координата y содержимого `d`. Содержимое `d` должно находиться на горизонтальной линии, но если это не так, то интерпретатор MetaPost просто возьмет координату y внутри `d`.

Картинка с образцом пунктира не должна содержать текста или заполненных областей, но она может содержать линии, которые пунктирные. Это дает маленькие и большие разрывы как показано на рис. 32.

```
beginfig(32);
draw dashpattern(on 15bp off 15bp) dashed evenly;
picture p;
p=currentpicture;
currentpicture:=nullpicture;
draw fullcircle scaled 1cm xscaled 3 dashed p;
endfig;
```

Рис. 32: Код MetaPost для образца пунктира и соответствующий вывод

Образцы пунктира также предназначены для использования с `pencircle` или с отсутствием пера; `pensquare` и других сложных перьев следует избегать. Последнее обусловлено тем, что вывод использует примитив PostScript `setdash`, который не очень хорошо взаимодействует с заполненными путями, созданными многоугольными перьями. См. раздел 9.7, стр. 49.

9.5 Включение PostScript

Если вы хотите добавить код PostScript к своим объектам, то вы можете использовать

`withprescript`⟨строковое выражение⟩

и

`withpostscript`⟨строковое выражение⟩

Строки-результаты будут записаны в файл вывода соответственно перед или после текущего объекта, начинаясь с новой строки каждая. Опции `withprescript` или `withpostscript` можно указывать многократно.

Когда вы указываете более одной опции `withprescript` или `withpostscript`, будьте внимательны с фактом того, что эти скрипты используют вложенность: пункты `withprescript` записываются в PostScript-файл как в стек, а пункты `withpostscript` записываются как в очередь.

9.6 Другие опции

Вы могли заметить, что пунктирные линии, производимые `dashed evenly`, содержат больше черного, чем белого. Это из-за эффекта параметра `linecap`, который управляет появлением концов линий и концов отрезков-тире. Есть также и другие способы влиять на появление того, что рисуется MetaPost.

Параметр `linecap` имеет три различных установки — точно такие же, как и в PostScript. Plain MetaPost дает этой внутренней переменной типовое значение `rounded`, что означает рисовать отрезки с закругленными концами, подобно отрезку от `z0` до `z3` на рис. 33. Установка `linecap := butt` обрезает концы так, что тире, производимые `dashed evenly`, имеют длину 3bp, а не 3bp плюс толщина линии. Вы также можете получить концы-квадраты, выходящие за пределы указанных точек-концов, установкой `linecap := squared`, как это сделано на линии от `z2` до `z5` в рис. 33.

Другой параметр, заимствованный из PostScript, влияет на способ, которым команда `draw` рисует углы на изображаемом пути. Параметр `linejoin` может быть `rounded`, `beveled` или `mitered`, как показано на рис. 34. Стандартное значение для plain MetaPost — это `rounded`, оно дает эффект рисования круглым пером.

Когда `linejoin` равен `mitered`, углы изображаются с длинным острием, как показано на рис. 35. Из-за того, что это может быть нежелательным, есть внутренняя переменная с именем `miterlimit`, которая управляет переходом от остроконечного соединения к срезанному соединению. Для Plain MetaPost `miterlimit` имеет стандартное значение 10.0 и соединение линий

```
beginfig(33);
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;
```

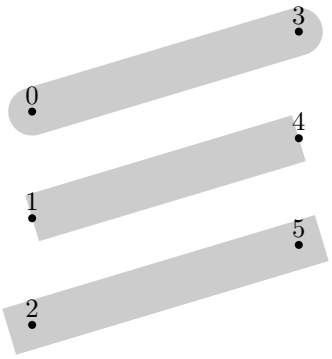


Рис. 33: Код MetaPost и соответствующий вывод

```
beginfig(34);
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;
```

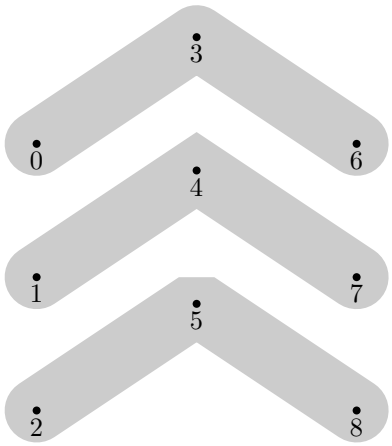


Рис. 34: Код MetaPost и соответствующий вывод

обращается к срезанному тогда, когда отношение длины острия к толщине линии достигает этого значения.

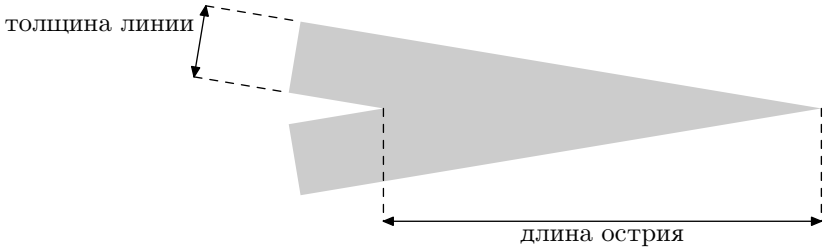


Рис. 35: Длина острия и толщина линии, чье отношение ограничивается `miterlimit`.

Параметры `linecap`, `linejoin` и `miterlimit` очень важны, потому что они влияют и на другие рисуемые объекты. Например, Plain MetaPost имеет команду для рисования стрелок и концы стрелок получаются слегка скругленными при `linejoin` равном `rounded`. Эффект зависит от толщины линии и почти незаметен при стандартной толщине линии в 5bp, как показано на рис. 36.

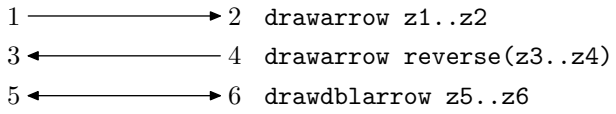


Рис. 36: Три способа рисовать стрелки.

Нарисованные стрелки, подобные тем, что на рис. 36, — это просто результат, когда вместо `draw` ⟨выражение-путь⟩ пишут

`drawarrow` ⟨выражение-путь⟩.

Последнее рисует путь с наконечником стрелы в последней точке пути. Если вы хотите наконечник стрелы в начале пути, то просто используйте унарный оператор `reverse`, который из пути-аргумента делает обратный путь относительно времени, т. е. для пути `p` с `length p` $= n$,

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

— синонимы.

Как показано на рис. 36, команда, начинающаяся с

`drawdblarrow` ⟨выражение-путь⟩,

рисует наконечники стрелок в обоих концах пути. Размер наконечника гарантируется большим, чем толщина линии, но возможно понадобится дополнительная настройка для очень толстых линий. Это делается присвоением нового значения внутренней переменной `ahlength`, которая определяет длину наконечника, как показано на рис. 37. Увеличение `ahlength` более стандартного значения 4 PostScript-пункта до 1.5 сантиметров производит большой наконечник на рис. 37. Есть еще параметр `ahangle`, контролирующий угол на верхушке наконечника стрелки. Типовое значение этого угла — 45 градусов, что показано на рисунке.

Наконечник создается заполнением треугольной области, выделенной белым на рис. 37, и затем прорисовкой вокруг нее текущим пером. Эта комбинация из заполнения и рисования может быть объединена в одну команду `filldraw`:

`filldraw` ⟨выражение-путь⟩ ⟨опциональные `dashed`, `withcolor` и `withpen` пункты⟩;

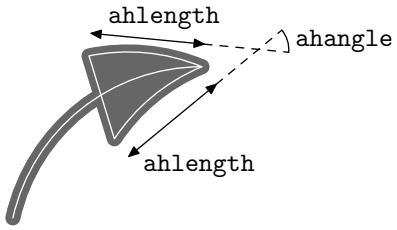


Рис. 37: Большой наконечник стрелки с отмеченными ключевыми параметрами и с выделенными белыми линиями путями для его рисования.

⟨Выражение-путь⟩ должно быть замкнутым циклом, подобным треугольному пути на рис. 37. Этот путь не следует смешивать с аргументом-путем к `drawarrow`, который изображен белой линией на рисунке.

Белые линии, подобные тем, что на рисунке, могут создаваться командой `undraw`. Это стирающая версия `draw`, которая использует `withcolor background` так же, как и команда `unfill`. Есть еще команда `unfilldraw` как раз для случая, который следует из ее названия.

Команды `filldraw`, `undraw`, `unfilldraw` и все команды для рисования стрелок подобны командам `fill` и `draw` в том, что они могут иметь опции `dashed`, `withpen` и `withcolor`. Когда у вас имеется много команд рисования, то удобно иметь возможность применить опцию, такую как `withcolor 0.8white`, ко всем ним без повторения ее всякий раз, как на рисунках 33 и 34. Команда для этой цели называется

`drawoptions(`⟨текст⟩`)`,

где аргумент ⟨текст⟩ дает последовательность опций `dashed`, `withcolor` и `withpen`, которые применяют автоматически ко всем рисующим командам. Если вы укажите

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

и затем захотите нарисовать черную линию, то вам придется переустановить `drawoptions` указанием

`draw` ⟨выражение-путь⟩ `withcolor black`.

Для отключения всех последствий `drawoptions` просто используйте пустой список-аргумент:

```
drawoptions().
```

(Это делается автоматически макросом `beginfig`.)

Из-за того, что неверные опции игнорируются, не будет вреда от команды, подобной

```
drawoptions(dashed evenly),
```

за которой идет последовательность команд `draw` и `fill`. Не имеет смысла использовать образец линии при заливке, поэтому `dashed evenly` игнорируется для команд `fill`. Очевидно, что

`drawoptions(withpen` ⟨выражение-перо⟩`)`

влияет на команды `fill` и `draw`. Фактически существует специальная переменная-перо с именем `currentpen` такая, что `fill ... withpen currentpen` эквивалентно команде `filldraw`.

Так что же точно значит сказать, что опции рисования влияют на команды тогда, когда это имеет смысл? Опция `dashed` ⟨образец пунктира⟩ влияет только на команды

`draw` ⟨выражение-путь⟩,

а появление текста в аргументе типа ⟨выражение-картинка⟩ для команды

`draw` ⟨выражение-картинка⟩

меняется только опцией `withcolor` ⟨выражение-цвет⟩. Во всех других комбинациях команд и опций рисования есть некоторый эффект. Опция, примененная к команде `draw` ⟨выражение-картинка⟩, будет, вообще говоря, влиять на некоторые части рисунка, но не все. Например, опции `dashed` или `withpen` будут влиять на все линии, но не на метки.

9.7 Перья

Предыдущие разделы дали множество примеров типа `pickup` ⟨выражение-перо⟩ и `withpen` ⟨выражение-перо⟩, но не было примеров, отличных от

`pencircle scaled` ⟨числовая первичность⟩,

что создавали линии указанной толщины. Для каллиграфических эффектов, таких как на рис. 38, вы можете применять любые трансформационные операторы, обсуждаемые в разделе 9.3. Отправной точкой для таких трансформаций служит `pencircle` — круг диаметром один PostScript пункт. Поэтому аффинные трансформации производят круговую или эллиптическую форму пера. Толщина линии, рисуемой пером, зависит от того, насколько перпендикулярна линия большей оси эллипса.

```
beginfig(38);
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;
```

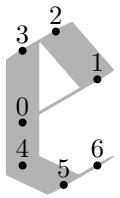


Рис. 38: Код MetaPost и “каллиграфический” рисунок-результат.

Рис. 38 демонстрирует операторы `lft`, `rt`, `top` и `bot`, отвечающие на вопрос: “Если текущее перо поместить в заданную аргументом позицию, то где будут находиться левый, правый, верхний или нижний край?” На этом рисунке текущее перо является эллипсом, заданным командой `pickup`, и его охватывающая рамка равна 0.1734 дюйма ширины и 0.101 дюйма высоты, поэтому `rt x3` равно `x3` + `0.0867in` и `bot y5` равно `y5` − `0.0505in`. Операторы `lft`, `rt`, `top` и `bot` могут также иметь аргумент типа пара — в этом случае они вычисляют координаты x и y наиболее левой, правой, верхней или нижней точки формы пера. Например,

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

для пера на рис. 38. Заметьте, что `beginfig` сбрасывает текущее перо к стандартному значению

`pencircle scaled 0.5bp`

в начале каждой фигуры. Это значение может быть также выбрано в любое время командой `pickup defaultpen`.

Здесь мог бы быть конец истории о перьях, но для совместимости с METAFONT, MetaPost также допускает многоугольные формы перьев. Есть стандартное перо с именем `pensquare`, которое можно трансформировать в перо в форме параллелограмма. Фактически есть даже оператор, называемый `makepen`, берущий путь в форме выпуклого многоугольника и делающий перо такой же формы и размера. Если путь не строго выпуклый или многоугольный, то оператор `makepen` будет спрямлять края и/или сбрасывать некоторые вершины. В частности, `pensquare` эквивалентно

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

Команды `pensquare` и `makepen` не должны использоваться с образцами пунктира. См. конец раздела 9.7, стр. 44.

Обратным к `makepen` является оператор `makepath`, который берет ⟨перо-первичность⟩ и возвращает соответствующий путь. Поэтому `makepath pencircle` производит круговой путь, идентичный `fullcircle`. Это также работает для многоугольного пера, так что

`makepath makepen` ⟨выражение-путь⟩

будет брать любой циклический путь и обращать его в выпуклый многоугольник.

9.8 Вырезка и низкоуровневые команды рисования

Команды рисования, такие как `draw`, `fill`, `filldraw` и `unfill`, — это части макропакета Plain, определенные через более примитивные команды. Основная разница между командами рисования, рассмотренными в предыдущих разделах, и более примитивными в том, что все примитивные команды рисования требуют указания переменной-картинки для хранения результатов. Для команд `fill`, `draw` и родственных им, результаты всегда направляются в переменную-картинку с именем `currentpicture`. Синтаксис для примитивных команд рисования, позволяющий указывать переменную-картинку, показан на рис. 39.

⟨команда addto⟩ →
  `addto`⟨переменная-картинка⟩`also`⟨выражение-картинка⟩⟨список опций⟩
  | `addto`⟨переменная-картинка⟩`contour`⟨выражение-путь⟩⟨список опций⟩
  | `addto`⟨переменная-картинка⟩`doublepath`⟨выражение-путь⟩⟨список опций⟩
⟨список опций⟩ → ⟨пусто⟩ | ⟨опция рисования⟩⟨список опций⟩
⟨опция рисования⟩ → `withcolor`⟨выражение-цвет⟩
  | `withrgbcolor`⟨выражение-rgb-цвет⟩ | `withcmykcolor`⟨выражение-cmyk-цвет⟩
  | `withgreyscale`⟨числовое выражение⟩ | `withoutcolor`
  | `withprescript`⟨строковое выражение⟩ | `withpostscript`⟨строковое выражение⟩
  | `withpen`⟨выражение-перо⟩ | `dashed`⟨выражение-картинка⟩

Рис. 39: Синтаксис для примитивных команд рисования

Синтаксис для примитивных команд рисования совместим с METAFONT. Таблица 4 показывает, как примитивные команды рисования относятся к знакомым `draw` и `fill`. Каждая команда в первой колонке таблицы может заканчиваться своим ⟨списком опций⟩, который эквивалентен добавлению ⟨списка опций⟩ к соответствующей записи во второй колонке. Например,

`draw` p `withpen pencircle`

эквивалентно

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

где currentpen — это специальная переменная, в которой всегда хранится текущее перо. Вторая опция withpen без шума отменяет withpen currentpen в раскрытии draw.

команда	эквивалентный примитив
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*

Таблица 4: Обычные команды рисования и эквивалентные версии примитивов, где q значит currentpen, b — background, p — любой путь, c — циклический путь и *pic* — ⟨выражение-картинка⟩. Заметьте, что непустое поле drawoptions могло бы усложнить записи во второй колонке.

Есть еще две примитивные рисующие команды, недопускающие никаких опций рисования. Первая — это setbounds, обсуждаемая в разделе 8.4; вторая — это clip:

clip ⟨переменная-картинка⟩ to ⟨выражение-путь⟩

По данному циклическому пути, команда clip обрезает содержимое ⟨переменной-картинки⟩ так, что удаляется все, что оказывается снаружи циклического пути. Нет “высокоуровневой” версии этой команды, так что вы должны использовать

clip currentpicture to ⟨выражение-путь⟩,

если вы хотите обрезать currentpicture. Рис. 40 иллюстрирует обрезку.

```
beginfig(40);
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;
```

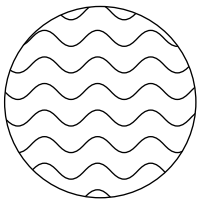


Рис. 40: Код MetaPost и вырезанная картинка-результат.

