The `pgfmolbio` package –
Molecular Biology Graphs with Ti*k*Z[*]

Wolfgang Skala[†]

CTAN: `http://www.ctan.org/pkg/pgfmolbio`

2012/10/01

The experimental package `pgfmolbio` draws graphs typically found in molecular biology texts. Currently, the package contains three modules: chromatogram creates DNA sequencing chromatograms from files in standard chromatogram format (`scf`); domains draws protein domain diagrams; convert integrates `pgfmolbio` with TeX engines that lack Lua support.

---

[*]This document describes version v0.2, dated 2012/10/01.

[†]Division of Structural Biology, Department of Molecular Biology, University of Salzburg, Austria; `Wolfgang.Skala@stud.sbg.ac.at`

Contents

1 Introduction

1.1 About `pgfmolbio`

Over the decades, TeX has gained popularity across a large number of disciplines. Although originally designed as a mere typesetting system, packages such as `pgf`[1] and `pstricks`[2] have strongly extended its *drawing* abilities. Thus, one can create complicated charts that perfectly integrate with the text.

Texts on molecular biology include a range of special graphs, e.g. multiple sequence alignments, membrane protein topologies, DNA sequencing chromatograms, protein domain diagrams, plasmid maps and others. The `texshade`[3] and `textopo`[4] packages cover alignments and topologies, respectively, but packages dedicated to the remaining graphs are absent. Admittedly, one may create those images with various external programs and then include them in the TeX document. Nevertheless, purists (like the author of this document) might prefer a TeX-based approach.

The `pgfmolbio` package aims at becoming such a purist solution. In the current development release, `pgfmolbio` is able to

- read DNA sequencing files in standard chromatogram format (`scf`) and draw the corresponding chromatogram;

- read protein domain information from Uniprot or general feature format files (`gff`) and draw domain diagrams.

To this end, `pgfmolbio` relies on routines from `pgf`'s TikZ frontend and on the Lua scripting language implemented in LuaTeX. Consequently, the package will not work directly with traditional engines like pdfTeX. However, a converter module ensures a high degree of backward compatibility.

Since this is a development release, `pgfmolbio` presumably includes a number of bugs, and its commands and features are likely to change in future versions. Moreover, the current version is far from complete, but since time is scarce, I am

---

[1] Tantau, T. (2010). The TikZ and PGF packages. `http://ctan.org/tex-archive/graphics/pgf/`.

[2] van Zandt, T., Niepraschk, R., and Voß, H. (2007). PSTricks: PostScript macros for Generic TeX. `http://ctan.org/tex-archive/graphics/pstricks`.

[3] Beitz, E. (2000). TeXshade: shading and labeling multiple sequence alignments using LaTeX 2ε. *Bioinformatics* **16**(2), 135–139. `http://ctan.org/tex-archive/macros/latex/contrib/texshade`.

[4] Beitz, E. (2000). TeXtopo: shaded membrane protein topology plots in LaTeX 2ε. *Bioinformatics* **16**(11), 1050–1051. `http://ctan.org/tex-archive/macros/latex/contrib/textopo`.

unable to predict when (and if) additional functions become available. Nevertheless, I would greatly appreciate any comments or suggestions.

1.2 Getting Started

Before you consider using `pgfmolbio`, please make sure that both your LuaTeX (at least 0.70.2) and `pgf` (at least 2.10) installations are up-to-date. Once your TeX system meets these requirements, just load `pgfmolbio` as usual, i.e. by

\usepackage[⟨*module*⟩]{pgfmolbio}

The package is divided into *modules*, each of which produces a certain type of graph. Currently, three ⟨*module*⟩s are available:

- chromatogram (chapter 2) allows you to draw DNA sequencing chromatograms obtained by the Sanger sequencing method.

- domains (chapter 3) provides macros for drawing protein domain diagrams and is also able to read domain information from files in Uniprot or general feature format.

- Furthermore, convert (chapter 4) is used with one of the modules above and generates "pure" Ti*k*Z code suitable for TeX engines lacking Lua support.

\pgfmolbioset[⟨*module*⟩]{⟨*key-value list*⟩}

Fine-tunes the graphs produced by each `pgfmolbio` module. The possible keys are described in the sections on the respective modules.

2 The **chromatogram** module

2.1 Overview

The chromatogram module draws DNA sequencing chromatograms stored in standard chromatogram format (`scf`), which was developed by Simon Dear and Rodger Staden[1]. The documentation for the Staden package[2] describes the current version of the `scf` format in detail. As far as they are crucial to understanding the Lua code, we will discuss some details of this file format in the documented source code (section 5.4). Note that `pgfmolbio` only supports `scf` version 3.00.

2.2 Drawing Chromatograms

\pmbchromatogram[⟨*key-value list*⟩]**{**⟨*scf file*⟩**}**

The chromatogram module defines a single command, which reads a chromatogram from an ⟨*scf file*⟩ and draws it with routines from Ti*k*Z (Example 2.1). The options, which are set in the ⟨*key-value list*⟩, configure the appearance of the chromatogram. The following sections will elaborate on the available keys.

Example 2.1
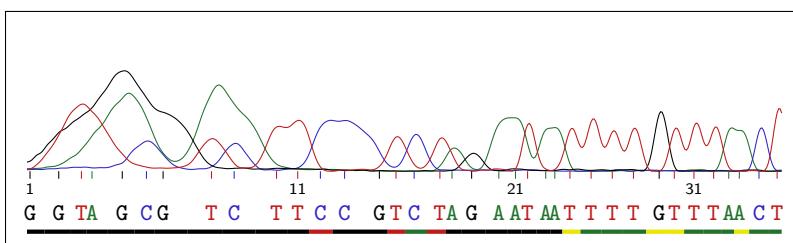

```
1  \begin{tikzpicture} % optional
2    \pmbchromatogram{SampleScf.scf}
3  \end{tikzpicture} % optional
```

---

[1]Dear, S. and Staden, R. (1992). A standard file format for data from DNA sequencing instruments. *DNA Seq.* **3**(2), 107–110.
[2]http://staden.sourceforge.net/

3

Although you will often put `\pmbchromatogram` into a `tikzpicture` environment, you may actually use the macro on its own. `pgfmolbio` checks whether the command is surrounded by a `tikzpicture` and adds this environment if necessary.

2.3 Displaying Parts of the Chromatogram

/pgfmolbio/chromatogram/**sample range =**⟨*lower*⟩**-**⟨*upper*⟩**[step** ⟨*int*⟩**]**

Default: `1-500 step 1`

`sample range` selects the part of the chromatogram which `pgfmolbio` should display. The value for this key consists of two or three parts, separated by the keywords `-` and `step`. The package will draw the chromatogram data between the ⟨*lower*⟩ and ⟨*upper*⟩ boundary. There are two ways of specifying these limits:

1. If you enter a number, `pgfmolbio` includes the data from the ⟨*lower*⟩ to the ⟨*upper*⟩ sample point (Example 2.2). A *sample point* represents one measurement of the fluorescence signal along the time axis, where the first sample point has index 1. One peak comprises about 20 sample points.

Example 2.2
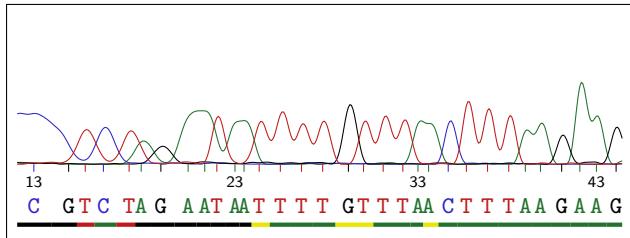

```
1 \pmbchromatogram[sample range=200-600]{SampleScf.scf}
```

2. If you enter the keyword `base` followed by an optional space and a number, the chromatogram starts or stops at the peak corresponding to the respective base. The first detected base peak has index 1. Compare Examples 2.2 and 2.3 to see the difference.

The optional third part of the value for `sample range` orders the package to draw every ⟨*int*⟩th sample point. If your document contains large chromatograms or a great number of them, drawing fewer sample points increases typesetting time at the cost of image quality (Example 2.4). Nevertheless, the key may be especially useful while optimizing the layout of complex chromatograms.

Example 2.3
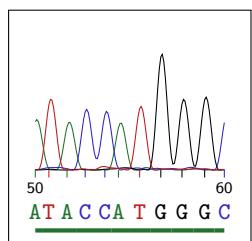

```
1  \pmbchromatogram[%
2      sample range=base 50-base60
3  ]{SampleScf.scf}
```

Example 2.4
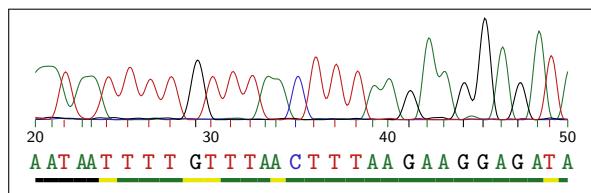

```
1  \pmbchromatogram[%
2      sample range=base 20-base 50 step 1
3  ]{SampleScf.scf}
```
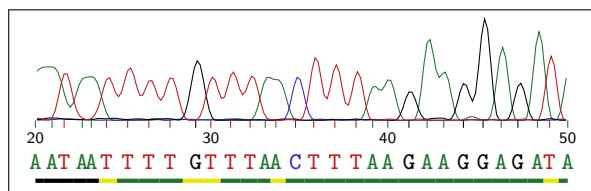


```
1  \pmbchromatogram[%
2      sample range=base 20-base 50 step 2
3  ]{SampleScf.scf}
```
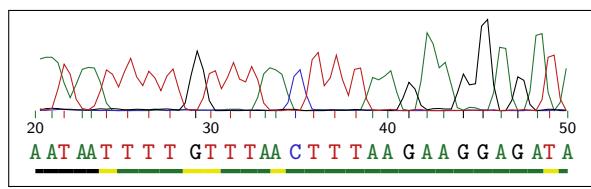


```
1  \pmbchromatogram[%
2      sample range=base 20-base 50 step 4
3  ]{SampleScf.scf}
```

2.4 General Layout

/pgfmolbio/chromatogram/**x unit** =⟨*dimension*⟩

Default: `0.2mm`

/pgfmolbio/chromatogram/**y unit** =⟨*dimension*⟩

Default: `0.01mm`

These keys set the horizontal distance between two consecutive sample points and the vertical distance between two fluorescence intensity values, respectively. Example 2.5 illustrates how you can enlarge a chromatogram twofold by doubling these values.

Example 2.5
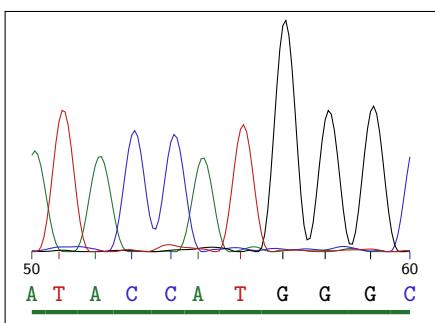

```
1  \pmbchromatogram[%
2      sample range=base 50-base 60,
3      x unit=0.4mm,
4      y unit=0.02mm
5  ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/**samples per line** =⟨*number*⟩

Default: `500`

/pgfmolbio/chromatogram/**baseline skip** =⟨*dimension*⟩

Default: `3cm`

A new chromatogram "line" starts after ⟨*number*⟩ sample points, and the baselines of adjacent lines (i.e., the y-value of fluorescence signals with zero intensity) are separated by ⟨*dimension*⟩. In Example 2.6, you see two lines, each of which contains 250 of the 500 sample points drawn. Furthermore, the baselines are 3.5 cm apart.

/pgfmolbio/chromatogram/**canvas style** /.style=⟨*style*⟩

Default: `draw=none, fill=none`

Example 2.6
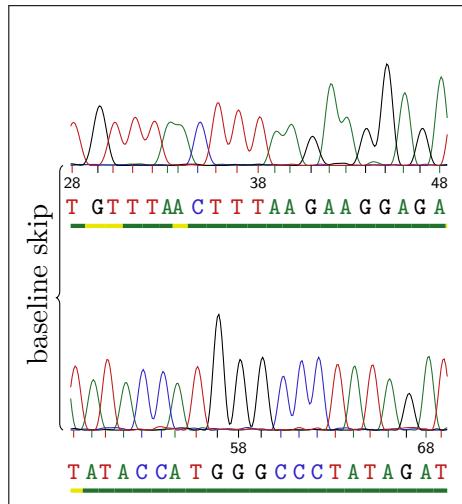

```
1  \begin{tikzpicture}%
2      [decoration=brace]
3    \pmbchromatogram[%
4        sample range=401-900,
5        samples per line=250,
6        baseline skip=3.5cm
7      ]{SampleScf.scf}
8    \draw[decorate]
9      (-0.1cm, -3.5cm) -- (-0.1cm, 0cm)
10     node[pos=0.5, rotate=90, above=5pt]
11       {baseline skip};
12 \end{tikzpicture}
```

Default: **2cm**

The *canvas* is the background of the trace area. Its left and right boundaries coincide with the start and the end of the chromatogram, respectively. Its lower boundary is the baseline, and its upper border is separated from the lower one by ⟨*dimension*⟩. Although the canvas is usually transparent, its ⟨*style*⟩ can be changed. In Example 2.7, we decrease the height of the canvas and color it light gray.

Example 2.7
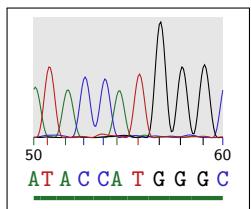

```
1  \pmbchromatogram[%
2      sample range=base 50-base 60,
3      canvas style/.style={draw=none, fill=black!10},
4      canvas height=1.6cm
5  ]{SampleScf.scf}
```

2.5 Traces

/pgfmolbio/chromatogram/**trace A style /.style=**⟨*style*⟩

Default: **pmbTraceGreen**

/pgfmolbio/chromatogram/**trace C style /.style=**⟨*style*⟩

Default: **pmbTraceBlue**

/pgfmolbio/chromatogram/**trace G style /.style=**⟨*style*⟩

Default: **pmbTraceBlack**

/pgfmolbio/chromatogram/**trace T style /.style=**⟨*style*⟩

Default: **pmbTraceRed**

/pgfmolbio/chromatogram/**trace style =**⟨*style*⟩

Default: (none)

The *traces* indicate variations in fluorescence intensity during chromatography, and each trace corresponds to a base. The first four keys set the respective ⟨*style*⟩

basewise, whereas `trace style` changes all styles simultaneously. Note the syntax differences between `trace style` and `trace A style` etc. The standard styles simply color the traces; Table 2.1 lists the color specifications.

Table 2.1: Colors defined by the chromatogram module.

Name	xcolor model	Values	Example
pmbTraceGreen	RGB	34, 114, 46	
pmbTraceBlue	RGB	48, 37, 199	
pmbTraceBlack	RGB	0, 0, 0	
pmbTraceRed	RGB	191, 27, 27	
pmbTraceYellow	RGB	233, 230, 0	

In Example 2.8, we change the style of all traces to a thin line and then add some patterns and colors to the A and T trace.

Example 2.8
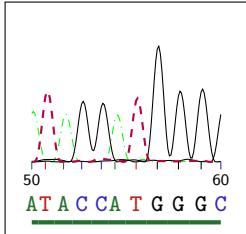