Все примитивные операции рисования были бы бесполезны без последней операции с именем shipout. Команда

shipout ⟨выражение-рисунок⟩

пишет картинку, как файл PostScript, чье имя определяется значением filenametemplate (см. раздел 3.3). Обычно имя файла заканчивается на .*nnn*, где nnn — это десятичное представление значения внутренней переменной charcode. Имя “charcode” используется для совместимости с METAFONT. Как правило, beginfig устанавливает charcode, а endfig вызывает shipout.

9.9 Направление вывода в переменную-картинку

Иногда может быть желательно сохранить вывод рисующих операций и использовать его потом. Это может быть легко сделано примитивами MetaPost типа `addto`. С другой стороны, из-за того, что высокоуровневые команды рисования, определенные в макропакете Plain, всегда пишут в `currentpicture`, сохранение их вывода требует временного сохранения `currentpicture`, затем сброса ее в `nullpicture`, затем исполнения рисующих операций, затем записи значения `currentpicture` в новую переменную типа `picture` и, наконец, восстановления `currentpicture` в исходное состояние. В MetaPost версии 0.60 новый макрос

`image(` ⟨**команды рисования**⟩ `)`

вводится для упрощения этого задания. Он воспринимает как ввод последовательность произвольных операций рисования и возвращает переменную типа `picture`, содержащую соответствующий вывод, без влияния на `currentpicture`.

Например, в коде рис. 41 объект `wheel` определяется сохранением вывода двух операций `draw` (см. код).

```
picture wheel;
wheel := image(
  draw fullcircle scaled 2u xscaled .8 rotated 30;
  draw fullcircle scaled .15u xscaled .8 rotated 30;
);
```

Объект `wheel` повторно используется в определении другого объекта `car`. Рис. 41 показывает три объекта `car`, нарисованные с двумя разными значениями наклона.

9.10 Работа с компонентами рисунка

Рисунки MetaPost составляются из нарисованных линий, заполненных контуров, текстовых фрагментов, путей для вырезки и путей `setbounds`. Путь `setbounds` дает искусственную охватывающую рамку, нужную для вывода TEX. Рисунок может иметь много компонент каждого типа. Эти компоненты доступны через итерацию в форме

`for` ⟨символьный знак⟩ `within` ⟨выражение-картинка⟩`:` ⟨тело цикла⟩ `endfor`

⟨Тело цикла⟩ может быть чем-угодно, сбалансированным относительно `for` и `endfor`. ⟨Символьный знак⟩ — это переменная-картинка для цикла, сканирующая компоненты рисунка в порядке их рисования. Компонента для пути вырезки или `setbounds` включает все, к чему применяется путь. Поэтому, если одиночный путь вырезки или `setbounds` применяется ко всему в ⟨выражении-картинке⟩, то вся картинка может рассматриваться как одна большая компонента. Для того, чтобы содержимое такой картинки было доступным, итерация `for...within` игнорирует охватывающий путь вырезки или `setbounds` в этом случае. Число компонент, находимых итерацией `for...within`, возвращается

`length` ⟨рисунок-первичность⟩

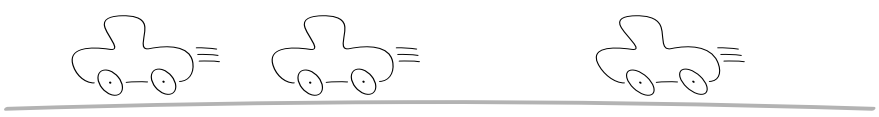


Рис. 41: Копирование объектов оператором `image`.

Как только итерация `for...within` находит компонент картинки, сразу возникает ряд операторов для его идентификации и получения относящейся к нему информации. Оператор

`stroked` ⟨первичное выражение⟩

проверяет, является ли выражение известной картинкой, чей первый компонент — это изображенная линия. Аналогичным образом, операторы `filled` и `textual` возвращают `true`, если первый компонент — это соответственно заполненный контур или фрагмент текста. Операторы `clipped` и `bounded` проверяют, является ли аргумент картинкой, начинающейся с пути вырезки или `setbounds`. Они будут истинны, если первая компонента вырезана или ограничена (командой `setbounds`) или если вся картинка заключена в путь вырезки или `setbounds`.

Есть еще ряд операторов извлечения частей, применяемых к первой компоненте рисунка. Если `p` — это картинка и `stroked p` — истинно, то `pathpart p` — это путь, описывающий нарисованную линию, `penpart p` — это использованное перо, `dashpart p` — это образец пунктира. Если линия без разрывов, то `dashpart p` возвращает пустую картинку.

Такие же операторы для извлечения частей работают, когда `filled p` — истинно, за исключением того, что `dashpart p` не имеет смыла в этом случае.

Для текстовых фрагментов, когда `textual p` — истинно, `textpart p` дает напечатанный текст, `fontpart p` — используемый шрифт и `xpart p`, `ypart p`, `xxpart p`, `xypart p`, `yxpart p`, `yypart p` показывают, как текст сдвигался, вращался и масштабировался.

Наконец, для компонент `stroked`, `filled` и `textual` можно получить цвет, сказав

`colorpart` ⟨компонента⟩

Это возвращает цвет компоненты в соответствующей модели цвета. Модель цвета компоненты может быть идентифицирована оператором `colormodel` (см. таблицу 3 на стр. 34).

Для более детальных операций со цветом есть операторы для извлечения отдельных составляющих цвета из компоненты рисунка. В зависимости от модели цвета цвет компоненты рисунка `p` — это

```
(cyanpart p, magentapart p, yellowpart p, blackpart p)
```

или

```
(redpart p, greenpart p, bluepart p)
```

или

```
(greypart p)
```

или

```
false.
```

Заметьте, что операторы частей цвета `redpart`, `cyanpart` и т. п. должны соответствовать модели цвета компонента рисунка в вопросе. Применение несоответствующего оператора части цвета к компоненту картинки инициирует ошибку и возвращает часть цвета `black` в запрошенной модели цвета. Таким образом, для кода

```
picture pic;
pic := image(fill unitsquare scaled 1cm withcolor (0.3, 0.6, 0.9););
for item within pic:
  show greypart item;
  show cyanpart item;
  show blackpart item;
  show redpart item;
endfor
```

выводом будет (сообщения об ошибках опущены)

```
>> 0
>> 0
>> 1
>> 0.3,
```

потому что в модели оттенков серого черный — это `0`, а в модели cmyk-цветов черный — это `(0,0,0,1)`. Для подошедшей модели rgb-цвета был возвращен верный цветовой компонент.

Когда `clipped p` или `bounded p` — истинно, то `pathpart p` дает путь вырезки или `setbounds`, а другие операторы извлечения частей не имеют смысла. Такие не имеющие смысла получения частей не генерируют ошибок. Они вместо этого возвращают (компоненты) нулевые значения или черный цвет: пустой путь `(0,0)` для `pathpart`, `nullpen` для `penpart`, пустой рисунок для `dashpart`, пустую строку для `textpart` или `fontpart`, 0 для `colormodel`, `greypart`, `redpart`, `greenpart`, `bluepart`, `cyanpart`, `magentapart`, `yellowpart`, один для `blackpart` и черный в текущей типовой модели цвета для `colorpart`.

Подведем итог дискуссии о несоответствующих операторах выделения частей.

1. Вопрос о бессмысленных частях компоненты рисунка, таких как `redpart` для пути вырезки, `textpart` для изображения пером или `pathpart` для текста, спокойно принимается и ответом на него будет либо нулевое значение, либо черный цвет (компонент).

2. Применение оператора цветовой части в неправильной модели цвета к цветному компоненту возвращает черный компонент. Более того, это иниициирует ошибку.

10 Макросы

Уже упоминалось, что MetaPost имеет множество автоматически включаемых макросов, называемое макропакет Plain, и некоторые команды, рассмотренные в предшествующих разделах, определены как макросы и не являются встроенными в MetaPost. Цель этого раздела объяснить, как писать такие макросы.

Макросы без аргументов очень просты. Определение макроса

`def` ⟨символический знак⟩ `=` ⟨текст замены⟩ `enddef`

делает ⟨символический знак⟩ сокращением для ⟨текста замены⟩, где ⟨текст замены⟩ может быть, в сущности, любой последовательностью знаков. Например, макропакет Plain мог бы определить команду `fill` примерно так:

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

Макросы с аргументами похожи, за исключением того, что они имеют формальные параметры, которые говорят, как использовать аргументы в ⟨тексте замены⟩. Например, макрос `rotatedaround` определяется примерно так:

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

Слово `expr` в этом определении значит, что формальные параметры `z` и `d` могут быть произвольными выражениями. Этим параметрам следует соответствовать выражениям-парам, но MetaPost не проверяет этого сразу.

Из-за того, что MetaPost — это интерпретируемый язык, макросы с аргументами очень похожи на подпрограммы. Макросы MetaPost часто используются подобно подпрограммам, поэтому язык включает в себя программные концепции для такого использования. Эти концепции включают локальные переменные, циклы и условные команды.

10.1 Группировка

Группировка в MetaPost весьма важна для функций и локальных переменных. Базовая идея в том, что группа — это последовательность команд, возможно завершающаяся выражением, с обеспечением того, что некоторые символьные знаки могут восстановить свои старые значения в конце группы. Если группа заканчивается выражением, то группа ведет себя подобно вызову функции, что возвращает это выражение. В противном случае, группа — это просто составная команда. Синтаксис для группы —

`begingroup` ⟨список команд⟩ `endgroup`

или

`begingroup` ⟨список команд⟩ ⟨выражение⟩ `endgroup`,

где ⟨список команд⟩ — это последовательность команд, за каждой из которых следует точка с запятой. Группа с ⟨выражением⟩ после ⟨списка команд⟩ ведет себя как ⟨первичность⟩ из рис. 14 или подобно ⟨числовому атому⟩ из рис. 15.

Из-за того, что ⟨текст замены⟩ для макроса `beginfig` начинается с `begingroup`, а ⟨текст замены⟩ для `endfig` заканчивается на `endgroup`, каждая картинка входного файла MetaPost ведет себя как группа. Это позволяет картинкам иметь локальные переменные. Мы уже видели в разделе 7.2, что имена переменных, начинающиеся с `x` или `y` являются локальными в том смысле, что они считаются неизвестными в начале каждой картинки и их значения забываются в конце каждой картинки. Следующий пример иллюстрирует как работает локальность.

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

Результат команды `show`

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

показывает, что `x23` возвращается к своему прежнему значению `3.1`, а `y3a` совершенно неизвестно, как оно и было в `beginfig(17)`.

Локальность переменных `x` и `y` достигается командой

```
save x,y
```

в ⟨тексте замены⟩ для `beginfig`. В общем, переменные делаются локальными командой

`save` ⟨список символических знаков⟩,

где ⟨список символических знаков⟩ — это разделенный запятыми список знаков:

⟨список символических знаков⟩ → ⟨символический знак⟩
| ⟨символический знак⟩`,`⟨список символических знаков⟩

Все переменные, чьи имена начинаются с одного из указанных символических знаков, становятся неизвестными числами, а их прежние значения сохраняются для восстановления в конце

текущей группы. Если команда `save` используется вне группы, то значения по-просту невосстановимо уничтожаются.

Главная цель команды `save` позволить макросам использовать переменные без взаимодействия с существующими переменными или переменными в нескольких вызовах одного и того же макроса. Например, предопределенный макрос `whatever` имеет такой ⟨текст замены⟩

`begingroup save ?; ? endgroup`

Он возвращает неизвестное числовое количество, но оно не зовется больше знаком вопроса, потому что это имя было локальным в группе. Спрашивая имя через `show whatever`, получаем

`>> %CAPSULE`*nnnn*,

где *nnnn* — это идентификационный номер, выбираемый, когда `save` организует исчезновение имени знак вопроса.

Вопреки универсальности, `save` не может использоваться для локального изменения любой внутренней переменной MetaPost. Команда, такая как

`save linecap`,

приведет MetaPost к временному забыванию специального значения этой переменной и сделает ее обычной неизвестной числовой величиной. Если вы хотите нарисовать одну пунктирную линию с `linecap:=butt` и затем вернуться назад к прежнему значению, то вы можете использовать команду `interim` как в следующем примере:

`begingroup interim linecap:=butt;`
`draw` ⟨выражение-путь⟩ `dashed evenly; endgroup`

Это сохранит значение внутренней переменной `linecap`, временно даст ей новое значение и это не приведет к забыванию того, что `linecap` — внутренняя переменная. Общий синтаксис:

`interim` ⟨внутренняя переменная⟩ `:=` ⟨числовое выражение⟩

10.2 Параметризованные макросы

Базовая идея макросов с параметрами в достижении большей гибкости передачей добавочной информации в макрос. Мы уже видели, что определения макросов могут иметь формальные параметры для выражений, задаваемых при вызове макроса. Например, определение

`def rotatedaround(expr z, d) =` ⟨текст замены⟩ `enddef`

позволяет MetaPost-интерпретатору понимать вызовы макроса в форме

`rotatedaround(`⟨выражение⟩`,`⟨выражение⟩`)`

Ключевое слово `expr` в определении макроса значит, что параметры могут быть выражениями любого типа. Когда определение указывает `(expr z, d)`, то формальные параметры `z` и `d` ведут себя подобно переменным подходящего типа. Внутри ⟨текста замены⟩ они могут быть использованы в выражениях, как обычные переменные, но они не могут повторно декларироваться или получать новые значения. Нет ограничений на неизвестные или частично известные аргументы. Поэтому определение

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

прекрасно работает для неизвестных `a` и `b`. Уравнение, такое как

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

может быть использовано для определения `z1` и `z2`.

Заметьте, что определение выше для `midpoint` работает для чисел, пар или цветов — нужно только совпадение типов обоих параметров. Если зачем-то нужен макрос `middlepoint`, работающий только для пути или рисунка, то будет необходимо выполнять проверку `if` типа аргумента. Есть унарный оператор

`path` ⟨первичность⟩,

возвращающий логический результат, показывающий является ли его аргумент путем. Из-за того, что проверка `if` имеет базовый синтаксис

`if` ⟨логическое выражение⟩`:` ⟨сбалансированные знаки⟩ `else:` ⟨сбалансированные знаки⟩ `fi`,

где ⟨сбалансированные знаки⟩ могут быть чем-угодно, сбалансированным относительно `if` и `fi`, полный макрос `middlepoint` с проверкой типа может выглядеть так:

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

Полный синтаксис проверки `if` показан на рис. 42. Позволяются многократные проверки `if`, например,

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

которые можно сократить до

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

где e_1 и e_2 представляют логические выражения.

Обратите внимание, что проверки `if` не являются командами и ⟨сбалансированные знаки⟩ в синтаксических правилах могут не быть полными выражениями или командами. Поэтому можно, пожертвовав наглядностью, убрать два знака в определении `middlepoint`:

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

⟨проверка if⟩ → `if`⟨логическое выражение⟩`:`⟨сбалансированные знаки⟩⟨альтернативы⟩`fi`
⟨альтернативы⟩ → ⟨пусто⟩
| `else:`⟨сбалансированные знаки⟩
| `elseif`⟨логическое выражение⟩`:`⟨сбалансированные знаки⟩⟨альтернативы⟩

Рис. 42: Синтаксис проверки `if`.

Настоящее назначение макросов и проверок `if` в автоматизации решения повторяющихся задач, а также в возможности решать подзадачи по-отдельности. Например, рис. 43 использует макросы `draw_marked`, `mark_angle` и `mark_rt_angle` для отметки линий и углов, появляющихся на рисунке.

Задача макроса `draw_marked` — нарисовать путь с заданным числом пересекающих отметок посередине. Стоит начать с подзадачи рисования одной пересекающей путь `p` перпендикулярно отметки в некоторое время `t`. Макрос `draw_mark` на рис. 44 решает эту подзадачу нахождением сначала вектора `dm`, перпендикулярного `p` в `t`. Для упрощения позиционирования пересекающей отметки определение макроса `draw_marked` берет длину дуги `a` вдоль `p` и оператором `arctime` вычисляет `t`

С решением подзадачи рисования одной отметки, макросу `draw_marked` останется только нарисовать путь и вызвать `draw_mark` с соответствующей длиной дуги. Макрос `draw_marked`

```
beginfig(42);
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;
```

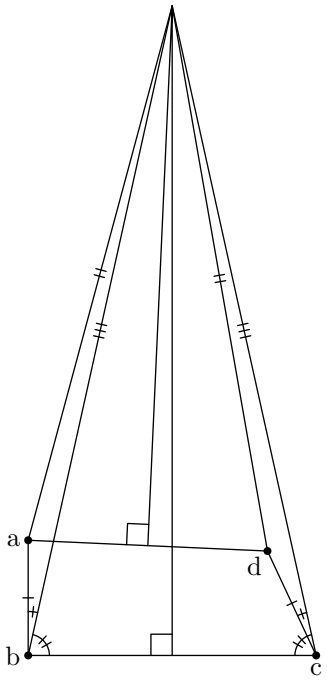


Рис. 43: Код MetaPost и соответствующий рисунок

```
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;
```

Рис. 44: Макросы для рисования пути `p` с `n` пересекающими отметками.