```
1  \pmbchromatogram[%
2      sample range=base 50-base 60,
3      trace style=thin,
4      trace A style/.append style={dashdotted, green},
5      trace T style/.style={thick, dashed, purple}
6  ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/**traces drawn** =A|C|G|T|any combination thereof

Default: `ACGT`

The value of this key governs which traces appear in the chromatogram. Any combination of the single-letter abbreviations for the standard bases will work. Example 2.9 only draws the cytosine and guanine traces.

Example 2.9
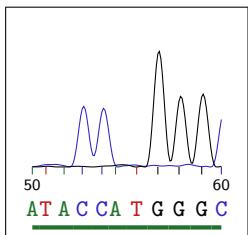

```
1  \pmbchromatogram[%
2      sample range=base 50-base 60,
3      traces drawn=CG
4  ]{SampleScf.scf}
```

2.6 Ticks

/pgfmolbio/chromatogram/**tick A style** /.style=⟨*style*⟩

Default: `thin, pmbTraceGreen`

/pgfmolbio/chromatogram/**tick C style** /.style=⟨*style*⟩

Default: `thin, pmbTraceBlue`

/pgfmolbio/chromatogram/**tick G style** /.style=⟨*style*⟩

Default: `thin, pmbTraceBlack`

/pgfmolbio/chromatogram/**tick T style** /.style=⟨*style*⟩

Default: `thin, pmbTraceRed`

/pgfmolbio/chromatogram/**tick style =**⟨*style*⟩

Default: (none)

Ticks below the baseline indicate the maxima of the trace peaks. The first four keys set the respective ⟨*style*⟩ basewise, whereas `tick style` changes all styles simultaneously. Note the syntax differences between `tick style` and `tick A style` etc. Example 2.10 illustrates how one can draw thick ticks, which are red if they indicate a cytosine peak.

Example 2.10
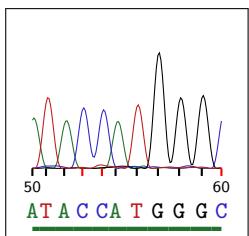

```
1 \pmbchromatogram[%
2     sample range=base 50-base 60,
3     tick style=thick,
4     tick C style/.append style={red}
5 ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/**tick length =**⟨*dimension*⟩

Default: `1mm`

This key determines the length of each tick. In Example 2.11, the ticks are twice as long as usual.

Example 2.11
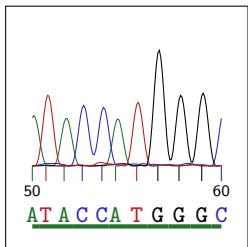

```
1 \pmbchromatogram[%
2     sample range=base 50-base 60,
3     tick length=2mm
4   ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/**ticks drawn** =A|C|G|T|any combination thereof

Default: **ACGT**

The value of this key governs which ticks appear in the chromatogram. Any combination of the single-letter abbreviations for the standard bases will work. Example 2.12 only displays the cytosine and guanine ticks.

Example 2.12
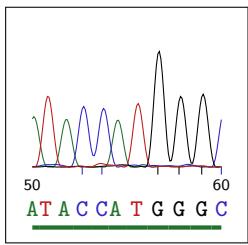

```
1 \pmbchromatogram[%
2     sample range=base 50-base 60,
3     ticks drawn=CG
4   ]{SampleScf.scf}
```

2.7 Base Labels

/pgfmolbio/chromatogram/**base label A text** =⟨*text*⟩

Default: **\strut A**

/pgfmolbio/chromatogram/**base label C text** =⟨*text*⟩

Default: **\strut C**

/pgfmolbio/chromatogram/**base label G text** =⟨*text*⟩

Default: **\strut G**

> /pgfmolbio/chromatogram/**base label T text** =⟨*text*⟩

Default: `\strut T`

Base labels below each tick spell the nucleotide sequence deduced from the traces. By default, the ⟨*text*⟩ that appears in these labels equals the single-letter abbreviation of the respective base. The `\strut` macro ensures equal vertical spacing. In Example 2.13, we print lowercase letters beneath adenine and thymine.

Example 2.13
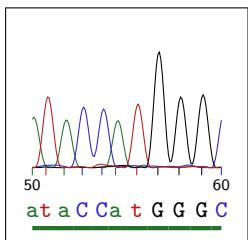

```
1  \pmbchromatogram[%
2      sample range=base 50-base 60,
3      base label A text=\strut a,
4      base label T text=\strut t
5  ]{SampleScf.scf}
```

> /pgfmolbio/chromatogram/**base label A style** /.style=⟨*style*⟩

Default: `below=4pt, font=\ttfamily\footnotesize, pmbTraceGreen`

> /pgfmolbio/chromatogram/**base label C style** /.style=⟨*style*⟩

Default: `below=4pt, font=\ttfamily\footnotesize, pmbTraceBlue`

> /pgfmolbio/chromatogram/**base label G style** /.style=⟨*style*⟩

Default: `below=4pt, font=\ttfamily\footnotesize, pmbTraceBlack`

> /pgfmolbio/chromatogram/**base label T style** /.style=⟨*style*⟩

Default: `below=4pt, font=\ttfamily\footnotesize, pmbTraceRed`

> /pgfmolbio/chromatogram/**base label style** =⟨*style*⟩

Default: (none)

The first four keys set the respective ⟨*style*⟩ basewise, whereas `base label style` changes all styles simultaneously. Each base label is a Ti*k*Z node anchored to the lower end of the respective tick. Thus, the ⟨*style*⟩ should contain placement keys such as `below` or `anchor=south`. Example 2.14 shows some (imaginative) base label styles.

12

Example 2.14
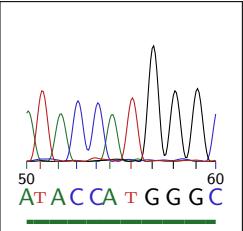

```
1  \pmbchromatogram[%
2      sample range=base 50-base 60,
3      base label style=%
4        {below=2pt, font=\sffamily\footnotesize},
5      base label T style/.append style=%
6        {below=4pt, font=\tiny}
7  ]{SampleScf.scf}
```


/pgfmolbio/chromatogram/**base labels drawn** =A|C|G|T|any combination thereof

---

Default: **ACGT**

The value of this key governs which base labels appear in the chromatogram. Any combination of the single-letter abbreviations for the standard bases will work. Example 2.15 only displays cytosine and guanine base labels.

Example 2.15
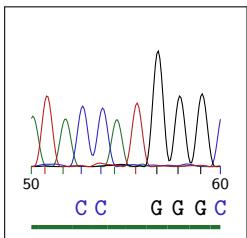

```
1  \pmbchromatogram[%
2      sample range=base 50-base 60,
3      base labels drawn=CG
4  ]{SampleScf.scf}
```

2.8 Base Numbers

/pgfmolbio/chromatogram/**show base numbers** =⟨*boolean*⟩

Default: **true**

Turns the *base numbers* on or off, which indicate the indices of the base peaks below the traces.

/pgfmolbio/chromatogram/**base number style** /.style=⟨*style*⟩

Default: **pmbTraceBlack, below=-3pt, font=\sffamily\tiny**

Determines the placement and appearance of the base numbers. Example 2.16 contains bold red base numbers that are shifted slightly upwards.

Example 2.16
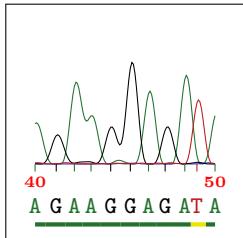

```
1  \pmbchromatogram[%
2      sample range=base 40-base 50,
3      base number style/.style={below=-3pt,%
4          font=\rmfamily\bfseries\tiny, red}
5  ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/**base number range** =⟨*lower*⟩-⟨*upper*⟩[step ⟨*interval*⟩]

Default: `auto-auto step 10`

This key decides that every ⟨*interval*⟩th base number from ⟨*lower*⟩ to ⟨*upper*⟩ should show up in the output; the `step` part is optional. If you specify the keyword `auto` instead of a number for ⟨*lower*⟩ or ⟨*upper*⟩, the base numbers start or finish at the leftmost or rightmost base peak shown, respectively. In Example 2.17, only peaks 42 to 46 receive a number.

Example 2.17
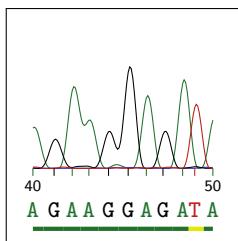

```
1  \pmbchromatogram[%
2      sample range=base 40-base 50,
3      base number range=42-46 step 1,
4  ]{SampleScf.scf}
```

2.9 Probabilities

Programs such as `phred`[3] assign a *probability* or *quality value* Q to each called base after chromatography. Q is calculated from the error probability P_e by $Q = -10\log_{10} P_e$. For example, a Q value of 20 means that 1 in 100 base calls is wrong.

/pgfmolbio/chromatogram/**probability distance** =⟨*dimension*⟩

Default: `0.8cm`

Sets the distance between the base probability rules and the baseline.

---

[3]Ewing, B., Hillier, L., Wendl, M. C., and Green, P. (1998). Base-calling of automated sequencer traces using phred. I. Accuracy assessment. *Genome Res.* **8**(3), 175–185.

/pgfmolbio/chromatogram/**probabilities drawn** =A|C|G|T|any combination thereof

> Default: **ACGT**

Governs which probabilities appear in the chromatogram. Any combination of the single-letter abbreviations for the standard bases will work. In Example 2.18, we shift the probability indicator upwards and only show the quality values of cytosine and thymine peaks.

Example 2.18
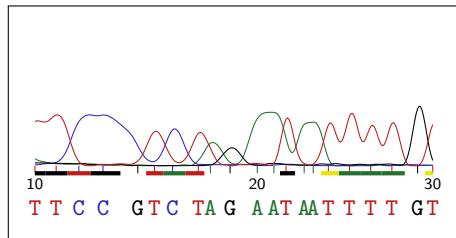

```
1  \pmbchromatogram[%
2      sample range=base 10-base 30,
3      probabilities drawn=CT,
4      probability distance=1mm
5  ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/**probability style function** =⟨*Lua function name*⟩

> Default: **nil**

By default, the probability rules are colored black, red, yellow and green for quality scores < 10, < 20, < 30 and ≥ 30, respectively. However, you can override this behavior by providing a ⟨*Lua function name*⟩ to **probability style function**. This Lua function must read a single argument of type number and return a string appropriate for the optional argument of TikZ's **\draw** command. For instance, the function shown in Example 2.19 determines the lowest and highest probability and colors intermediate values according to a red–yellow–green gradient.

2.10 Miscellaneous Keys

/pgfmolbio/chromatogram/**bases drawn** =A|C|G|T|any combination thereof

> Default: **ACGT**

This key simultaneously sets **traces drawn**, **ticks drawn**, **base labels drawn** and **probabilities drawn** (see Example 2.20).

Example 2.19
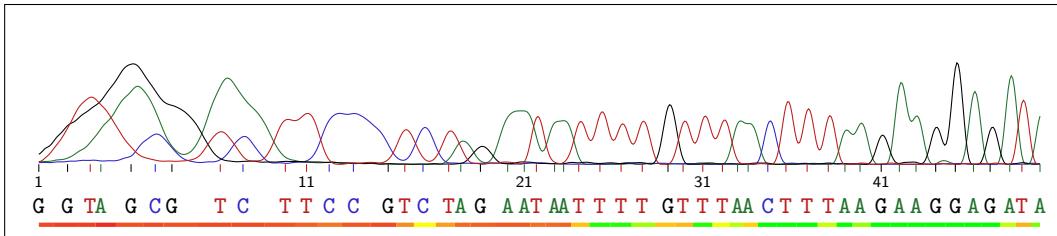

```
1  \directlua{
2    function probabilityGradient (prob)
3      local minProb, maxProb = pmbChromatogram:getMinMaxProbability()
4      local scaledProb = prob / maxProb * 100
5      local color = ''
6      if scaledProb < 50 then
7        color = 'yellow!' .. scaledProb * 2 .. '!red'
8      else
9        color = 'green!' .. (scaledProb - 50) * 2 .. '!yellow'
10     end
11     return 'ultra thick, ' .. color
12   end
13 }
14 \pmbchromatogram[%
15     samples per line=1000,
16     sample range=base 1-base 50,
17     probability style function=probabilityGradient
18   ]{SampleScf.scf}
```

Example 2.20
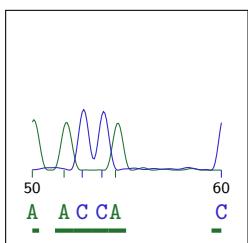

```
1  \pmbchromatogram[%
2      sample range=base 50-base 60,
3      bases drawn=AC
4    ]{SampleScf.scf}
```

3 The **domains** module

3.1 Overview

Protein domain diagrams appear frequently in databases such as Pfam[1] or PROSITE[2]. Domain diagrams are often drawn using standard graphics software or tools such as PROSITE's MyDomains image creator[3]. However, the domains module provides an integrated approach for generating domain diagrams from TeX code or from external files.

3.2 Domain Diagrams and Their Features

```
\begin{pmbdomains}[⟨key-value list⟩]{⟨sequence length⟩}
    ⟨features⟩
\end{pmbdomains}
```

Draws a domain diagram with the ⟨features⟩ given. The ⟨key-value list⟩ configures its appearance. ⟨sequence length⟩ is the total number of residues in the protein. (Although you must eventually specify a sequence length, you may actually leave the mandatory argument empty and use the sequence length key instead; see section 3.10).

You can put a pmbdomains environment into a tikzpicture, but you also may use the environment on its own. pgfmolbio checks whether it is surrounded by a tikzpicture and adds this environment if necessary.