на рис. 44 использует арифметическую прогрессию из `n` значений `a`, центр которых помещается в `.5*arclength p`.

Из-за того, что `draw_marked` работает для кривых, он может быть использован для рисования дуг, генерируемых макросом `mark_angle`. С заданными точками `a`, `b` и `c`, определяющими направленный против часовой стрелки угол в `b`, `mark_angle` должен сгенерировать маленькую дугу от отрезка `ba` до отрезка `bc`. Определение макроса на рис. 45 делает это созданием дуги `p` с единичным радиусом и затем вычислением масштабирующего множителя `s`, который делает ее достаточно большой на вид.

Макрос `mark_rt_angle` намного проще. Он берет общий прямой угол и использует оператор `zscaled` для необходимых вращения и масштабирования.

10.3 Суффиксные и текстовые параметры

Параметры макроса не всегда должны быть выражениями, как в предыдущих примерах. Замена ключевого слова `expr` на `suffix` или `text` в определении макроса объявляет параметры соответственно именами переменных или произвольной последовательностью знаков. Например, есть предопределенный макрос с именем `hide` и текстовым параметром, интерпретируемым как последовательность команд, производящий пустой ⟨текст замены⟩. Другими словами, `hide` исполняет свой аргумент, а следующий знак после него берется так, как будто ничего не произошло. Таким образом,

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

печатает “`>> 2`.”

Если бы `hide` не был предопределен, то его можно было определить примерно так:

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

Рис. 45: Макросы для отметки углов.

Команды, представленные параметром-текстом `t`, будут считаться как часть группы, формирующей аргумент для `ignore`. Из-за того, что `ignore` имеет пустой ⟨текст замены⟩, раскрытие `hide` не произведет совершенно ничего.

Другой пример предопределенного макроса с текстовым параметром — это `dashpattern`. Определение `dashpattern` начинает

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

и затем оно определяет `on` и `off` макросами, создающими требуемую картинку при появлении текстового параметра `t` в тексте замены.

Текстовые параметры очень общие, но их общность иногда допустима. Если же вы хотите в точности передавать имя переменной в макрос, то лучше объявить ее как параметр-суффикс. Например,

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

определяет макрос, что будет получать любую числовую переменную, добавлять один к ней и возвращать новое значение. Из-за того, что имена переменных могут состоять из более одного знака,

```
incr(a3b)
```

вполне допустимо, если `a3b` — числовая переменная. Параметры-суффиксы являются несколько более общими, чем имена переменных, потому что определения на рис. 16 допускает ⟨суффикс⟩, начинающийся с ⟨индекса⟩.

Рис. 46 показывает как параметры suffix и expr могут быть использованы вместе. Макрос `getmid` берет переменную-путь и создает массивы точек и направления, чьи имена получаются присоединением `mid`, `off` и `dir` к переменной-пути. Макрос `joinup` берет массив точек и направлений и создает путь длины `n`, проходящий через каждую `pt[i]` с направлением `d[i]` или `−d[i]`.

Начинающееся с

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

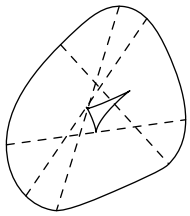
определение может восприниматься так, что вызов макроса `joinup` может иметь два набора скобок как в

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

```
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(45)
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;
```



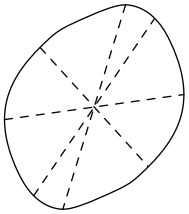


Рис. 46: Код MetaPost и соответствующий рисунок

вместо

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

Обе формы являются допустимыми. Параметры в вызове макроса могут разделяться запятыми или парами `)(`. Есть только одно ограничение — за текстовым параметром должна идти правая скобка. Например, макрос `foo` с одним параметром text и другим expr может быть вызван

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

в этом случае текстовый параметр — это "`a,b`", а параметр-выражение — `c`, но

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

устанавливает параметр-текст в "`a,b,c`" и оставляет интерпретатор MetaPost продолжать искать параметр-выражение.

10.4 Макросы vardef

Определение макроса может начинаться с `vardef` вместо `def`. Макрос, определенный таким образом, называется vardef-макросом. Они особенно хорошо подходят к приложениям, где макросы используются как функции или процедуры. Главная идея в том, что vardef-макрос подобен переменной типа "макрос".

Вместо `def` ⟨символический знак⟩ vardef-макрос начинает

`vardef` ⟨обобщенная переменная⟩,

где ⟨обобщенная переменная⟩ – это имя переменной с числовыми индексами, замененными на символ обобщенного индекса — `[]`. Другими словами, имя после `vardef` обязано иметь в точности такой же синтаксис как и имя в декларации переменной. Оно — последовательность этикеток и символов обобщенного индекса, начинающаяся с этикетки, где этикетка — это символический знак, не являющийся макросом или примитивным оператором, как объяснялось в разделе 7.2.

В простейшем случае имя vardef-макроса состоит из одной этикетки. В таких обстоятельствах `def` и `vardef` обеспечивают почти одинаковую функциональность. Наиболее очевидная разница в том, что `begingroup` и `endgroup` автоматически вставляются соответственно в начало и конец ⟨текста замены⟩ каждого vardef-макроса. Это делает ⟨текст замены⟩ группой и поэтому vardef-макрос ведет себя подобно процедуре или функции.

Другое свойство vardef-макросов заключается в разрешении имен из многих этикеток и имен, содержащих обобщенные индексы. Когда имя vardef-макроса имеет обобщенные индексы, то числовые значения для них должны быть заданы при вызове этого макроса. После определения макроса

`vardef a[]b(expr p) =` ⟨текст замены⟩ `enddef;`

`a2b((1,2))` и `a3b((1,2)..(3,4))` — это вызовы макроса. Но как может ⟨текст замены⟩ сказать о разнице между `a2b` и `a3b`? Два неявных параметра-суффикса автоматически предоставляются для этой цели. Каждый vardef-макрос имеет параметры-суффиксы `#@` и `@`, где `@` — это последний знак в имени вызова макроса, а `#@` — это все предшествующее последнему знаку. Поэтому `#@` равно `a2`, когда данное имя — `a2b`, и `a3`, когда данное имя — `a3b`.

Предположим, например, что макрос `a[]b` сдвигает свой аргумент на величину, зависящую от имени вызова этого макроса. Такой макрос можно определить примерно так:

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

Тогда `a2b((1,2))` означает `(1,2) shifted (a2,b)` и `a3b((1,2)..(3,4))` означает

```
((1,2)..(3,4)) shifted (a3,b).
```

Если макрос был `a.b[]`, то `#@` всегда будет `a.b`, а параметр `@` будет давать числовой индекс. Таким образом, `a@` будет ссылаться на элемент массива `a[]`. Заметьте, что `@` — это параметр-суффикс, а не параметр-выражение, поэтому выражение, подобное `@+1`, ошибочно. Единственный способ получить числовые значения индексов параметра-суффикса в извлечении их из строки, возвращаемой оператором `str`. Этот оператор по суффиксу возвращает его строковое представление. Поэтому `str @` будет `"3"` для `a.b3` и `"3.14"` для `a.b3.14` или `a.b[3.14]`. Из-за того, что синтаксис для ⟨суффикса⟩ на рис. 16 требует заключать отрицательные индексы в квадратные скобки, `str @` возвращает `"[-3]"` для `a.b[-3]`.

Оператор `str`, как правило, используется только при крайней необходимости. Лучше использовать параметры-суффиксы только как имена переменных или суффиксы. Наилучший пример vardef-макроса, использующего суффиксы, — это макрос, определяющий соглашение по `z`. Определение использует специальный знак `@#`, который ссылается на суффикс после имени макроса:

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

Оно означает, что любое имя переменной, чей первый знак `z`, эквивалентно паре переменных, чьи имена получаются заменой `z` на `x` и `y`. Например, `z.a1` вызывает макрос `z` с параметром-суффиксом `@#`, установленным в `a1`.

В общем,

`vardef` ⟨обобщенная переменная⟩`@#`

является альтернативой `vardef` ⟨обобщенная переменная⟩, приводящей интерпретатор MetaPost к выделению суффикса, следующего за именем, заданным в вызове макроса, и делающей его доступным как параметр-суффикс `@#`.

Суммируем особые свойства vardef-макросов: они допускают как широкий класс имен для макросов, так и имена макросов, за которыми следует специальный параметр-суффикс. Более того, `begingroup` и `endgroup` автоматически добавляются к ⟨тексту замены⟩ vardef-макроса. Таким образом, использование `vardef` вместо `def` для определения макроса `joinup` на рис. 46 позволит избежать нужды явно включать `begingroup` и `endgroup` в определение макроса.

Фактически, большинство определений макросов в предыдущих примерах могут использовать `vardef` вместо `def`. Обычно не имеет большого значения, который из них использовать, но есть хорошее общее правило: используйте `vardef`, если вы собираетесь использовать макрос как функцию или процедуру. Следующее сравнение должно помочь при решении, когда использовать `vardef`.

- Vardef-макрос автоматически окружен `begingroup` и `endgroup`.
- Имя vardef-макроса может быть более одного знака длиной и оно может содержать индексы.
- Vardef-макрос может иметь доступ к суффиксу, следующему за именем макроса при его вызове.
- Когда символический знак используется в имени vardef-макроса, он остается этикеткой и может по-прежнему использоваться в именах других переменных. Поэтому `p5dir` — это правильное имя переменной, несмотря на то, что `dir` — это vardef-макрос, но обычный макрос, такой как `...`, не может быть использован в имени переменной. Это к лучшему, потому как `z5...z6` естественно считать выражением-путем, а не составным именем переменной.

10.5 Определение унарных и бинарных макросов

Уже не раз упоминалось, что некоторые обсуждаемые до сих пор операторы и команды на самом деле являются предопределенными макросами. Это касается унарных операторов `round` и

unitvector, команд fill и draw, бинарных операторов dotprod и intersectionpoint. Главная разница между этими макросами и теми, определения которых мы уже знаем, в том, как определять синтаксис их аргументов.

Макросы round и unitvector — это примеры того, что на рис. 14 зовется ⟨унарным оператором⟩. За ними следует выражение-первичность. Для указания аргумента макроса такого типа определение макроса должно выглядеть подобно этому:

vardef round primary u =⟨текст замены⟩ enddef;

Параметр u — это expr-параметр и он может быть использован в точности также как параметр-выражение, определяемый обычным синтаксисом,

(expr u)

Как предполагает пример с round, макрос может определяться с параметром ⟨вторичностью⟩, ⟨третичностью⟩ или ⟨выражением⟩. Например, предопределенное определение макроса fill примерно такое

def fill expr c = addto currentpicture contour c enddef;

Возможно даже определить макрос в роли ⟨of-оператора⟩ из рис. 14. Например, макрос direction of имеет определение такой формы:

vardef direction expr t of p =⟨текст замены⟩ enddef;

Макросы можно также определять с поведением, подобным бинарным операторам. Например, определение макроса dotprod имеет форму

primarydef w dotprod z =⟨текст замены⟩ enddef;

Эта форма делает dotprod ⟨первичным бинарным оператором⟩. Похожим образом secondarydef и tertiarydef вводят определения ⟨вторичного бинарного оператора⟩ и ⟨третичного бинарного оператора⟩. Все они определяют обычные макросы, а не vardef-макросы, например, "primaryvardef" не существует.

Таким образом, определения макросов могут вводиться с def, vardef, primarydef, secondarydef или tertiarydef. ⟨Текст замены⟩ — это любой список знаков, сбалансированный относительно пар def-enddef, где все пять определяющих макрос знаков рассматриваются подобными def в соответствии def-enddef.

Весь синтаксис для определений макросов приводится на рис. 47. В этом синтаксисе есть несколько сюрпризов. Параметры макроса могут иметь ⟨отделенную часть⟩ и ⟨неотделенную часть⟩. Обычно одна из них ⟨пусто⟩, но возможно иметь обе части непустыми:

def foo(text a) expr b =⟨текст замены⟩ enddef;

Это определяет макрос foo с текстовым параметром в скобках, за которым идет выражение.

Синтаксис также позволяет ⟨неотделенной части⟩ указывать тип аргумента suffix или text. Пример макроса с неотделенным параметром-суффиксом — это предопределенный макрос incr, который в действительности определяется примерно так:

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

Это делает incr функцией, что берет переменную, увеличивает ее и возвращает новое значение. Неотделенные параметры-суффиксы могут быть в скобках, поэтому и incr a, и incr(a) правильны, если a — это числовая переменная. Есть также похожий предопределенный макрос decr, вычитающий 1.

⟨определение макроса⟩ → ⟨заголовок макроса⟩=⟨текст замены⟩ enddef
⟨заголовок макроса⟩ → def ⟨символический знак⟩⟨отделенная часть⟩⟨неотделенная часть⟩
| vardef ⟨обобщенная переменная⟩⟨отделенная часть⟩⟨неотделенная часть⟩
| vardef ⟨обобщенная переменная⟩@#⟨отделенная часть⟩⟨неотделенная часть⟩
| ⟨определение бинарности⟩⟨параметр⟩⟨символический знак⟩⟨параметр⟩
⟨отделенная часть⟩ → ⟨пусто⟩
| ⟨отделенная часть⟩(⟨тип параметра⟩⟨знаки параметра⟩)
⟨тип параметра⟩ → expr | suffix | text
⟨знаки параметра⟩ → ⟨параметр⟩ | ⟨знаки параметра⟩,⟨параметр⟩
⟨параметр⟩ → ⟨символический знак⟩
⟨неотделенная часть⟩ → ⟨пусто⟩
| ⟨тип параметра⟩⟨параметр⟩
| ⟨приоритет⟩⟨параметр⟩
| expr ⟨параметр⟩ of ⟨параметр⟩
⟨приоритет⟩ → primary | secondary | tertiary
⟨определение бинарности⟩ → primarydef | secondarydef | tertiatydef

Рис. 47: Синтаксис определений макросов

Неотделенные текстовые параметры распространяются до конца команды. Более точно, неотделенный текстовый параметр — это список знаков, следующих за вызовом макроса до первого "`;`" или "`endgroup`" или "`end`", с тем уточнением, что аргумент, содержащий "`begingroup`" будет всегда включать соответствующий "`endgroup`". Пример неотделенного текстового параметра — в предопределенном макросе `cutdraw`, чье определение выглядит примерно так

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

Это делает `cutdraw` синонимом `draw`, но с другим значением `linecap`. Это макрос предоставляется в основном для совместимости с METAFONT.

11 Циклы

Многочисленные примеры в предыдущих разделах использовали простую форму цикла `for`,

for ⟨символический знак⟩ = ⟨выражение⟩ upto ⟨выражение⟩ : ⟨тело цикла⟩ endfor

Конструировать цикл с уменьшением также просто — нужно лишь заменить `upto` на `downto` и сделать второе ⟨выражение⟩ меньшим первого. Этот раздел описывает более сложные ситуации: прогрессии; циклы, где параметр цикла ведет себя как суффикс; способы выхода из цикла.

Начнем с представления общего факта того, что `upto` — это предопределенный макрос, равный

`step 1 until`,

и `downto` — макрос, равный `step -1 until`. Цикл, начинающийся с

`for i=a step b until c`,

перебирает последовательность `i` из значений `a`, $a+b$, $a+2b$, ..., останавливаясь до того как `i` пройдет за `c`, т. е. цикл перебирает значения `i`, где $i \leqslant c$ при $b > 0$ и $i \geqslant c$ при $b < 0$. Для $b = 0$ цикл никогда не закончится даже при $a = c$.

Лучше всего использовать такой цикл только тогда, когда шаг цикла — это целое или число, представимое точно в виде дроби, кратной $\frac{1}{65536}$. В противном случае будет накапливаться ошибка и параметр цикла может не достичь ожидаемого конечного значения. Например,

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

покажет десять значений `i`, последнее из которых будет 0.90005.

Стандартный способ избежать проблемы, связанной с нецелым размером шага, в итерации по целым значениям и умножению их при использовании на масштабирующий множитель, как это было сделано на рисунках 1 и 40.

Есть альтернатива — можно указывать значения для перебора явно. Любая последовательность из нуля и более выражений, разделенных запятыми, может использоваться на месте `a step b upto c`. Причем, все выражения не обязаны иметь одинаковый тип или известные значения. Таким образом,

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

покажет четыре значения из списка.