/pgfmolbio/domains/**name** =⟨text⟩

Default: Protein

The name of the protein, which usually appears centered above the diagram.

/pgfmolbio/domains/**show name** =⟨boolean⟩

Default: true

---

[1] Finn, R. D., Mistry, J. *et al.* (2010). The Pfam protein families database. *Nucleic Acids Res.* **38**, D211–D222.

[2] Sigrist, C. J. A., Cerutti, L. *et al.* (2010). PROSITE, a protein domain database for functional characterization and annotation. *Nucleic Acids Res.* **38**, D161–D166.

[3] http://prosite.expasy.org/mydomains/

17

Determines whether both the name and sequence length are shown.

\addfeature[⟨*key-value list*⟩]{⟨*type*⟩}{⟨*start*⟩}{⟨*stop*⟩}

Adds a feature of the given ⟨*type*⟩ to the current domain diagram (only defined inside `pmbdomains`). The feature spans the residues from ⟨*start*⟩ to ⟨*stop*⟩. These arguments are either numbers, which refer to residues in the relative numbering scheme, or numbers in parentheses, which refer to absolute residue numbers (see section 3.3).

/pgfmolbio/domains/`description` =⟨*text*⟩

Default: (none)

Sets the feature description (Example 3.1).

Example 3.1
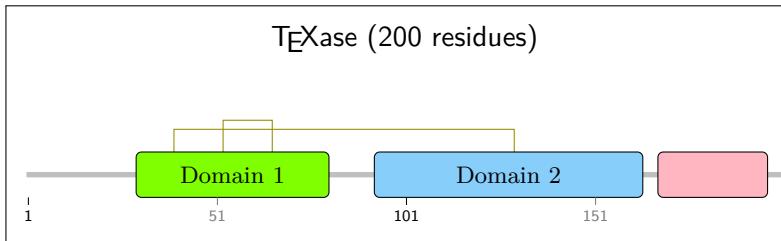

```
1 \begin{tikzpicture} % optional
2   \begin{pmbdomains}[name=\TeX ase]{200}
3     \addfeature{disulfide}{40}{129}
4     \addfeature{disulfide}{53}{65}
5     \addfeature[description=Domain 1]{domain}{30}{80}
6     \addfeature[description=Domain 2]{domain}{93}{163}
7     \addfeature{domain}{168}{196}
8   \end{pmbdomains}
9 \end{tikzpicture} % optional
```

3.3 General Layout

/pgfmolbio/domains/`x unit` =⟨*dimension*⟩

Default: `0.5mm`

The width of a single residue.

/pgfmolbio/domains/`y unit` =⟨*dimension*⟩

Default: `6mm`

The height of a default `domain` feature.

/pgfmolbio/domains/**residues per line** =⟨*number*⟩

Default: `200`

A new domain diagram "line" starts after ⟨*number*⟩ residues.

/pgfmolbio/domains/**baseline skip** =⟨*factor*⟩

Default: `3`

The baselines of consecutive lines (i. e., the main chain y-coordinates) are separated by ⟨*factor*⟩ times the value of `y unit`. In Example 3.2, you see four lines, each of which contains up to 30 residues. Note how domains are correctly broken across lines. Furthermore, the baselines are $2 \times 4 = 8$ mm apart.

Example 3.2
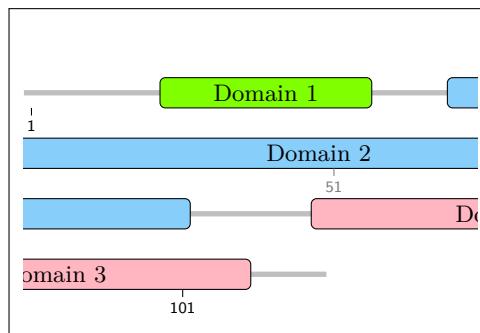

```
1  \begin{pmbdomains}%
2     [show name=false, x unit=2mm, y unit=4mm,
3     residues per line=30, baseline skip=2]{110}
4  \addfeature[description=Domain 1]{domain}{10}{23}
5  \addfeature[description=Domain 2]{domain}{29}{71}
6  \addfeature[description=Domain 3]{domain}{80}{105}
7  \end{pmbdomains}
```

/pgfmolbio/domains/**residue numbering** =⟨*numbering scheme*⟩

Default: `auto`

A protein's amino acid residues are usually numbered consecutively starting from 1. However, there are different numbering schemes. For example, residue numbering in a serine protease related to chymotrypsin typically follows the numbering in chy-

motrypsinogen[4]. The target protease sequence is aligned to the chymotrypsinogen sequence, and equivalent residues receive the same number. Insertions into the target sequence are indicated by appending letters to the last aligned residue (e. g., 186, *186A*, *186B*, 187), whereas gaps in the target sequence cause gaps in the numbering (e. g., 124, 125, 128, 129).

In `pgfmolbio`, you can specify a relative ⟨*numbering scheme*⟩ via the `residue numbering` key. The keyword `auto` indicates that residues are numbered from 1 to (sequence length), i. e. absolute and relative numberings coincide. This is the case in all examples above. The complete syntax for the key is

> ⟨*numbering scheme*⟩ := {⟨*range*⟩[,⟨*range*⟩,...]}
> ⟨*range*⟩ := ⟨*start*⟩-⟨*end*⟩ | ⟨*start*⟩
> ⟨*start*⟩ := ⟨*number*⟩ | ⟨*number*⟩⟨*letter*⟩
> ⟨*end*⟩ := ⟨*number*⟩ | ⟨*letter*⟩

Example 3.3 shows a custom ⟨*numbering scheme*⟩, in this case for kallikrein-related peptidase 2 (KLK2), a chymotrypsin-like serine proteases. (In the following explanation, the subscripts 'abs' and 'rel' denote absolute and relative numbering, respectively).

- Residue 1_{abs} is labeled 16_{rel}, residue 2_{abs} is labeled 17_{rel} etc. until residue 24_{abs}, which is labeled 39_{rel} (range `16-39`).

- Residue 25_{abs} corresponds to 41_{rel} etc. until residue $57_{abs}/73_{rel}$ (range `41-73`).

- Residue 40_{rel} is missing – no residue in KLK2 is equivalent to residue 40 in chymotrypsinogen.

- An insertion of 11 amino acids follows residue 95_{rel}. These residues are numbered from $95A_{rel}$ to $95K_{rel}$. Note that both `95A-K` and `95A-95K` are valid ranges.

- The number of the last residue is $245A_{rel}$(range `245A`).

/pgfmolbio/domains/**residue range** =⟨*lower*⟩-⟨*upper*⟩

Default: `auto-auto`

All residues from ⟨*lower*⟩ to ⟨*upper*⟩ will appear in the output. Possible values for ⟨*lower*⟩ and ⟨*upper*⟩ are:

- `auto`, which indicates the first or last residue, respectively;

- a plain number, which denotes a residue in the *relative* numbering scheme set by `residue numbering`;

---

[4]Bode, W., Mayr, I. *et al.* (1989). The refined 1.9 Å crystal structure of human α-thrombin: interaction with D-Phe-Pro-Arg chloromethylketone and significance of the Tyr-Pro-Pro-Trp insertion segment. *EMBO J.* **8**(11), 3467–3475.

Example 3.3
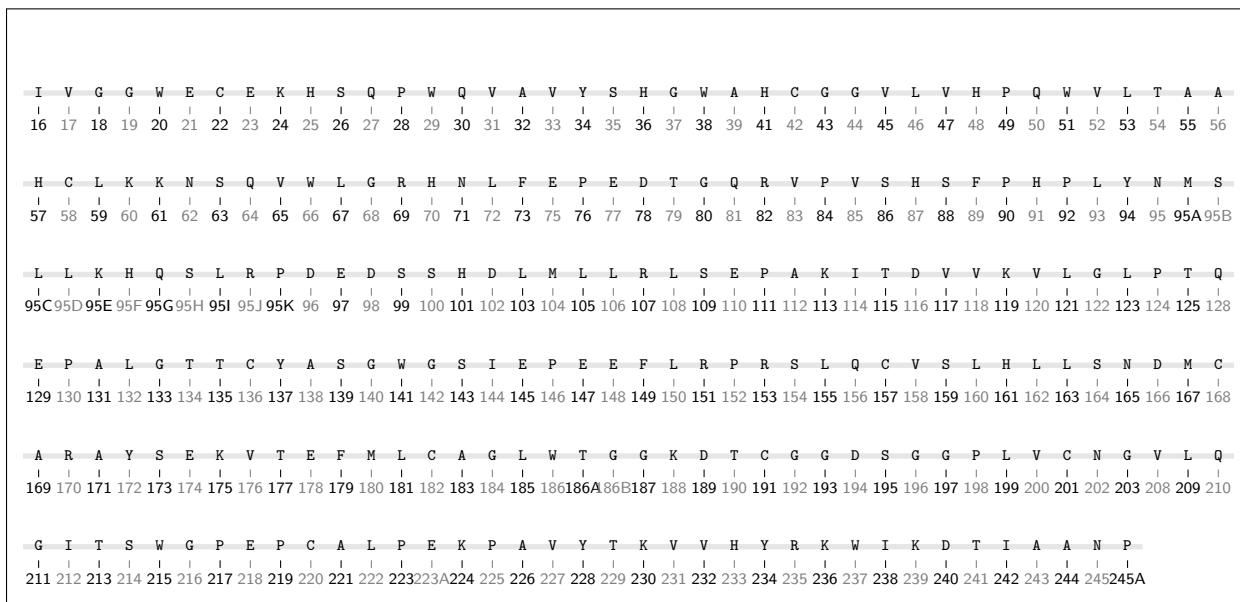

```
1  \begin{pmbdomains}[%
2    sequence=IVGGWECEKHSQPWQVAVYSHGWAHCGGVLVHPQWVLTAAHCLK%
3      KNSQVWLGRHNLFEPEDTGQRVPVSHSFPHPLYNMSLLKHQSLRPDEDSSH%
4      DLMLLRLSEPAKITDVVKVLGLPTQEPALGTTCYASGWGSIEPEEFLRPRS%
5      LQCVSLHLLSNDMCARAYSEKVTEFMLCAGLWTGGKDTCGGDSGGPLVCNG%
6      VLQGITSWGPEPCALPEKPAVYTKVVHYRKWIKDTIAANP,
7    residue numbering={16-39,41-73,75-95,95A-K,96-125,%
8      128-186,186A-186B,187-203,208-223,223A,224-245,245A},
9    x unit=4mm,
10   residues per line=40,
11   show name=false,
12   ruler range=auto-auto step 1,
13   ruler distance=-.3,
14   baseline skip=2
15 ]{237}
16 \setfeaturestyle{other/main chain}{*1{draw, line width=2pt, black!10}}
17 \addfeature{other/sequence}{16}{245A}
18 \end{pmbdomains}
```

21

- a parenthesized number, which denotes a residue in the *absolute* numbering scheme.

In Example 3.4, only residues 650_{abs} to 850_{rel} are shown. If a domain boundary lies outside of the range shown, only the appropriate part of the domain appears.

Example 3.4
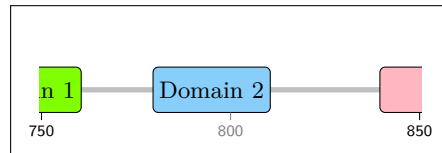

```
1  \begin{pmbdomains}[%
2      show name=false, residue range=(650)-850,
3      residue numbering={1-500,601-1100}]{1000}
4      \addfeature[description=Domain 1]{domain}{(630)}{(660)}
5      \addfeature[description=Domain 2]{domain}{(680)}{(710)}
6      \addfeature[description=Domain 3]{domain}{840}{1000}
7      \addfeature[description=Domain 4 (invisible)]{domain}{1010}{1040}
8  \end{pmbdomains}
```

/pgfmolbio/domains/**enlarge left** =⟨*dimension*⟩

　Default: `0cm`

/pgfmolbio/domains/**enlarge right** =⟨*dimension*⟩

　Default: `0cm`

/pgfmolbio/domains/**enlarge top** =⟨*dimension*⟩

　Default: `1cm`

/pgfmolbio/domains/**enlarge bottom** =⟨*dimension*⟩

　Default: `0cm`

`pgfmolbio` clips features that would protrude into the left or right margin. However, limits in the TikZ clipping mechanism prevent correct automatic updates of the bounding box for the domain diagram. Although the package tries hard to establish a bounding box that is sufficiently large, the process may require manual intervention. To this end, each `enlarge ...` key enlarges the bounding box at the respective side (Example 3.5).

Example 3.5
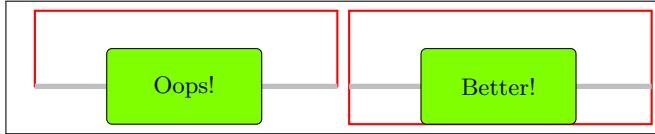

```
1  \tikzset{%
2    baseline, tight background,%
3    background rectangle/.style={draw=red, thick}%
4  }
5  \pgfmolbioset[domains]{show name=false, y unit=1cm, show ruler=false}
6
7  \begin{tikzpicture}[show background rectangle]
8    \begin{pmbdomains}{80}
9      \addfeature[description=Oops!]{domain}{20}{60}
10   \end{pmbdomains}
11 \end{tikzpicture}
12 \begin{tikzpicture}[show background rectangle]
13   \begin{pmbdomains}[enlarge bottom=-5mm]{80}
14     \addfeature[description=Better!]{domain}{20}{60}
15   \end{pmbdomains}
16 \end{tikzpicture}
```

3.4 Feature Styles and Shapes

Each (implicit and explicit) feature of a domain chart has a certain *shape* and *style*. For instance, you can see five different feature *shapes* in Example 3.1: We explicitly added two features of shape (and type) `disulfide` and three features of shape `domain`. Furthermore, the package implicitly included features of shape `other/name`, `other/main chain` and `other/ruler`.

Although the three `domain` features agree in shape, they differ in color, or (more generally) *style*. Since `pgfmolbio` distinguishes between shapes and styles, you may draw equally shaped features with different colors, strokes, shadings etc.