Заметьте, что тело цикла в примере выше — это команда, за которой следует точка с запятой. Для тела цикла естественно содержать одну или более команду, хотя это и не обязательно. Цикл подобен определению макроса, за которым следует вызов этого макроса. Тело цикла может быть практически любой последовательностью знаков, имеющих смысл в совокупности. Поэтому (нелепая) команда

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

эквивалента

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

См. рис 18 с более реалистическим подходящим примером.

Если цикл подобен определению макроса, то параметр цикла подобен `expr`-параметру. Он может представлять любое значение, но он не является переменной и его нельзя менять командой присваивания. Последнее можно отменить использованием цикла `forsuffixes`. Цикл `forsuffixes` во многом похож на цикл `for`, но в нем параметр цикла ведет себя как параметр-суффикс. Используется синтаксис

`forsuffixes` ⟨символический знак⟩ `=` ⟨список суффиксов⟩ `:` ⟨тело цикла⟩ `endfor`,

где ⟨список суффиксов⟩ — это разделенный запятыми список. Если некоторые суффиксы ⟨пусты⟩, то ⟨тело цикла⟩ выполняется с параметром цикла, установленным в пустой суффикс.

Хороший пример для цикла `forsuffixes` — это определение макроса `dotlabels`:

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

Он объясняет, почему параметр `dotlabels` должен быть разделенный запятыми список суффиксов. Большинство макросов с разделенными запятыми списками переменной длины используют их в циклах `for` или `forsuffixes` именно таким образом, т. е. для перебора значений.

Когда нет значений для перебора, вы можете использовать цикл `forever`,

`forever:` ⟨тело цикла⟩ `endfor`

Для завершения такого цикла, когда логическое условие станет истинным, используйте пункт выхода:

`exitif` ⟨логическое выражение⟩`;`

Когда интерпретатор MetaPost встречает пункт выхода, он вычисляет ⟨логическое выражение⟩ и выходит из текущего цикла, если оно истинно. Если более удобно выйти из цикла, когда выражение станет ложным, то используйте предопределенный макрос `exitunless`.

Поэтому вариант MetaPost для цикла **while** — это

`forever: exitunless` ⟨логическое выражение⟩`;` ⟨тело цикла⟩ `endfor`

Пункт выхода может помещаться как непосредственно перед `endfor`, так и в любом другом месте ⟨тела цикла⟩. Любой цикл `for`, `forever` или `forsuffixes` может практически содержать любое количество пунктов выхода.

Сводка синтаксиса цикла, показанная на рис. 48, не упоминает явно пункты выхода, потому что ⟨тело цикла⟩ может быть в действительности любой последовательностью знаков. Единственное ограничение в том, что ⟨тело цикла⟩ должно быть сбалансировано относительно `for` и `endfor`. Конечно, в этом балансе `forsuffixes` и `forever` рассматриваются как `for`.

⟨цикл⟩ → ⟨заголовок цикла⟩`:` ⟨тело цикла⟩`endfor`
⟨заголовок цикла⟩ → `for` ⟨символический знак⟩ `=` ⟨прогрессия⟩
| `for` ⟨символический знак⟩ `=` ⟨список for⟩
| `forsuffixes` ⟨символический знак⟩ `=` ⟨список суффиксов⟩
| `forever`
⟨прогрессия⟩ → ⟨числовое выражение⟩ `upto` ⟨числовое выражение⟩
| ⟨числовое выражение⟩ `downto` ⟨числовое выражение⟩
| ⟨числовое выражение⟩ `step` ⟨числовое выражение⟩ `until` ⟨числовое выражение⟩
⟨список for⟩ → ⟨выражение⟩ | ⟨список for⟩`,` ⟨выражение⟩
⟨список суффиксов⟩ → ⟨суффикс⟩ | ⟨список суффиксов⟩`,` ⟨суффикс⟩

Рис. 48: Синтаксис для циклов

12 Изготовление рамок

Этот раздел описывает дополнительные макросы, не включенные в Plain MetaPost, что предоставляют удобства делать то, в чем хорош *pic* [3]. А то, что следует далее — это описание, как использовать макросы, содержащиеся в файле `boxes.mp`. Этот файл размещается в специальном каталоге для макросов MetaPost и обеспечивающего программного обеспечения[16] и должен стать доступным через команду MetaPost `input boxes` до любых рисунков, использующих макросы для создания рамок. Синтаксис команды `input` —

`input` ⟨имя файла⟩,

где окончание "`.mp`" может опускаться в имени файла. Команда `input` смотрит сначала в текущем каталоге, а затем в специальном каталоге для макросов. Пользователи, заинтересовавшиеся в написании макросов, возможно захотят посмотреть на файл `boxes.mp` в этом каталоге.

С времени появления пакета `boxes` в MetaPost-сообществе были разработаны несколько альтернативных пакетов разных видов для рисования рамок. Наиболее известные из них — это `MetaObj`, `MetaUML`, `expressg` и `blockdraw_mp`. Если вы собираетесь создавать много структурных рисунков, диаграмм и т. п., то эти пакеты тоже могут стать для вас интересным ресурсом.

12.1 Прямоугольные рамки

Главная идея создающих рамки макросов в том, что можно сказать

`boxit.`⟨суффикс⟩`(`⟨выражение-картинка⟩`),`

[16] Имя этого каталога подобно чему-то типа `/usr/lib/mp/lib`, но зависит от системы.

где ⟨суффикс⟩ не начинается с индекса[17]. Эта конструкция создает переменные-пары ⟨суффикс⟩`.c`, ⟨суффикс⟩`.n`, ⟨суффикс⟩`.e`, ..., что могут затем использоваться при размещении рисунка перед его рисованием отдельной командой, например,

`drawboxed(`⟨список суффиксов⟩`)`

Аргумент `drawboxed` должен быть разделенным запятыми списком имен рамок, где имя рамки — это ⟨суффикс⟩, с которым вызывается `boxit`.

Для команды `boxit.bb(pic)` имя рамки — это `bb`, а содержимое рамки — это рисунок `pic`. В этом случае, `bb.c` — это позиция помещения центра картинки `pic`, а `bb.sw`, `bb.se`, `bb.ne` и `bb.nw` — это углы прямоугольного пути, что будет окружать картинку-результат. Переменные `bb.dx` и `bb.dy` дают промежутки между сдвинутой версией `pic` и окружающим прямоугольником, а `bb.off` — это расстояние, на которое `pic` сдвигается для достижения всего этого.

Когда макрос `boxit` вызывается с именем рамки b, то это дает линейные уравнения, что заставляют b`.sw`, b`.se`, b`.ne` и b`.nw` быть углами прямоугольника вдоль осей x и y, содержащим внутри картинку, показанную серым прямоугольником на рис. 49. Значения b`.dx`, b`.dy` и b`.c` остаются неуказанными, так что пользователь может задать уравнения для размещения рамок. Если такие уравнения не задаются, то макросы типа `drawboxed` могут это обнаруживать и задавать стандартные значения. Стандартные значения для переменных `dx` и `dy` управляются внутренними переменными `defaultdx` и `defaultdy`.

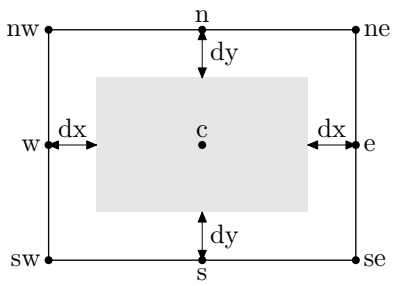


Рис. 49: Отношения между рисунком, заданным для `boxit`, и связанными с этим переменными. Рисунок изображен серым прямоугольником.

Если b представляет имя рамки, то `drawboxed(`b`)` рисует прямоугольную границу содержимого b, а затем и само содержимое. Этот охватывающий прямоугольник может быть использован отдельно как `bpath` b или, в общем случае,

`bpath` ⟨имя рамки⟩

Это полезно в комбинации с операторами типа `cutbefore` и `cutafter` для контроля за путями, входящими в рамку. Например, если a и b — это имена рамок и p — это путь из a`.c` в b`.c`, то

`drawarrow` p `cutbefore bpath` a `cutafter bpath` b

рисует стрелку от края рамки a до края рамки b.

Рис. 50 показывает практический пример включения нескольких стрелок, нарисованных с `cutafter bpath` ⟨имя рамки⟩. Поучительно сравнить рис. 50 с похожей картинкой в руководстве по *pic* [3]. Рисунок использует макрос

`boxjoin(`⟨текст уравнений⟩`)`

[17]Некоторые ранние версии делающих рамки макросов не позволяли никаких индексов в суффиксе `boxit`.

для управления отношениями между последовательными рамками. В ⟨тексте уравнений⟩ `a` и `b` представляют имена рамок, заданные в последовательных вызовах `boxit`, а ⟨текст уравнений⟩ дает уравнения для контроля за относительными размерами и позициями рамок.

Например, вторая строка ввода для приведенного рисунка содержит

```
boxjoin(a.se=b.sw; a.ne=b.nw)
```

Это размещает рамки горизонтально в ряд, благодаря заданию дополнительных уравнений, вызываемых всякий раз, когда некоторая рамка `b` следует за некоторой рамкой `a`. Этим достигается

```
a.se=ni.sw; a.ne=ni.nw
```

Следующая пара рамок — это `ni` и `di`. На этот раз неявно генерируемые уравнения —

```
ni.se=di.sw; ni.ne=di.nw
```

Этот процесс продолжается до тех пор, пока новый `boxjoin` не будет задан. В нашем случае новой декларацией будет

```
boxjoin(a.sw=b.nw; a.se=b.ne)
```

— она соединяет рамки друг под другом.

После вызова `boxit` для первых восьми рамок от `a` до `dk`, высоты рамок принуждаются к взаимному соответствию, но широты остаются неизвестными. Таким образом, макрос `drawboxed` нуждается в присваивании типовых значений переменным ⟨имя рамки⟩`.dx` и ⟨имя рамки⟩`.dy`. Первым делом `di.dx` и `di.dy` получают значения по-умолчанию, так что все рамки принуждаются быть достаточно большими для вмещения содержимого `di`.

Макрос, который в действительности присваивает типовые значения переменным `dx` и `dy`, зовется `fixsize`. Он берет список имен рамок и рассматривает их по-одному за раз, гарантируя, что каждая рамка будет иметь заданные размер и форму. Затем макрос с именем `fixpos` берет этот же самый список имен рамок и присваивает типовые значения переменным ⟨имя рамки⟩`.off`, как необходимо для фиксации позиции каждой рамки. Использование `fixsize` для фиксации размеров каждой рамки до присваивания позиции любой из них может обычно сократить число требуемых позиций по-умолчанию до одной.

Из-за того, что охватывающий путь для рамки не может быть вычислен до тех пор, пока размер, форма и позиция рамки неопределены, макрос `bpath` применяет `fixsize` и `fixpos` к своим аргументам. Другие макросы, что делают также, включают

`pic` ⟨имя рамки⟩,

где ⟨имя рамки⟩ — это суффикс, возможно, в скобках. Он возвращает содержание именованной рамки как картинки, позиционированной так, что

`draw pic`⟨имя рамки⟩

изображает содержимое рамки без охватывающего прямоугольника. Эта операция может также быть усовершенствована макросом `drawunboxed`, берущим разделенный запятыми список имен рамок. Есть еще макрос `drawboxes`, рисующий только охватывающие прямоугольники.

Другой способ нарисовать пустые прямоугольники — это просто

`boxit`⟨имя рамки⟩`()`

без рисунка-аргумента, как это делалось несколько раз на рис. 50. Это похоже на вызов `boxit` с пустым аргументом. Кроме того, аргумент может быть строковым выражением вместо выражения-рисунка и в этом случае строка изображается шрифтом по-умолчанию.

```
input boxes
beginfig(49);
boxjoin(a.se=b.sw; a.ne=b.nw);
boxit.a(btex\strut$\cdots$ etex);    boxit.ni(btex\strut$n_i$ etex);
boxit.di(btex\strut$d_i$ etex);      boxit.ni1(btex\strut$n_{i+1}$ etex);
boxit.di1(btex\strut$d_{i+1}$ etex); boxit.aa(btex\strut$\cdots$ etex);
boxit.nk(btex\strut$n_k$ etex);      boxit.dk(btex\strut$d_k$ etex);
drawboxed(di,a,ni,ni1,di1,aa,nk,dk); label.lft("ndtable:", a.w);
interim defaultdy:=7bp;
boxjoin(a.sw=b.nw; a.se=b.ne);
boxit.ba(); boxit.bb(); boxit.bc();
boxit.bd(btex $\vdots$ etex); boxit.be(); boxit.bf();
bd.dx=8bp; ba.ne=a.sw-(15bp,10bp);
drawboxed(ba,bb,bc,bd,be,bf); label.lft("hashtab:",ba.w);
vardef ndblock suffix $ =
  boxjoin(a.sw=b.nw; a.se=b.ne);
  forsuffixes $$=$1,$2,$3: boxit$$(); ($$dx,$$dy)=(5.5bp,4bp);
  endfor; enddef;
ndblock nda;  ndblock ndb;  ndblock ndc;
nda1.c-bb.c = ndb1.c-nda3.c = (whatever,0);
xpart ndb3.se = xpart ndc1.ne = xpart di.c;
ndc1.c - be.c = (whatever,0);
drawboxed(nda1,nda2,nda3, ndb1,ndb2,ndb3, ndc1,ndc2,ndc3);
drawarrow bb.c -- nda1.w;
drawarrow be.c -- ndc1.w;
drawarrow nda3.c -- ndb1.w;
drawarrow nda1.c{right}..{curl0}ni.c cutafter bpath ni;
drawarrow nda2.c{right}..{curl0}di.c cutafter bpath di;
drawarrow ndc1.c{right}..{curl0}ni1.c cutafter bpath ni1;
drawarrow ndc2.c{right}..{curl0}di1.c cutafter bpath di1;
drawarrow ndb1.c{right}..nk.c cutafter bpath nk;
drawarrow ndb2.c{right}..dk.c cutafter bpath dk;
x.ptr=xpart aa.c;   y.ptr=ypart ndc1.ne;
drawarrow subpath (0,.7) of (z.ptr..{left}ndc3.c) dashed evenly;
label.rt(btex \strut ndblock etex, z.ptr); endfig;
```

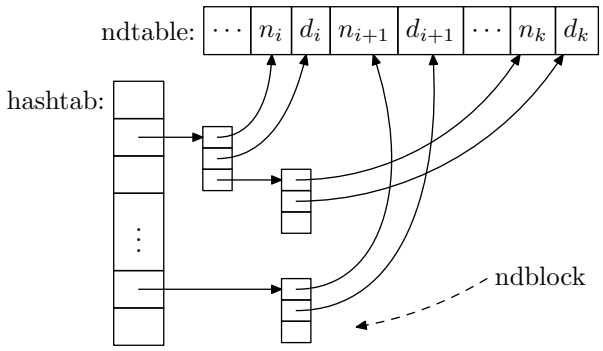


Рис. 50: Код MetaPost и соответствующий рисунок

12.2 Круглые и овальные рамки

Круговые и овальные рамки во многом похожи на прямоугольные, отличаясь только формой охватывающего пути. Такие рамки устанавливаются макросом `circleit`:

`circleit`⟨имя рамки⟩`(`⟨содержание рамки⟩`)`,

где ⟨имя рамки⟩ — это суффикс, а ⟨содержание рамки⟩ — это либо выражение-картинка, либо строковое выражение, либо ⟨пусто⟩.

Макрос `circleit` определяет переменные-пары также, как `boxit`, но без угловых точек ⟨имя рамки⟩`.ne`, ⟨имя рамки⟩`.sw` и т. д. Вызов

```
circleit.a(...)
```

задает отношения между точками `a.c`, `a.s`, `a.e`, `a.n`, `a.w` и расстояния `a.dx` и `a.dy`. Вместе с `a.c` и `a.off` эти переменные определяют, как картинка центрируется в овале, что можно увидеть на рис. 51.

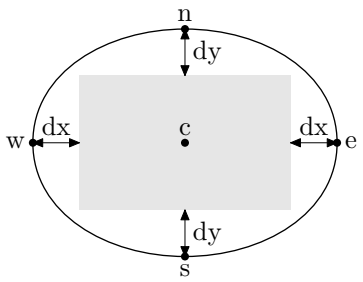


Рис. 51: Отношения между заданным `circleit` рисунком и связанными с этим переменными. Рисунок изображен серым прямоугольником.

Макросы `drawboxed`, `drawunboxed`, `drawboxes`, `pic` и `bpath` работают для рамок `circleit` также как и для рамок `boxit`.

Типовой охватывающий путь для рамки `circleit` достаточно велик для окружения содержимого рамки с маленькими предохраняющими отступами, управляемыми внутренней переменной `circmargin`. Рис. 52 предоставляет базовый пример использования `bpath` с рамками `circleit`.

Полный пример рамок `circleit` приводится на рис. 53. Пути овальной границы вокруг “Start” и “Stop” соответствуют уравнениям

`aa.dx=aa.dy;` и `ee.dx=ee.dy`

после

`circleit.ee(btex\strut Stop etex)` и `circleit.ee(btex\strut Stop etex)`.

Общее правило в том, что `bpath.`c выходит круглым, если все c`.dx`, c`.dy` и c`.dx` − c`.dy` неизвестны. Иначе макросы выбирают достаточно большой овал для вмещения данной картинки с предохраняющими отступами `circmargin`.

13 Файловые чтение и запись

Доступ к файлам — это одно из новых свойств, введенное в версию 0.60 языка MetaPost. Новый оператор

`readfrom` ⟨имя файла⟩

```
vardef drawshadowed(text t) =
  fixsize(t);
  forsuffixes s=t:
    fill bpath.s shifted (1pt,-1pt);
    unfill bpath.s;
    drawboxed(s);
  endfor
enddef;

beginfig(51)
circleit.a(btex Box 1 etex);
circleit.b(btex Box 2 etex);
b.n = a.s - (0,20pt);
drawshadowed(a,b);
drawarrow a.s -- b.n;
endfig;
```

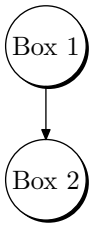


Рис. 52: Код MetaPost и рисунок-результат. Заметьте, что макрос `drawshadowed`, используемый здесь, не входит в макропакет `boxes.mp`.

возвращает очередную строку из именованного входного файла. ⟨Имя файла⟩ может быть любым первичным выражением типа строка. Если файл закончился или не может читаться, то результат чтения — это строка из одного нулевого символа. Макропакет `plain` вводит имя `EOF` для такой строки. После возвращения `EOF` от `readfrom`, следующие чтения этого же файла приведут к его повторному чтению сначала.

Все файлы, открытые `readfrom`, что еще не считаны полностью, закрываются автоматически, когда программа заканчивает выполнение, хотя существует команда

`closefrom` ⟨имя файла⟩

для явного закрытия файлов, открытых `readfrom`. Разумно явно закрывать файлы, которые не нужно считывать полностью, т. е. до получения `EOF`, потому что в противном случае такие файлы продолжат использовать внутренние ресурсы и возможно обусловят ошибку `capacity exceeded!`[18]

Противоположностью `readfrom` является команда

`write` ⟨строковое выражение⟩ `to` ⟨имя файла⟩

Она пишет строку текста в указанный файл вывода, открывая сначала файл, если это нужно. Все такие файлы закрываются автоматически, когда программа завершается. Они могут быть также закрыты явно использованием `EOF` как ⟨строкового выражения⟩. Единственный способ узнать была ли команда `write` успешной в закрытии файла и использовании `readfrom` для его просмотра.

14 Полезные средства

Этот раздел описывает некоторые полезные средства, включенные в каталог `mplib` иерархии разработки исходников. Будущие версии этой документации должны содержать больше информации, пока же, пожалуйста, читайте файлы исходников — большинство из них имеют объяснительные комментарии в начале. Исходники также включают в дистрибутивы как MetaPost, так и в бóльшие TEX, как правило, в каталог `texmf/metapost/base`.

[18] емкость превышена!

```
vardef cuta(suffix a,b) expr p =
  drawarrow p cutbefore bpath.a cutafter bpath.b;
  point .5*length p of p
enddef;

vardef self@# expr p =
  cuta(@#,@#) @#.c{curl0}..@#.c+p..{curl0}@#.c  enddef;

beginfig(52);
verbatimtex \def\stk#1#2{$\displaystyle{\matrix{#1\cr#2\cr}}$} etex
circleit.aa(btex\strut Start etex); aa.dx=aa.dy;
circleit.bb(btex \stk B{(a|b)^*a} etex);
circleit.cc(btex \stk C{b^*} etex);
circleit.dd(btex \stk D{(a|b)^*ab} etex);
circleit.ee(btex\strut Stop etex); ee.dx=ee.dy;
numeric hsep;
bb.c-aa.c = dd.c-bb.c = ee.c-dd.c = (hsep,0);
cc.c-bb.c = (0,.8hsep);
xpart(ee.e - aa.w) = 3.8in;
drawboxed(aa,bb,cc,dd,ee);
label.ulft(btex$b$etex, cuta(aa,cc) aa.c{dir50}..cc.c);
label.top(btex$b$etex, self.cc(0,30pt));
label.rt(btex$a$etex, cuta(cc,bb) cc.c..bb.c);
label.top(btex$a$etex, cuta(aa,bb) aa.c..bb.c);
label.llft(btex$a$etex, self.bb(-20pt,-35pt));
label.top(btex$b$etex, cuta(bb,dd) bb.c..dd.c);
label.top(btex$b$etex, cuta(dd,ee) dd.c..ee.c);
label.lrt(btex$a$etex, cuta(dd,bb) dd.c..{dir140}bb.c);
label.bot(btex$a$etex, cuta(ee,bb) ee.c..tension1.3 ..{dir115}bb.c);
label.urt(btex$b$etex, cuta(ee,cc) ee.c{(cc.c-ee.c)rotated-15}..cc.c);
endfig;
```

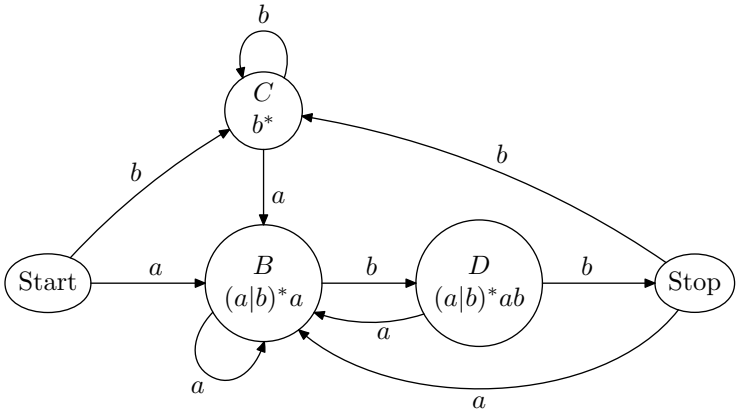


Рис. 53: Код MetaPost и соответствующий рисунок

14.1 TEX.mp

`TEX.mp` предоставляет способ печатать текст строковых выражений MetaPost. Предположим, например, что вам нужны метки в форме n_0, n_1, ..., n_{10} по оси x. Вы можете сделать их с (относительным) удобством с `TEX.mp`:

```
input TEX;
beginfig(100)
  last := 10;
  for i := 0 upto last:
   label(TEX("$n_{" & decimal(i) & "}$"), (5mm*i,0));
  endfor
  ...
endfig;
```

В отличии от этого, базовая команда `btex` (см. стр. 27) печатает текст буквально. Получается, что `btex s etex` печатает литеральный символ 's', а `TEX(s)` печатает значение текстовой переменной MetaPost s.

В версию 0.9 `TEX.mp` добавлены два дополнительных средства, позволяющие использовать LaTeX для печати меток: `TEXPRE` и `TEXPOST`. Их значения запоминаются и включаются соответственно перед и после каждого вызова `TEX`. Без них каждый вызов `TEX` печатает совершенно независимо. Вызовы `TEX` также не взаимодействуют с использованием `verbatimtex` (стр. 29).

Вот тот же самый пример, что и выше, но с использованием команд LaTeX `\(` и `\)`:

```
input TEX;
TEXPRE("%&latex" & char(10) & "\documentclass{article}\begin{document}");
TEXPOST("\end{document}");
beginfig(100)
  last := 10;
  for i := 0 upto last:
   label(TEX("\( n_{" & decimal(i) & "} \)"), (5mm*i,0));
  endfor
  ...
endfig;
```

Объяснения:

- `%&latex` приводит к вызову LaTeX вместо TeX (см. также ниже). Основанные на Web2C и MiKTeX дистрибутивы TeX, как минимум, понимают `%&` спецификацию; см., например, документацию Web2C для деталей, http://tug.org/web2c. (Информация о том, как делать это же в других системах будет весьма приветствоваться.)
- `char(10)` помещает маркер новой строки (десятичный код символа ASCII — 10) в вывод.
- `\documentclass...` — обычный способ начать документ LaTeX.
- Из-за поведения `mpto TEXPOST("\end{document}")` не является строго необходимым, но надежнее его включать.

К сожалению, инструкции TeX `\special` исчезают в этом процессе. Поэтому нельзя использовать пакеты, подобные `xcolor` и `hyperref`.

В случае, если вы любопытны, то средства `TEX.mp` реализуются очень просто: они пишут команды `btex` во временный файл и затем используют `scantokens` (стр. 19) для его обработки. Механизм `makempx` (стр. 30) выполняет всю работу по использованию TeX.

Магические %& в первой строке — это не единственный путь указать на вызов программы, отличной от (plain) TEX. Здесь проявляется преимущество максимума гибкости: разные конструкции TEX могут использовать разные процессоры TEX. Возможны не менее двух других способов:

- Установка переменный среды TEX в latex или в какой-угодно процессор, желаемый для вызова. (Для работы с фрагментами ConTEXt нужно вызывать texexec.) Этот способ удобен, когда пишется сценарий или идет работа над проектом, всегда требующим latex.
- Вызов MetaPost с опцией командной строки -tex=latex (или другим процессором, конечно). Это может быть полезно в Makefile или при единственном исполнении.

14.2 mproof.tex

mproof.tex — это средство (plain) TEX, а не MetaPost. Оно печатает гранки вывода MetaPost. Вызывайте его примерно так:

tex mproof *имя-выходного-файла-MetaPost*

Затем работайте с результирующим файлом dvi обычным образом.

15 Отладка

MetaPost унаследовал многие возможности METAFONT для интерактивной отладки, значительная часть которых может лишь вкратце быть упомянута здесь. Больше информации по сообщениям об ошибках, отладке, генерации трассирующей информации можно найти в *The METAFONTbook* [4].

Предположим, что ваш входной файл содержит в строке 17

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

без предварительного определения значений z1 и z2. Рис. 54 показывает то, что интерпретатор MetaPost печатает на вашем терминале, когда находит ошибку. Собственно сообщение об ошибке — это строка, начинающаяся с "!", следующие шесть строк дают контекст, точно показывающий, что читалось с ввода, когда была обнаружена ошибка, а "?" на последней строке — это приглашение для вашего ответа. Из-за того, что сообщение об ошибке говорит о неопределенной координате x, это значение печатается в первой строке после '>>". В этом случае координата x пары z1 — это неизвестная переменная x1, поэтому интерпретатор печатает имя переменной x1 точно также, как если бы в этом месте было сказано "show x1".

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

Рис. 54: Пример сообщения об ошибке.

Листинг контекста может показаться слегка путаным в первый раз, но он по-просту выдает несколько строк текста, показывая, как много из каждой строки было уже считано. Каждая строка ввода печатается в две строки, подобные следующим:

⟨дескриптор⟩ Уже считанный текст
Текст для считывания

⟨Дескриптор⟩ идентифицирует исходник. Это либо номер строки типа "`l.17`" для строки 17 текущего файла, либо имя макроса перед "`->`", либо объясняющая фраза в угловых скобках. Таким образом, значение листинга контекста на рис. 54: интерпретатор только что считал строку 17 входного файла до "`--`", раскрытие макроса `--` как раз началось и начальный "`{`" возвращен назад, позволяя ввод пользователя до разбора этого знака.

Среди возможных ответов на приглашение `?` есть такие:

x прекращает исполнение — вы можете исправить ваш входной файл и перезапустить MetaPost.

h печатает подсказку, за которой идет другое `?`-приглашение.

⟨**return**⟩ приводит интерпретатор к продолжению работы так хорошо, как он сможет.

? печатает список возможных опций, за которыми следует `?`-приглашение.

Сообщения об ошибках и ответы на команды `show` также печатаются и в файл-дубликат, чье имя получается из имени главного входного файла изменением "`.mp`" на "`.log`". Когда внутренняя переменная `tracingonline` имеет исходное значение ноль, то некоторые команды `show` печатают свои результаты во всех деталях только в файл-дубликат.

Только один тип команды `show` обсуждался до сих пор: `show` с разделенным запятыми списком выражений печатает символьное представление этих выражений.

Команда `showtoken` может быть использована для показа параметров и текста замены макроса. Она берет разделенный запятыми список знаков и идентифицирует каждый из них. Если знак — это примитив, например, "`showtoken +`", то он по-просту идентифицируется собой:

```
> +=+
```

Применение `showtoken` к переменной или `vardef`-макросу приведет к

```
> ⟨знак⟩=variable
```

Для получения дополнительной информации о переменной используйте `showvariable` вместо `showtoken`. Аргумент к `showvariable` — это разделенный запятыми список символических знаков, а ее результат — это описание всех переменных, чьи имена начинаются с одного из знаков из списка. Это работает даже для vardef-макросов. Например, `showvariable z` печатает

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

Есть еще команда `showdependencies`, не имеющая аргументов и печатающая список всех *зависимых* переменных и то, как линейные уравнения заданные ранее делают их зависимыми с другими переменными. Таким образом, после

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

`showdependencies` напечатает то, что показано на рис. 55. Это может быть полезным при ответе на вопрос типа "Что значит, что '`! Undefined x coordinate`?'[19] — я думаю, что приведенные ранее уравнения определяют `x1`."

[19]Координата x — неопределена

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

Рис. 55: Результат `z2-z1=(5,10); z1+z2=(a,b); showdependencies;`

Если все это не приводит к успеху, то есть еще предопределенный макрос `tracingall`, приводящий интерпретатор к печати детального отчета обо всем, что он делает. Вследствие того, что трассирующая информация часто имеет весьма большой размер, возможно будет лучше использовать макрос `loggingall`, производящий ту же самую информацию, но только для записи в файл-дубликат. Есть еще макрос `tracingnone`, отключающий все распечатки по трассировке.

Вывод трассировки контролируется множеством внутренних переменных, приводимых далее. Когда любой из этих переменных задают положительное значение, то этим устанавливается соответствующая форма трассировки. Здесь приведено множество переменных трассировки и то, что случается, когда каждая из них становится положительной.

`tracingcapsules` показывает значения временных количеств (капсул), когда они становятся известными.

`tracingchoices` показывает контрольные точки Безье на каждом пути, где они выбираются.

`tracingcommands` показывает команды перед их выполнением. Установка в > 1 также покажет проверки `if` и циклы до их раскрытия; установка в > 2 покажет алгебраические операции до их выполнения.

`tracingequations` показывает каждую переменную, когда она становится известной.

`tracinglostchars` предупреждает о символах, отсутствующих в картинке, из-за их отсутствия в шрифте, используемом для печати меток.

`tracingmacros` печатает макросы до их раскрытия.

`tracingoutput` печатает картинки при их отправке в PostScript-файлы.

`tracingrestores` показывает символы и внутренние переменные при их восстановлении в конце группы.

`tracingspecs` показывает выделения, генерируемые при рисовании многоугольным пером.

`tracingstats` в конце работы печатает в файл-дубликат о том, как много ограниченных ресурсов интерпретатора MetaPost было использовано.

Признательность

Я рад поблагодарить Дональда Кнута за возможность проделать эту работу — за развитие METAFONT и помещения его в открытый доступ. Я также в долгу перед ним за полезные предложения, особенно в связи с обработкой включаемого материала TEX.

A Справочное руководство

Таблицы 5–10 суммируют встроенные возможности Plain MetaPost и возможности, определенные в файле с макросами `boxes.mp`. Как объяснялось в разделе 12, файл с макросами `boxes.mp` не включается автоматически и макросы из него недоступны до тех пор, пока вы не запросите их командой

```
input boxes
```

Возможности, зависящие от `boxes.mp` отмечены символами ∗. Возможности из макропакета Plain отмечены символами †, а примитивы MetaPost приводятся без ∗ или †. Разница между примитивами и макросами из plain может игнорироваться случайным пользователем, но важно помнить, что возможности с меткой ∗ могут быть использованы только после чтения файла `boxes.mp` с макросами.

Таблицы в этом приложении приводят имя каждой возможности, номер страницы, где она объясняется, и краткое описание. Небольшое число свойств нигде не объяснялось и они не имеют номеров страниц. Эти возможности существуют, в основном, для совместимости с METAFONT и предполагаются самообъясняющими. Некоторые определенные возможности METAFONT полностью отсутствуют из-за ограниченного интереса к ним пользователей MetaPost и/или из-за требуемых длинных объяснений. Все они документированы в *The METAFONTbook* [4] как объяснено в приложении B.

Таблица 5 перечисляет внутренние переменные с числовыми значениями. Таблица 6 перечисляет предопределенные переменные других типов. Таблица 7 перечисляет предопределенные константы. Некоторые из них реализованы как переменные, чьи значения предполагается не менять.