\setfeaturestyle{⟨*type*⟩}{⟨*style list*⟩}

Specifies a ⟨*style list*⟩ for the given feature ⟨*type*⟩. The complete syntax ist

> ⟨*style list*⟩ := {⟨*style list item*⟩[,⟨*style list item*⟩,...]}
> ⟨*style list item*⟩ := ⟨*multiplier*⟩⟨*style*⟩
> ⟨*multiplier*⟩ := [∗⟨*number*⟩]
> ⟨*style*⟩ := ⟨*single key-value pair*⟩ | {⟨*key-value list*⟩}

A style list item of the general form ∗⟨*n*⟩{⟨*style*⟩} instructs the package to repeat the ⟨*style*⟩ ⟨*n*⟩-times. (This syntax is reminiscent of column specifications in a

`tabular` environment. However, do *not* enclose numbers with more than one digit in curly braces!) You may omit the trivial multiplier `*1`, but never forget the curly braces surrounding a ⟨*style*⟩ that contains two or more key-value pairs. Furthermore, `pgfmolbio` loops over the style list until all features have been drawn.

For instance, the style list in Example 3.6 fills the first feature red, then draws a green one with a thick stroke, and finally draws two dashed blue features.

Example 3.6
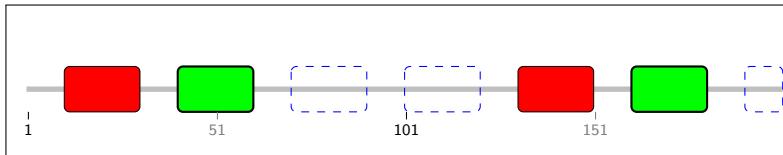

```
1  \begin{pmbdomains}[show name=false]{200}
2    \setfeaturestyle{domain}%
3      {fill=red, {thick, fill=green}, *2{blue, dashed}}
4    \addfeature{domain}{11}{30}
5    \addfeature{domain}{41}{60}
6    \addfeature{domain}{71}{90}
7    \addfeature{domain}{101}{120}
8    \addfeature{domain}{131}{150}
9    \addfeature{domain}{161}{180}
10   \addfeature{domain}{191}{200}
11 \end{pmbdomains}
```

/pgfmolbio/domains/`style =`⟨*style*⟩

Default: (empty)

Although `\setfeaturestyle` may appear in a `pmbdomains` environment, changes introduced in this way are not limited to the current TEX group (since feature styles are stored in Lua variables). Instead, use the `style` key to locally override a feature style (Example 3.7).

`\setfeaturestylealias`{⟨*new type*⟩}{⟨*existing type*⟩}

After calling this macro, the ⟨*new type*⟩ and ⟨*existing type*⟩ share a common style, while they still differ in their shapes.

`\setfeatureshape`{⟨*type*⟩}{⟨*TikZ code*⟩}

Defines a new feature shape named ⟨*type*⟩ or changes an existing one. **Caution:** If you change a shape within `pmbdomains`, you will also change the features of equal

24

Example 3.7
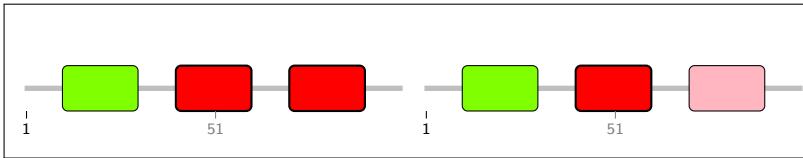

```
1  \begin{pmbdomains}[show name=false]{100}
2    \addfeature{domain}{11}{30}
3    \begingroup
4      \setfeaturestyle{domain}{{thick, fill=red}}
5      \addfeature{domain}{41}{60}
6    \endgroup
7    \addfeature{domain}{71}{90} % the new style persists ...
8  \end{pmbdomains}
9
10 \begin{pmbdomains}[show name=false]{100}
11   \addfeature{domain}{11}{30}
12   \addfeature[style={thick, fill=red}]{domain}{41}{60}
13   \addfeature{domain}{71}{90} % correct solution
14 \end{pmbdomains}
```

type that you already added. Thus, it is best to use `\setfeatureshape` only outside of this environment.

Several commands that are only available in the ⟨*TikZ code*⟩ allow you to design generic feature shapes:

- `\xLeft`, `\xMid` and `\xRight` expand to the left, middle and right x-coordinate of the feature. The coordinates are in a format suitable for `\draw` and similar commands.

- `\yMid` expands to the y-coordinate of the feature, i.e. the y-coordinate of the current line.

- You can access any values stored in the package's ⟨*key*⟩s with the macro `\pmbdomvalueof{`⟨*key*⟩`}`.

- The style key `/pgfmolbio/domains/current style` represents the current feature style selected from the associated style list.

The commands above are available for all features. By contrast, the following macros are limited to certain feature types:

- `\featureSequence` provides the amino acid sequence of the current feature. This macro is only available for explicitly added features and for `other/main chain`.

25

Example 3.8
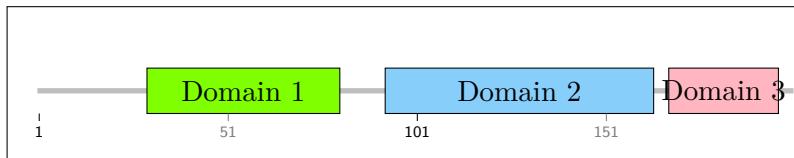

```
1  \setfeatureshape{domain}{%
2    \draw [/pgfmolbio/domains/current style]
3      (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
4      (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
5    \node at (\xMid, \yMid) {\pmbdomvalueof{description}};
6  }
7
8  \begin{pmbdomains}[show name=false]{200}
9    \addfeature[description=Domain 1]{domain}{30}{80}
10   \addfeature[description=Domain 2]{domain}{93}{163}
11   \addfeature[description=Domain 3]{domain}{168}{196}
12 \end{pmbdomains}
```

Example 3.9
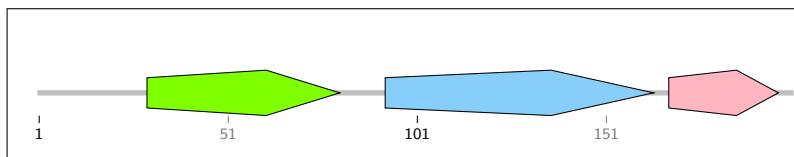

```
1  \setfeatureshape{domain}{%
2    \pgfmathsetmacro\middlecorners{%
3      \xLeft + (\xRight - \xLeft) * .618%
4    }
5    \draw [/pgfmolbio/domains/current style]
6      (\xLeft, \yMid + 2mm) --
7      (\middlecorners pt, \yMid + 3mm) --
8      (\xRight, \yMid) --
9      (\middlecorners pt, \yMid - 3mm) --
10     (\xLeft, \yMid - 2mm) --
11     cycle;
12 }
13
14 \begin{pmbdomains}[show name=false]{200}
15   \addfeature[description=Domain 1]{domain}{30}{80}
16   \addfeature[description=Domain 2]{domain}{93}{163}
17   \addfeature[description=Domain 3]{domain}{168}{196}
18 \end{pmbdomains}
```