Таблица 8 приводит операторы MetaPost и перечисляет возможные типы для аргументов и результата каждого из них. Пункт “–” для левого аргумента задает унарный оператор, а “–” для обоих задает оператор без аргументов. Операторы с параметрами-суффиксами не приводятся в этой таблице, потому что они обрабатываются как “похожие на функции макросы”.

Две последние таблицы — это таблица 9 для команд и таблица 10 для макросов, ведущих себя как функции или процедуры. Такие макросы берут списки аргументов в скобках и/или параметры-суффиксы и возвращают либо значение приведенного в таблице типа, либо ничего. Последний случай для макросов, ведущих себя как процедуры. Их возвращаемые значения приводится как “–”.

Картинки в этом приложении представляют синтаксис языка MetaPost, начиная с выражений на рисунках 56–58. Хотя правила иногда указывают типы для выражений, первичностей, вторичностей и третичностей, но отдельный синтаксис для ⟨числового выражения⟩, ⟨выражения-пары⟩ и т. п. не приводится. Своей простоте правила на рис. 59 обязаны этим отсутствием информации о типах. Информация о типах может быть найдена в таблицах 5–10.

Рисунки 60 и 61 содержат синтаксис программ MetaPost, включая команды и их обобщения. Они не упоминают циклов и проверок `if`, потому что эти конструкции не ведут себя как команды. Синтаксис на рисунках 56–62 применим к результатам раскрытия всех условных конструкций и циклов. Условные конструкции и циклы имеют синтаксис, но они работают практически с произвольными последовательностями знаков. Рис. 62 определяет условные конструкции через ⟨сбалансированные знаки⟩, а циклы через ⟨тело цикла⟩, где ⟨сбалансированные знаки⟩ — это любая последовательность знаков, сбалансированная относительно `if` и `fi`, а ⟨тело цикла⟩ — это последовательность знаков, сбалансированная относительно `for`, `forsuffixes`, `forever` и `endfor`.

Таблица 5: Внутренние переменные с числовыми значениями

Имя	Стр.	Описание
†ahangle	47	угол для наконечника стрелки в градусах (стандартно 45)
†ahlength	47	размер наконечника стрелки (стд. 4bp)
†bboxmargin	32	особый промежуток, допускаемый bbox (стд. 2bp)
charcode	51	номер текущей картинки
*circmargin	71	пустота вокруг содержимого круговой или овальной рамки
day	–	текущий день месяца
defaultcolormodel	–	начальная цветовая модель (стд. 5, rgb)
*defaultdx	68	обычный гориз. отступ вокруг содержимого рамки (стд. 3bp)
*defaultdy	68	обычный верт. отступ вокруг содержимого рамки (стд. 3bp)
†defaultpen	50	число, используемое pickup для выбора стд. пера
†defaultscale	27	масштабирующий множитель шрифта для строк-меток (стд. 1)
†labeloffset	26	расстояние отступа для меток (стд. 3bp)
linecap	45	0 для butt, 1 для round, 2 для square
linejoin	45	0 для mitered, 1 для round, 2 для beveled
miterlimit	45	контролирует длину острия как в PostScript
month	–	текущий месяц (например, 3 ≡ Март)
mpprocset	–	установите в 1, если хотите включить словарь сокращений PostScript в вывод
pausing	–	> 0 — показывать строки на терминале до их чтения
prologues	29	> 0 — выводить PostScript с встроенными шрифтами
restoreclipcolor	–	восстановление состояния графики после вырезки (стд. 1)
showstopping	–	> 0 — останавливать после каждой команды show
time	–	число минут после полуночи в начале этой работы
tracingcapsules	77	> 0 — показывать и капсулы
tracingchoices	77	> 0 — показывать контрольные точки для путей
tracingcommands	77	> 0 — показывать команды при их выполнении
tracingequations	77	> 0 — показывать каждую ставшую известной переменную
tracinglostchars	77	> 0 — показывать символы не из infont
tracingmacros	77	> 0 — показывать макросы до их раскрытия
tracingonline	18	> 0 — показывать длинные диагностики на терминале
tracingoutput	77	> 0 — показывать цифровые края при выводе
tracingrestores	77	> 0 — показывать переменные при их восстановлении
tracingspecs	77	> 0 — показывать деления пути (исп-ся многоугольное перо)
tracingstats	77	> 0 — показывать использование памяти в конце работы
tracingtitles	–	> 0 — показывать заголовки при их появлении
troffmode	29	будет 1, если есть опция -troff или -T
truecorners	32	> 0 — делать llcorner и т. д., игнорировать setbounds
warningcheck	19	сообщение об ошибке при большом значении переменной
year	–	текущий год (например, 1992)

Таблица 6: Другие предопределенные переменные

Имя	Тип	Стр.	Объяснение
†`background`	color	34	Цвет для `unfill` и `undraw` (обычно белый)
†`currentpen`	pen	51	Текущее перо (для команды `draw`)
†`currentpicture`	picture	50	Результат команд `draw` и `fill`
†`cuttings`	path	38	Подпуть, отрезанный последней `cutbefore` или `cutafter`
†`defaultfont`	string	27	Шрифт для команд печати строк
†`extra_beginfig`	string	98	Дополнительные команды для `beginfig`
†`extra_endfig`	string	98	Дополнительные команды для `endfig`

Таблица 7: Предопределенные константы

Имя	Тип	Стр.	Объяснение
†beveled	numeric	45	Значение linejoin для срезанных соединений [2]
†black	color	19	Эквивалентно (0,0,0)
†blue	color	19	Эквивалентно (0,0,1)
†bp	numeric	4	Один пункт PostScript в bp-единицах [1]
†butt	numeric	45	Значение linecap для butt-конца [0]
†cc	numeric	–	Одна единица цицеро в bp-единицах [12.79213]
†cm	numeric	4	Один сантиметр в bp-единицах [28.34645]
†dd	numeric	–	Один дидот в bp-единицах [1.06601]
†ditto	string	–	Строка " длины 1
†down	pair	13	Вектор вниз $(0,-1)$
†epsilon	numeric	–	Наименьшее положительное число MetaPost $[\frac{1}{65536}]$
†evenly	picture	42	Образец пунктира из тире и равных промежутков
†EOF	string	72	Одиночный нулевой символ
false	boolean	19	Логическое значение *false*
†fullcircle	path	34	Окружность диаметра 1 с центром в $(0,0)$
†green	color	19	Эквивалентно (0,1,0)
†halfcircle	path	34	Верхняя полуокружность диаметра 1
†identity	transform	41	Тождественная трансформация
†in	numeric	4	Один дюйм в bp-единицах [72]
†infinity	numeric	38	Наибольшее положительное значение [4095.99998]
†left	pair	13	Направление влево $(-1,0)$
†mitered	numeric	45	Значение linejoin для “острых” соединений [0]
†mm	numeric	4	Один миллиметр в bp-единицах [2.83464]
mpversion	string	3	Номер версии MetaPost
nullpen	pen	54	Пустое перо
nullpicture	picture	21	Пустая картинка
†origin	pair	–	Пара $(0,0)$
†pc	numeric	–	Одна пика в bp-единицах [11.95517]
pencircle	pen	49	Круговое перо диаметра 1
†pensquare	pen	50	Квадратное перо высоты и ширины 1
†pt	numeric	4	Один принтерный пункт в bp-единицах [0.99626]
†quartercircle	path	–	Первый квадрант окружности диаметра 1
†red	color	19	Эквивалентно (1,0,0)
†right	pair	13	Направление вправо $(1,0)$
†rounded	numeric	45	Значение для linecap и linejoin для круглых соединений и концов [1]
†squared	numeric	45	Значение linecap для квадратных концов [2]
true	boolean	19	Логическая величина true
†unitsquare	path	–	Путь (0,0)--(1,0)--(1,1)--(0,1)--cycle
†up	pair	13	Направление вверх $(0,1)$
†white	color	19	Эквивалентно (1,1,1)
†withdots	picture	42	Образец пунктира из точек

Таблица 8: Операторы

Имя	Аргумент/типы результата			Стр.	Объяснение
	Левый	Правый	Результат		
&	string path	string path	string path	21	Склейка (для путей l&r, если r начинается точно там, где кончается l)
*	numeric	(cmyk)color numeric pair	(cmyk)color numeric pair	20	Умножение
*	(cmyk)color numeric pair	numeric	(cmyk)color numeric pair	20	Умножение
**	numeric	numeric	numeric	20	Возведение в степень
+	(cmyk)color numeric pair	(cmyk)color numeric pair	(cmyk)color numeric pair	21	Сложение
++	numeric	numeric	numeric	21	Пифагорово сложение $\sqrt{l^2 + r^2}$
+-+	numeric	numeric	numeric	21	Пифагорово вычитание $\sqrt{l^2 - r^2}$
-	(cmyk)color numeric pair	(cmyk)color numeric pair	(cmyk)color numeric pair	21	Вычитание
-	–	(cmyk)color numeric pair	(cmyk)color numeric pair	20	Унарный минус
/	(cmyk)color numeric pair	numeric	(cmyk)color numeric pair	20	Деление
< = >> <= >= <>	string numeric pair (cmyk)color transform	string numeric pair (cmyk)color transform	boolean	19	Операции сравнения
†abs	–	numeric pair	numeric	22	Модуль Евклидова длина $\sqrt{(\texttt{xpart}\ r)^2 + (\texttt{ypart}\ r)^2}$
and	boolean	boolean	boolean	19	Логическое И
angle	–	pair	numeric	22	2−аргументный арктангенс (в градусах)
arclength	–	path	numeric	40	Длина дуги пути
arctime of	numeric	path	numeric	40	Время на пути, где длина дуги от начала достигает заданной величины
ASCII	–	string	numeric	–	Значение ASCII первого символа в строке
†bbox	–	picture path pen	path	32	Прямоугольный путь охватывающей рамки

Таблица 8: Операторы *(продолжение)*

Имя	Аргумент/типы результата			Стр.	Объяснение
	Левый	Правый	Результат		
`blackpart`	–	cmykcolor	numeric	23	Выделение четвертой компоненты
`bluepart`	–	color	numeric	23	Выделение третьей компоненты
`boolean`	–	любой	boolean	22	Выражение логического типа?
†`bot`	–	numeric pair	numeric pair	49	Низ текущего пера, центрированного по заданным координатам
`bounded`	–	любой	boolean	53	Аргумент — это картинка в охватывающей рамке?
†`ceiling`	–	numeric	numeric	22	Наименьшее целое, большее или равное данному
†`center`	–	picture path pen	pair	32	Центр охватывающей рамки
`char`	–	numeric	string	31	Символ с заданным кодом ASCII
`clipped`	–	любой	boolean	53	Аргумент — вырезка из картинки?
`cmykcolor`	–	любой	boolean	22	Выражение типа cmyk-цвет?
`color`	–	любой	boolean	22	Выражение типа цвет?
`colormodel`	–	изображение	numeric	53	Какая модель цвета в объекте-изображении?
†`colorpart`	–	изображение	(cmyk)color numeric boolean	53	Каков цвет объекта-изображения?
`cosd`	–	numeric	numeric	22	Косинус угла в градусах
†`cutafter`	path	path	path	38	Левый аргумент с частью, отбрасываемой после пересечения
†`cutbefore`	path	path	path	38	Левый аргумент с частью, отбрасываемой до пересечения
`cyanpart`	–	cmykcolor	numeric	23	Извлечь первый аргумент
`cycle`	–	path	boolean	22	Определяет цикличен ли путь
`dashpart`	–	picture	picture	53	Образец пунктира пути в рисуемой картинке
`decimal`	–	numeric	string	22	Десятичное представление
†`dir`	–	numeric	pair	11	$(\cos\theta, \sin\theta)$ по заданному θ в градусах
†`direction of`	numeric	path	pair	38	Направление пути в данное ‘время’
†`direction-point of`	pair	path	numeric	40	Точка, где путь имеет заданное направление

Таблица 8: Операторы *(продолжение)*

Имя	Аргумент/типы результата			Стр.	Объяснение
	Левый	Правый	Результат		
`direction-time of`	pair	path	numeric	38	‘Время’, когда путь имеет заданное направление
`†div`	numeric	numeric	numeric	–	Целочисленное деление $\lfloor l/r \rfloor$
`†dotprod`	pair	pair	numeric	21	скалярное произведение векторов
`filled`	–	любой	boolean	53	Аргумент — это заполненное выделение?
`floor`	–	numeric	numeric	22	Наибольшее целое, меньшее или равное данному
`fontpart`	–	picture	string	53	Шрифт текстовой компоненты картинки
`fontsize`	–	string	numeric	27	Размер шрифта в пунктах
`greenpart`	–	color	numeric	23	Выделить второй компонент
`greypart`	–	numeric	numeric	23	Выделить первый (единственный) компонент
`hex`	–	string	numeric	–	Интерпретировать как 16-ричное число
`infont`	string	string	picture	31	Печать строку в заданном шрифте
`†intersectionpoint`	path	path	pair	37	Точка пересечения
`intersectiontimes`	path	path	pair	37	Времена (t_l, t_r) на путях l и r, когда пути пересекаются
`†inverse`	–	transform	transform	41	Обратить трансформацию
`known`	–	любой	boolean	22	Имеет ли аргумент известное значение?
`length`	–	path string picture	numeric	38 21 52	Число компонент (дуг, символов, нарисованных объектов, . . .) в аргументе
`†lft`	–	numeric pair	numeric pair	49	Левый край текущего пера с центром по заданным координатам
`llcorner`	–	picture path pen	pair	32	Нижний левый угол охватывающей рамки
`lrcorner`	–	picture path pen	pair	32	Нижний правый угол охватывающей рамки
`magentapart`	–	cmykcolor	numeric	23	Извлечь второй компонент
`makepath`	–	pen	path	50	Замкнутый путь, охватывающий форму пера
`makepen`	–	path	pen	50	Многоугольное перо из выпуклой части узлов пути
`mexp`	–	numeric	numeric	–	Функция $\exp(x/256)$
`mlog`	–	numeric	numeric	–	Функция $256\ln(x)$
`†mod`	–	numeric	numeric	–	Функция-остаток $l - r\lfloor l/r \rfloor$

Таблица 8: Операторы *(продолжение)*

Имя	Аргумент/типы результата			Стр.	Объяснение
	Левый	Правый	Результат		
normal-deviate	–	–	numeric	–	Выбор случайного числа со средним 0 и стандартным отклонением 1
not	–	boolean	boolean	19	Логическое НЕ
numeric	–	любой	boolean	22	Выражение числового типа?
oct	–	string	numeric	–	Интерпретировать строку как 8-ричное число
odd	–	numeric	boolean	–	Ближайшее целое нечетное?
or	boolean	boolean	boolean	19	Логическое ИЛИ
pair	–	любой	boolean	22	Выражение типа пара?
path	–	любой	boolean	22	Выражение типа путь?
pathpart	–	picture	path	53	Компонент-путь нарисованной картинки
pen	–	любой	boolean	22	Выражение типа перо?
penoffset of	pair	pen	pair	–	Крайняя точка пера с заданным направлением
penpart	–	picture	pen	53	Компонента-перо нарисованной картинки
picture	–	любой	boolean	22	Выражение типа картинка?
point of	numeric	path	pair	37	Точка на пути с заданным значением времени
postcontrol of	numeric	path	pair	–	Первая управляющая точка Безье на отрезке пути, начинающимся в данное время
precontrol of	numeric	path	pair	–	Последняя управляющая точка Безье на отрезке пути, заканчивающимся в данное время
readfrom	–	string	string	72	Читать строку из файла
redpart	–	color	numeric	23	Выделить первый компонент
reverse	–	path	path	47	путь в обратном ‘времени’, конец меняется с началом
rgbcolor	–	любой	boolean	22	Выражение типа цвет?
rotated	picture path pair pen transform	numeric	picture path pair pen transform	40	Вращение (в градусах) против часовой стрелки
†round	–	numeric pair	numeric pair	22	округлить каждую компоненту до ближайшего целого
†rt	–	numeric pair	numeric pair	49	Правая сторона текущего пера, центрированного по данным координатам

Таблица 8: Операторы *(продолжение)*

Имя	Аргумент/типы результата			Стр.	Объяснение
	Левый	Правый	Результат		
`scaled`	picture path pair pen transform	numeric	picture path pair pen transform	40	Масштабируй все координаты в заданное число раз
`scantokens`	–	string	token sequence	19	Преобрази строку в знак или последовательность знаков. Обеспечивает преобразование строки в число и т. п.
`shifted`	picture path pair pen transform	pair	picture path pair pen transform	40	Добавляет заданный сдвиг к каждой паре координат
`sind`	–	numeric	numeric	22	Синус угла в градусах
`slanted`	picture path pair pen transform	numeric	picture path pair pen transform	40	Применение трансформации-наклона, переводящей (x, y) в $(x + sy, y)$, где s — аргумент-число
`sqrt`	–	numeric	numeric	22	Квадратный корень
`str`	–	suffix	string	63	Строковое представление суффикса
`string`	–	любой	boolean	22	Выражение типа строка?
`stroked`	–	любой	boolean	53	Аргумент — это нарисованная линия?
`subpath of`	pair	path	path	38	Часть пути для заданного диапазона времени
`substring of`	pair	string	string	21	Подстрока, ограниченная индексами
`textpart`	–	picture	string	53	Текст текстовой компоненты картинки
`textual`	–	любой	boolean	53	Аргумент — это текст?
†`top`	–	numeric pair	numeric pair	49	Верх текущего пера, центрированного по заданным координатам
`transform`	–	любой	boolean	22	Аргумент типа трансформация?
`transformed`	picture path pair pen transform	transform	picture path pair pen transform	41	Примени данную трансформацию ко всем координатам
`ulcorner`	–	picture path pen	pair	32	Верхний левый угол охватывающей рамки

Таблица 8: Операторы *(продолжение)*

Имя	Аргумент/типы результата			Стр.	Объяснение
	Левый	Правый	Результат		
`uniform-deviate`	–	numeric	numeric	–	Случайное число от нуля до значения аргумента
†`unitvector`	–	pair	pair	22	Масштабируй вектор к длине 1
`unknown`	–	любой	boolean	22	Значение неизвестно?
`urcorner`	–	picture path pen	pair	32	Верхний правый угол охватывающей рамки
†`whatever`	–	–	numeric	17	Создай новую анонимную неизвестную
`xpart`	–	pair transform	number	23	x или t_x компонента
`xscaled`	picture path pair pen transform	numeric	picture path pair pen transform	40	Масштабируй все координаты x в заданное число раз
`xxpart`	–	transform	number	42	t_{xx} в матрице трансформации
`xypart`	–	transform	number	42	t_{xy} в матрице трансформации
`yellowpart`	–	cmykcolor	numeric	23	Выделить третью компоненту
`ypart`	–	pair transform	number	23	Компонента y или t_y
`yscaled`	picture path pair pen transform	numeric	picture path pair pen transform	40	Масштабируй все координаты y в заданное число раз
`yxpart`	–	transform	number	42	t_{yx} в матрице трансформации
`yypart`	–	transform	number	42	t_{yy} в матрице трансформации
`zscaled`	picture path pair pen transform	pair	picture path pair pen transform	40	Вращать и масштабировать все координаты так, что $(1, 0)$ становится заданной парой, т. е. произвести комплексное умножение.

Таблица 9: Команды

Имя	Стр.	Объяснение
addto	50	Низкоуровневая команда для рисования и заполнения
clip	51	Применяет путь вырезки к картинке
closefrom	72	Закрывает файл, открытый readfrom
†cutdraw	65	Рисовать с butt-концом
dashed	42	Применять образец пунктира в команде рисования
†draw	9	Рисовать линию или картинку
†drawarrow	47	Рисовать линию со стрелкой на конце
†drawdblarrow	47	Рисовать линию со стрелками в обоих концах
filenametemplate	8	Установить шаблон имени выходного файла
†fill	33	Заполнить циклический путь
†filldraw	47	Рисовать циклический путь и заполнить его внутри
interim	56	Сделать локальное изменение внутренней переменной
let	–	Назначить символическому знаку значение другого знака
†loggingall	77	Включить трассировку (только для файла-журнала)
newinternal	25	Объявить новые внутренние переменные
†pickup	20	Указать новое перо для рисования линии
save	55	Делает переменные локальными
setbounds	32	Устанавливает охватывающую рамку для картинки
shipout	51	Низкоуровневая команда печати рисунка
show	18	Печать выражений в символической форме
showdependencies	76	Печать всех нерешенных уравнений
showtoken	76	Печать информации по знаку
showvariable	76	Печать переменных в символьной форме
special	98	Печать строки прямо в PostScript-файл вывода
†tracingall	77	Включить трассировку
†tracingnone	77	Отключить трассировку
†undraw	48	Стереть линию или рисунок
†unfill	34	Стереть внутри замкнутого пути
†unfilldraw	48	Стереть циклический путь и все внутри него
withcmykcolor	33	Используй CMYK-цвет в команде рисования
withcolor	33	Используй обычный цвет в команде рисования
withgreyscale	33	Используй оттенок серого в команде рисования
withoutcolor	33	Не используй спецификации цвета в команде рисования
withpen	49	Используй перо в команде рисования
withpostscript	45	Конец кода PostScript
withprescript	45	Начало кода PostScript
withrgbcolor	33	Используй RGB-цвет в команде рисования
write to	72	Писать строку в файл

Таблица 10: Макросы, похожие на функции

Имя	Аргументы	Резу-т	С.	Объяснение
*boxit	суффикс, картинка	–	67	Задает рамку, содержащую картинку
*boxit	суффикс, строка	–	69	Определяет рамку, содержащую текст
*boxit	суффикс, ⟨пусто⟩	–	69	Определяет пустую рамку
*boxjoin	уравнения	–	68	Задает уравнения для соединяемых рамок
*bpath	суффикс	path	68	Охватывающий круг или прямоугольник
†buildcycle	список путей	path	35	Строить замкнутый путь
*circleit	суффикс, картинка	–	71	Поместить картинку в круговую рамку
*circleit	суффикс, картинка	–	71	Поместить строку в круговую рамку
*circleit	суффикс, ⟨пусто⟩	–	71	Определить пустую круговую рамку
†dashpattern	расстояния вкл./выкл.	picture	44	Создать образец пунктирных линий
†decr	числовая переменная	numeric	64	Уменьшить и возвратить новое значение
†dotlabel	суффикс, картинка, пара	–	26	Нарисовать точку и рядом картинку
†dotlabel	суффикс, строка, пара	–	26	Отметить точку и поместить рядом текст
†dotlabels	суффикс, номера точек	–	26	Отметить точки z их номерами
*drawboxed	список суффиксов	–	68	Нарисовать именованные рамки и их содержимое
*drawboxes	список суффиксов	–	69	Нарисовать именованные рамки
†drawdot	пара	–	4	Поставить точку в данном месте
†drawoptions	опции рисования	–	48	Установить опции для команд рисования
*drawunboxed	список суффиксов	–	69	Рисовать содержимое именованных рамок
*fixpos	список суффиксов	–	69	Найти размер и позицию именованных рамок
*fixsize	список суффиксов	–	69	Найти размер именованных рамок
†image	строка	picture	52	Возвращает рисунок из текста
†incr	числовая переменная	numeric	64	Увеличить и возвратить новое значение
†label	суффикс, картинка, пара	–	25	Изобразить рис. возле заданной точки
†label	суффикс, строка, пара	–	25	Поместить текст возле заданной точки
†labels	суффикс, номера точек	–	27	Нарисовать числа пар z, без точек
†max	список чисел	numeric	–	Найти максимум
†max	список строк	string	–	Найти словарно последнюю строку
†min	список чисел	numeric	–	Найти минимум
†min	список строк	string	–	Найти словарно первую строку
*pic	суффикс	picture	69	Содержимое рамки, сдвинутое в позицию
†thelabel	суффикс, картинка, пара	picture	26	Картинка, сдвинутая как для метки точки
†thelabel	суффикс, строка, пара	picture	26	Текст, размещенный как для метки точки
†z	суффикс	pair	25	Пара x⟨суффикс⟩, y⟨суффикс⟩)

⟨атом⟩ → ⟨переменная⟩ | ⟨аргумент⟩
| ⟨число или дробь⟩
| ⟨внутренняя переменная⟩
| `(`⟨выражение⟩`)`
| `begingroup`⟨список команд⟩⟨выражение⟩`endgroup`
| ⟨оператор 0-уровня⟩
| `btex`⟨команды печати⟩`etex`
| ⟨псевдофункция⟩

⟨первичность⟩ → ⟨атом⟩
| `(`⟨числовое выражение⟩`,`⟨числовое выражение⟩`)`
| `(`⟨числовое выражение⟩`,`⟨числовое выражение⟩`,`⟨числовое выражение⟩`)`
| ⟨of-оператор⟩⟨выражение⟩`of`⟨первичность⟩
| ⟨унарный оператор⟩⟨первичность⟩
| `str`⟨суффикс⟩
| `z`⟨суффикс⟩
| ⟨числовой атом⟩`[`⟨выражение⟩`,`⟨выражение⟩`]`
| ⟨операция скалярного умножения⟩⟨первичность⟩

⟨вторичность⟩ → ⟨первичность⟩
| ⟨вторичность⟩⟨первичный бинарный оператор⟩⟨первичность⟩
| ⟨вторичность⟩⟨трансформация⟩

⟨третичность⟩ → ⟨вторичность⟩
| ⟨третичность⟩⟨вторичный бинарный оператор⟩⟨вторичность⟩

⟨подвыражение⟩ → ⟨третичность⟩
| ⟨выражение-путь⟩⟨соединение путей⟩⟨узел пути⟩

⟨выражение⟩ → ⟨подвыражение⟩
| ⟨выражение⟩⟨третичный бинарный оператор⟩⟨третичность⟩
| ⟨подвыражение пути⟩⟨указатель направления⟩
| ⟨подвыражение пути⟩⟨соединение путей⟩`cycle`

⟨узел пути⟩ → ⟨третичность⟩

⟨соединение путей⟩ → −−
| ⟨указатель направления⟩⟨базовое соединение путей⟩⟨указатель направления⟩

⟨указатель направления⟩ → ⟨пусто⟩
| `{curl`⟨числовое выражение⟩`}`
| `{`⟨выражение-пара⟩`}`
| `{`⟨числовое выражение⟩`,`⟨числовое выражение⟩`}`

⟨базовое соединение путей⟩ → `..` | `...` | `..`⟨напряжение⟩`..` | `..`⟨управление⟩`..`

⟨напряжение⟩ → `tension`⟨числовая первичность⟩
| `tension`⟨числовая первичность⟩`and`⟨числовая первичность⟩

⟨управление⟩ → `controls`⟨первичность-пара⟩
| `controls`⟨первичность-пара⟩`and`⟨первичность-пара⟩

⟨аргумент⟩ → ⟨символически знак⟩

⟨число или дробь⟩ → ⟨число⟩`/`⟨число⟩
| ⟨число, за которым нет ‘`/`⟨число⟩’⟩

⟨операция скалярного умножения⟩ → + | −
| ⟨‘⟨число или дробь⟩’, за которым нет ‘⟨операции сложения⟩⟨число⟩’⟩

Рис. 56: Часть 1 синтаксиса выражений

⟨трансформация⟩ → `rotated`⟨числовая первичность⟩
| `scaled`⟨числовая первичность⟩
| `shifted`⟨первичность-пара⟩
| `slanted`⟨числовая первичность⟩
| `transformed`⟨трансформация-первичность⟩
| `xscaled`⟨числовая первичность⟩
| `yscaled`⟨числовая первичность⟩
| `zscaled`⟨первичность-пара⟩
| `reflectedabout(`⟨выражение-пара⟩`,`⟨выражение-пара⟩`)`
| `rotatedaround(`⟨выражение-пара⟩`,`⟨выражение-пара⟩`)`

⟨оператор 0-уровня⟩ → `false` | `normaldeviate` | `nullpen` | `nullpicture` | `pencircle`
| `true` | `whatever`
⟨унарный оператор⟩ → ⟨тип⟩
| `abs` | `angle` | `arclength` | `ASCII` | `bbox` | `blackpart` | `bluepart` | `bot` | `bounded`
| `ceiling` | `center` | `char` | `clipped` | `colormodel` | `cosd` | `cyanpart` | `cycle`
| `dashpart` | `decimal` | `dir` | `floor` | `filled` | `fontpart` | `fontsize`
| `greenpart` | `greypart` | `hex` | `inverse` | `known` | `length` | `lft` | `llcorner`
| `lrcorner` | `magentapart` | `makepath` | `makepen` | `mexp` | `mlog` | `not` | `oct` | `odd`
| `pathpart` | `penpart` | `readfrom` | `redpart` | `reverse` | `round` | `rt` | `sind` | `sqrt`
| `stroked` | `textpart` | `textual` | `top` | `ulcorner`
| `uniformdeviate` | `unitvector` | `unknown` | `urcorner` | `xpart` | `xxpart`
| `xypart` | `yellowpart` | `ypart` | `yxpart` | `yypart`
⟨тип⟩ → `boolean` | `cmykcolor` | `color` | `numeric` | `pair`
| `path` | `pen` | `picture` | `rgbcolor` | `string` | `transform`
⟨первичный бинарный оператор⟩ → `*` | `/` | `**` | `and`
| `dotprod` | `div` | `infont` | `mod`
⟨вторичный бинарный оператор⟩ → + | − | ++ | +−+ | `or`
| `intersectionpoint` | `intersectiontimes`
⟨третичный бинарный оператор⟩ → `&` | `<` | `<=` | `<>` | `=` | `>` | `>=`
| `cutafter` | `cutbefore`
⟨of-оператор⟩ → `arctime` | `direction` | `directiontime` | `directionpoint`
| `penoffset` | `point` | `postcontrol` | `precontrol` | `subpath`
| `substring`

⟨переменная⟩ → ⟨этикетка⟩⟨суффикс⟩
⟨суффикс⟩ → ⟨пусто⟩ | ⟨суффикс⟩⟨индекс⟩ | ⟨суффикс⟩⟨этикетка⟩
| ⟨параметр-суффикс⟩
⟨индекс⟩ → ⟨число⟩ | `[`⟨числовое выражение⟩`]`

⟨внутренняя переменная⟩ → `ahangle` | `ahlength` | `bboxmargin`
| `charcode` | `day` | `defaultcolormodel` | `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`
| ⟨символический знак, определенный `newinternal`⟩

Рис. 57: Часть 2 синтаксиса выражений

⟨псевдофункция⟩ → `min(`⟨список выражений⟩`)`
| `max(`⟨список выражений⟩`)`
| `incr(`⟨числовая переменная⟩`)`
| `decr(`⟨числовая переменная⟩`)`
| `dashpattern(`⟨список on/off⟩`)`
| `interpath(`⟨числовое выражение⟩`,`⟨выражение-путь⟩`,`⟨выражение-путь⟩`)`
| `buildcycle(`⟨список выражений-путей⟩`)`
| `thelabel`⟨суффикс метки⟩`(`⟨выражение⟩`,`⟨выражение-пара⟩`)`
⟨список выражений-путей⟩ → ⟨выражение-путь⟩
| ⟨список выражений-путей⟩`,`⟨выражение-путь⟩
⟨список on/off⟩ → ⟨список on/off⟩⟨пункт on/off⟩ | ⟨пункт on/off⟩
⟨пункт on/off⟩ → `on`⟨числовая третичность⟩ | `off`⟨числовая третичность⟩

Рис. 58: Синтаксис макросов, похожих на функции

⟨логическое выражение⟩ → ⟨выражение⟩
⟨выражение-cmyk-цвет ⟩ → ⟨выражение⟩
⟨выражение-цвет⟩ → ⟨выражение⟩
⟨числовой атом⟩ → ⟨атом⟩
⟨числовое выражение⟩ → ⟨выражение⟩
⟨числовая первичность⟩ → ⟨первичность⟩
⟨числовая третичность⟩ → ⟨третичность⟩
⟨числовая переменная⟩ → ⟨переменная⟩ | ⟨внутренняя переменная⟩
⟨выражение-пара⟩ → ⟨выражение⟩
⟨пара-первичность⟩ → ⟨первичность⟩
⟨выражение-путь⟩ → ⟨выражение⟩
⟨подвыражение-путь⟩ → ⟨подвыражение⟩
⟨выражение-перо⟩ → ⟨выражение⟩
⟨выражение-картинка⟩ → ⟨выражение⟩
⟨переменная-картинка⟩ → ⟨переменная⟩
⟨выражение-rgb-цвет⟩ → ⟨выражение⟩
⟨строковое выражение⟩ → ⟨выражение⟩
⟨параметр-суффикс⟩ → ⟨параметр⟩
⟨трансформация-первичность⟩ → ⟨первичность⟩

Рис. 59: Различные правила, нужные для завершения НФБН

⟨программа⟩ → ⟨список метакоманд⟩`end`
⟨список метакоманд⟩ → ⟨пусто⟩ | ⟨список метакоманд⟩`;`⟨метакоманда⟩
⟨метакоманда⟩ → ⟨пусто⟩
| ⟨уравнение⟩ | ⟨присваивание⟩
| ⟨декларация⟩ | ⟨определение макроса⟩
| ⟨блок⟩ | ⟨псевдопроцедура⟩
| ⟨команда⟩
⟨блок⟩ → `begingroup`⟨список метакоманд⟩`endgroup`
| `beginfig(`⟨числовое выражение⟩`);`⟨список метакоманд⟩`;endfig`

⟨уравнение⟩ → ⟨выражение⟩`=`⟨правая часть⟩
⟨присваивание⟩ → ⟨переменная⟩`:=`⟨правая часть⟩
| ⟨внутренняя переменная⟩`:=`⟨правая часть⟩
⟨правая часть⟩ → ⟨выражение⟩ | ⟨уравнение⟩ | ⟨присваивание⟩

⟨декларация⟩ → ⟨тип⟩⟨список декларации⟩
⟨список декларации⟩ → ⟨обобщение переменной⟩
| ⟨список декларации⟩`,`⟨обобщение переменной⟩
⟨обобщение переменной⟩ → ⟨символический знак⟩⟨обобщение суффикса⟩
⟨обобщение суффикса⟩ → ⟨пусто⟩ | ⟨обобщение суффикса⟩⟨этикетка⟩
| ⟨обобщение суффикса⟩`[]`

⟨определение макроса⟩ → ⟨заголовок макроса⟩`=`⟨текст замены⟩`enddef`
⟨заголовок макроса⟩ → `def`⟨символический знак⟩⟨отделенная часть⟩⟨неотделенная часть⟩
| `vardef`⟨обобщение переменной⟩⟨отделенная часть⟩⟨неотделенная часть⟩
| `vardef`⟨обобщение переменной⟩`@#`⟨отделенная часть⟩⟨неотделенная часть⟩
| ⟨определение бинарности⟩⟨параметр⟩⟨символический знак⟩⟨параметр⟩
⟨отделенная часть⟩ → ⟨пусто⟩
| ⟨отделенная часть⟩`(`⟨тип параметра⟩⟨параметры-знаки⟩`)`
⟨тип параметра⟩ → `expr` | `suffix` | `text`
⟨параметры-знаки⟩ → ⟨параметр⟩ | ⟨параметры-знаки⟩`,`⟨параметр⟩
⟨параметр⟩ → ⟨символический знак⟩
⟨неотделенная часть⟩ → ⟨пусто⟩
| ⟨тип параметра⟩⟨параметр⟩
| ⟨уровень приоритета⟩⟨параметр⟩
| `expr`⟨параметр⟩`of`⟨параметр⟩
⟨уровень приоритета⟩ → `primary` | `secondary` | `tertiary`
⟨определение бинарности⟩ → `primarydef` | `secondarydef` | `tertiarydef`

⟨псевдопроцедура⟩ → `drawoptions(`⟨список опций⟩`)`
| `label`⟨суффикс метки⟩`(`⟨выражение⟩`,`⟨выражение-пара⟩`)`
| `dotlabel`⟨суффикс метки⟩`(`⟨выражение⟩`,`⟨выражение-пара⟩`)`
| `labels`⟨суффикс метки⟩`(`⟨список номеров точек⟩`)`
| `dotlabels`⟨суффикс метки⟩`(`⟨список номеров точек⟩`)`
⟨список номеров точек⟩ → ⟨суффикс⟩ | ⟨список номеров точек⟩`,`⟨суффикс⟩
⟨суффикс метки⟩ → ⟨пусто⟩ | `lft` | `rt` | `top` | `bot` | `ulft` | `urt` | `llft` | `lrt`

Рис. 60: Полный синтаксис программ MetaPost

⟨команда⟩ → `clip`⟨переменная-картинка⟩`to`⟨выражение-путь⟩
| `interim`⟨внутренняя переменная⟩`:=`⟨правая часть⟩
| `let`⟨символический знак⟩`=`⟨символический знак⟩
| `newinternal`⟨список символических знаков⟩
| `pickup`⟨выражение⟩
| `randomseed:=`⟨числовое выражение⟩
| `save`⟨список символических знаков⟩
| `setbounds`⟨переменная-картинка⟩`to`⟨выражение-путь⟩
| `shipout`⟨выражение-картинка⟩
| `special`⟨строковое выражение⟩
| `write`⟨строковое выражение⟩`to`⟨строковое выражение⟩
| ⟨команда addto⟩
| ⟨команда рисования⟩
| ⟨команда метрики шрифта⟩
| ⟨команда-сообщение⟩
| ⟨команда show⟩
| ⟨команда трассировки⟩

⟨команда show⟩ → `show`⟨список выражений⟩
| `showvariable`⟨список символических знаков⟩
| `showtoken`⟨список символических знаков⟩
| `showdependencies`

⟨список символических знаков⟩ → ⟨символический знак⟩
| ⟨символический знак⟩`,`⟨список символических знаков⟩
⟨список выражений⟩ → ⟨выражение⟩ | ⟨список выражений⟩`,`⟨выражение⟩

⟨команда addto⟩ →
`addto`⟨переменная-картинка⟩`also`⟨выражение-картинка⟩⟨список опций⟩
| `addto`⟨переменная-картинка⟩`contour`⟨выражение-путь⟩⟨список опций⟩
| `addto`⟨переменная-картинка⟩`doublepath`⟨выражение-путь⟩⟨список опций⟩
⟨список опций⟩ → ⟨пусто⟩ | ⟨опция рисования⟩⟨список опций⟩
⟨опция рисования⟩ → `withcolor`⟨выражение-цвет⟩
| `withrgbcolor`⟨выражение-rgb-цвет⟩ | `withcmykcolor`⟨выражение-cmyk-цвет⟩
| `withgreyscale`⟨числовое выражение⟩ | `withoutcolor`
| `withprescript`⟨строковое выражение⟩ | `withpostscript`⟨строковое выражение⟩
| `withpen`⟨выражение-перо⟩ | `dashed`⟨выражение-картинка⟩

⟨команда рисования⟩ → `draw`⟨выражение-картинка⟩⟨список опций⟩
| ⟨тип заполнения⟩⟨выражение-путь⟩⟨список опций⟩
⟨тип заполнения⟩ → `fill` | `draw` | `filldraw` | `unfill` | `undraw` | `unfilldraw`
| `drawarrow` | `drawdblarrow` | `cutdraw`

⟨команда-сообщение⟩ → `errhelp`⟨строковое выражение⟩
| `errmessage`⟨строковое выражение⟩
| `filenametemplate`⟨строковое выражение⟩
| `message`⟨строковое выражение⟩

⟨команда трассировки⟩ → `tracingall` | `loggingall` | `tracingnone`

Рис. 61: Синтаксис команд

⟨проверка if⟩ → `if`⟨логическое выражение⟩`:`⟨сбалансированные знаки⟩⟨альтернативы⟩`fi`
⟨альтернативы⟩ → ⟨пусто⟩
| `else:`⟨сбалансированные знаки⟩
| `elseif`⟨логическое выражение⟩`:`⟨сбалансированные знаки⟩⟨альтернативы⟩

⟨цикл⟩ → ⟨заголовок цикла⟩`:`⟨тело цикла⟩`endfor`
⟨заголовок цикла⟩ → `for`⟨символический знак⟩`=`⟨прогрессия⟩
| `for`⟨символический знак⟩`=`⟨список for⟩
| `for`⟨символический знак⟩`within`⟨выражение-картинка⟩
| `forsuffixes`⟨символический знак⟩`=`⟨список суффиксов⟩
| `forever`
⟨прогрессия⟩ → ⟨числовое выражение⟩`upto`⟨числовое выражение⟩
| ⟨числовое выражение⟩`downto`⟨числовое выражение⟩
| ⟨числовое выражение⟩`step`⟨числовое выражение⟩`until`⟨числовое выражение⟩
⟨список for⟩ → ⟨выражение⟩ | ⟨список for⟩`,`⟨выражение⟩
⟨список суффиксов⟩ → ⟨суффикс⟩ | ⟨список суффиксов⟩`,`⟨суффикс⟩

Рис. 62: Синтаксис для условий и циклов

B MetaPost и METAFONT

Из-за того, что языки METAFONT и MetaPost имеют так много общего, пользователи-эксперты METAFONT возможно захотят пропустить большую часть объяснений из этого документа и сконцентрироваться на уникальных концепциях MetaPost. Сравнения в этом приложение приводятся для помощи экспертам, хорошо знакомым с *The METAFONTbook*, а также другим пользователям, что хотят получить пользу из более детальных объяснений работы Кнута [4].

Вследствие того, что METAFONT предназначен для изготовления шрифтов TEX, он имеет ряд примитивов для генерации `tfm`-файлов, нужных TEX для измерения символов, информации об отступах, лигатурах и кернинге. MetaPost может также использоваться для генерации шрифтов и он также имеет примитивы METAFONT для создания `tfm`-файлов. Они перечисляются в таблице 11. Их объяснения могут быть найдены в документации METAFONT [4, 7].

команды	`charlist`, `extensible`, `fontdimen`, `headerbyte` `kern`, `ligtable`
операторы лигатурных таблиц	`::`, `=:`, `=:|`, `=:|>`, `|=:`, `|=:>`, `|=:|`, `|=:|>`, `|=:|>>`, `||:`
внутренние переменные	`boundarychar`, `chardp`, `charext`, `charht`, `charic`, `charwd`, `designsize`, `fontmaking`
другие операторы	`charexists`

Таблица 11: Примитивы MetaPost для создания `tfm`-файлов.

Даже хотя MetaPost имеет примитивы для генерации шрифтов, многие такие примитивы и внутренние переменные, которые входят в Plain METAFONT, не определены в Plain MetaPost. Вместо этого имеется отдельный макропакет `mfplain`, определяющий макросы, требуемые для возможности обработать через MetaPost шрифты Computer Modern Кнута, — они показаны в таблице 12 [6]. Для загрузки этих макросов поставьте "`&mfplain`" перед именем входного файла. Это может быть сделано по приглашению `**` после вызова интерпретатора MetaPost без аргументов или в командной строке, например[20],

```
mpost '&mfplain' cmr10
```

Командой, аналогичной METAFONT-команде

```
mf '\mode=lowres; mag=1.2; input cmr10',
```

будет

```
mpost '&mfplain \mode=lowres; mag=1.2; input cmr10'
```

Результат — это множество файлов PostScript, по одному на каждый символ шрифта. Потребуется некоторое редактирование для их соединения в загружаемый PostScript Type 3 шрифт [1]..

Другим ограничением пакета `mfplain` является то, что определенные переменные из Plain METAFONT не могут получить разумных определений в MetaPost. Среди них `displaying`, `currentwindow`, `screen_rows` и `screen_cols`, которые зависят от способности METAFONT изображать рисунки на экране компьютера. Кроме того, `pixels_per_inch` является неуместным из-за того, что MetaPost использует фиксированные единицы пунктов PostScript.

Основания того, что некоторые макросы и внутренние переменные ничего не значат в MetaPost в том, что примитивные команды METAFONT `cull`, `display`, `openwindow`, `numspecial` и `totalweight` не реализованы в MetaPost. Также не реализованы ряд внутренних переменных

[20] Синтаксис командной строки зависит от системы. Кавычки нужны в большинстве систем Unix для защиты специальных символов, подобных `&`.

Определены в пакете mfplain	
beginchar	font_identifier
blacker	font_normal_shrink
capsule_def	font_normal_space
change_width	font_normal_stretch
define_blacker_pixels	font_quad
define_corrected_pixels	font_size
define_good_x_pixels	font_slant
define_good_y_pixels	font_x_height
define_horizontal_corrected_pixels	italcorr
define_pixels	labelfont
define_whole_blacker_pixels	makebox
define_whole_pixels	makegrid
define_whole_vertical_blacker_pixels	maketicks
define_whole_vertical_pixels	mode_def
endchar	mode_setup
extra_beginchar	o_correction
extra_endchar	proofrule
extra_setup	proofrulethickness
font_coding_scheme	rulepen
font_extra_space	smode
Определены как пустые операции в пакете mfplain	
cullit	proofoffset
currenttransform	screenchars
gfcorners	screenrule
grayfont	screenstrokes
hround	showit
imagerules	slantfont
lowres_fix	titlefont
nodisplays	unitpixel
notransforms	vround
openit	

Таблица 12: Макросы и внутренние переменные, определенные только в пакете mfplain.

и ⟨опция рисования⟩ `withweight`. Далее следует полный список внутренних переменных, чьи примитивные METAFONT-значения не имеют смысла в MetaPost:

```
autorounding fillin       proofing     tracingpens   xoffset
chardx       granularity  smoothing    turningcheck  yoffset
chardy       hppp         tracingedges vppp
```

Есть еще примитив METAFONT, имеющий несколько иное значение в MetaPost. Оба языка допускают команду в форме

`special` ⟨строковое выражение⟩`;` ,

но METAFONT копирует строку в свой выходной файл "обобщенного шрифта", тогда как MetaPost интерпретирует строку как последовательность команд PostScript, которые помещаются в начало следующего выходного файла

В этом рассмотрении стоит заметить, что линейки в материале TEX, включаемом через `btex..etex`, в MetaPost округляются до правильного числа пикселов, согласно правилам преобразования PostScript [1]. В METAFONTлинейки напрямую не генерируются, а просто включаются в special-команды и интерпретируются позже другими программами, такими как `gftodvi`, поэтому здесь нет никаких преобразований.

Все другие различия между METAFONT и MetaPost — это возможности, которые можно найти только в MetaPost. Они перечислены в таблице 13. Единственными командами, которые не обсуждались в предыдущих разделах и которые есть в этой таблице, являются `extra_beginfig`, `extra_endfig` и `mpxbreak`. Первые две — это строки, содержащие дополнительные команды для обработки в соответственно `beginfig` и `endfig`, — они подобны `extra_beginchar` и `extra_endchar`, обрабатываемым в `beginchar` и `endchar`. Файл `boxes.mp` использует эти возможности.

Другой новой возможностью, приведенной в таблице 13, является `mpxbreak`. Она используется для разделения блоков транслированных команд TEX или troff в `mpx`-файлах. Она не должна затрагивать пользователей, т.к. `mpx`-файлы генерируются автоматически.

Список литературы

[1] Adobe Systems Inc. *PostScript Language Reference Manual.* Addison Wesley, Reading, Massachusetts, second edition, 1990.

[2] J. D. Hobby. Smooth, easy to compute interpolating splines. *Discrete and Computational Geometry*, 1(2), 1986.

[3] Brian W. Kernighan. Pic—a graphics language for typesetting. In *Unix Research System Papers, Tenth Edition*, pages 53–77. AT&T Bell Laboratories, 1990.

[4] D. E. Knuth. *The METAFONTbook.* Addison Wesley, Reading, Massachusetts, 1986. Volume C of *Computers and Typesetting.* Д. Е. Кнут *Все про METAFONT.* — ТД "Вильямс", 2003.

[5] D. E. Knuth. *The TEXbook.* Addison Wesley, Reading, Massachusetts, 1986. Volume A of *Computers and Typesetting.* Д. Е. Кнут *Все про TEX.* — Протвино: АО RDTEX, 1993.

[6] D. E. Knuth. *Computer Modern Typefaces.* Addison Wesley, Reading, Massachusetts, 1986. Volume E of *Computers and Typesetting.*

[7] D. E. Knuth. The new versions of TEX and METAFONT. *TUGboat, the TEX User's Group Newsletter*, 10(3):325–328, November 1989.

Примитивы MetaPost, отсутствующие в METAFONT		
blackpart	fontsize	setbounds
bluepart	for within	stroked
bounded	greenpart	textpart
btex	greypart	textual
clip	infont	tracinglostchars
clipped	linecap	troffmode
closefrom	linejoin	truecorners
cmykcolor	llcorner	ulcorner
color	lrcorner	urcorner
colormodel	magentapart	verbatimtex
cyanpart	miterlimit	withcmykcolor
dashed	mpprocset	withcolor
dashpart	mpxbreak	withgreyscale
defaultcolormodel	pathpart	withoutcolor
etex	penpart	withpostscript
filenametemplate	prologues	withprescript
filled	readfrom	withrgbcolor
fontmapfile	redpart	write to
fontmapline	restoreclipcolor	yellowpart
fontpart	rgbcolor	
Переменные и макросы, определенные только в Plain MetaPost		
ahangle	cutbefore	evenly
ahlength	cuttings	extra_beginfig
background	dashpattern	extra_endfig
bbox	defaultfont	green
bboxmargin	defaultpen	image
beginfig	defaultscale	label
beveled	dotlabel	labeloffset
black	dotlabels	mitered
blue	drawarrow	red
buildcycle	drawdblarrow	rounded
butt	drawoptions	squared
center	endfig	thelabel
cutafter	EOF	white

Таблица 13: Макросы и внутренние переменные, определенные только в MetaPost, — их нет в METAFONT.

Предметный указатель