Example 3.10
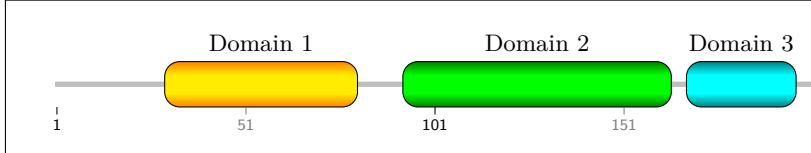

```
1  \pgfdeclareverticalshading[bordercolor,middlecolor]{mydomain}{100bp}{
2    color(0bp)=(bordercolor);
3    color(25bp)=(bordercolor);
4    color(40bp)=(middlecolor);
5    color(60bp)=(middlecolor);
6    color(75bp)=(bordercolor);
7    color(100bp)=(bordercolor)
8  }
9
10 \tikzset{%
11   domain middle color/.code=\colorlet{middlecolor}{#1},%
12   domain border color/.code=\colorlet{bordercolor}{#1}%
13 }
14
15 \setfeatureshape{domain}{%
16   \draw [shading=mydomain, rounded corners=2mm,
17     /pgfmolbio/domains/current style]
18     (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
19     (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
20   \node [above=3mm] at (\xMid, \yMid)
21     {\pmbdomvalueof{domain font}{\pmbdomvalueof{description}}};
22 }
23
24 \begin{pmbdomains}[show name=false]{200}
25   \setfeaturestyle{domain}{%
26     {domain middle color=yellow!85!orange,%
27     domain border color=orange},%
28     {domain middle color=green,%
29     domain border color=green!50!black}%
30     {domain middle color=cyan,%
31     domain border color=cyan!50!black}%
32   }
33   \addfeature[description=Domain 1]{domain}{30}{80}
34   \addfeature[description=Domain 2]{domain}{93}{163}
35   \addfeature[description=Domain 3]{domain}{168}{196}
36 \end{pmbdomains}
```

- `\residueNumber` equals the current residue number. This macro is only available for shape `other/ruler` (see section 3.7).

- `\currentResidue` expands to a single letter amino acid abbreviation. This macro is only available for shape `other/sequence` (see section 3.8).

In Example 3.8, we develop a simple `domain` shape, which is a rectangle containing a centered label with the feature description. Example 3.9 calculates an additional coordinate for a pentagonal domain shape and stores this coordinate in `\middlecorners`. Note that you have to insert "pt" after `\middlecorners` when using the stored coordinate. The domains in Example 3.10 display a custom shading and inherit their style from the style list.

> `\setfeatureshapealias{`⟨*new type*⟩`}{`⟨*existing type*⟩`}`

After calling this macro, the ⟨*new type*⟩ and ⟨*existing type*⟩ share a common shape, while they still differ in their styles.

> `\setfeaturealias{`⟨*new type*⟩`}{`⟨*existing type*⟩`}`

This is a shorthand for calling both `\setfeatureshape` and `\setfeaturestyle`.

3.5 Standard Features

`pgfmolbio` provides a range of standard features. This section explains simple features (i. e., those that support no or only few options), while later sections cover advanced ones. Some features include predefined aliases, which facilitate inclusion of external files (see section 3.10).

Feature `default`   (*no alias*)
A fallback for undefined features, in which case TeX issues a warning (Example 3.11).

<div align="center">

Example 3.11

</div>
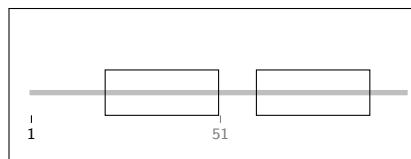
```
1 \begin{pmbdomains}[show name=false]{100}
2   \addfeature{default}{21}{50}
3   \addfeature{unknown}{61}{90} % i.e. default shape/style
4 \end{pmbdomains}
```

Feature `domain`   (*alias* `DOMAIN`)

A generic feature for protein domains. It consists of a rectangle with rounded corners and a label in the center, which shows the value of `description`.

`/pgfmolbio/domains/`**`domain font`**` =`⟨*font commands*⟩

  Default: `\footnotesize`

Sets the font for the label of a `domain` feature. The last command may take a single argument (Example 3.12).

Example 3.12
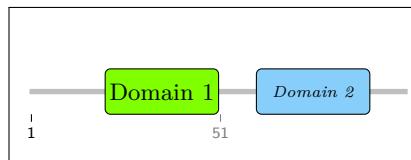

```
1 \begin{pmbdomains}[show name=false]{100}
2   \addfeature[description=Domain 1]{domain}{21}{50}
3   \addfeature[description=Domain 2,%
4     domain font=\tiny\textit]{DOMAIN}{61}{90}
5 \end{pmbdomains}
```

Feature `signal peptide`   (*alias* `SIGNAL`)

Adds a signal peptide (Example 3.13).

Feature `propeptide`   (*alias* `PROPEP`)

Adds a propeptide (Example 3.13).

Example 3.13
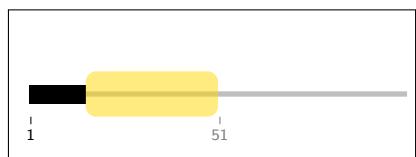

```
1 \begin{pmbdomains}[show name=false]{100}
2   \addfeature{signal peptide}{1}{15}
3   \addfeature{propeptide}{16}{50}
4 \end{pmbdomains}
```

Feature `carbohydrate`   (*alias* `CARBOHYD`)

Adds glycosylation (Example 3.14).

29

Example 3.14
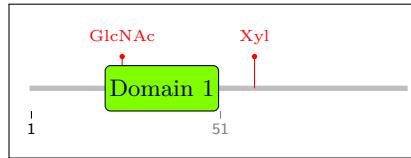

```
1 \begin{pmbdomains}[show name=false]{100}
2   \addfeature[description=GlcNAc]{carbohydrate}{25}{25}
3   \addfeature[description=Xyl]{CARBOHYD}{60}{60}
4   \addfeature[description=Domain 1]{domain}{21}{50}
5 \end{pmbdomains}
```

Feature other/main chain   (*no alias*)
This feature is automatically added to the feature list at the end of each `pmbdomains` environment. It represents the protein main chain, which appears as a grey line by default. Nevertheless, you can alter the backbone just like any other feature (Example 3.15).

Feature other/name   (*no alias*)
This feature is automatically added to the feature list at the end of each `pmbdomains` environment. It relates to the protein name, which is normally displayed at the top center of the chart, together with the number of residues (Example 3.16). The following auxiliary commands are available for the feature style Ti*k*Z code: `\xLeft`, `\xMid`, `\xRight` and `current style`.

3.6 Disulfides and Ranges

Feature disulfide   (*alias* DISULFID)
`pgfmolbio` indicates disulfide bridges by brackets above the main chain. Since disulfides are often interleaved in linear representations of proteins, the package automatically stacks them in order to avoid overlaps (Example 3.17).

/pgfmolbio/domains/**level** =⟨*number*⟩

Default: (empty)

Manually sets the level of a disulfide feature.

/pgfmolbio/domains/**disulfide base distance** =⟨*number*⟩

Default: **1**

The distance (as a multiple of y-units) between the main chain and the first level.

Example 3.15
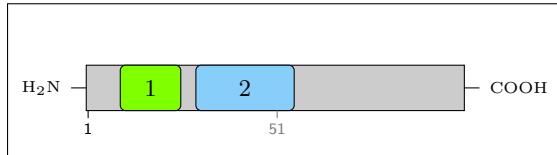

```
1  \setfeatureshape{other/main chain}{%
2    \draw [/pgfmolbio/domains/current style]
3      (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
4      (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
5    \draw (\xLeft, \yMid) --
6      (\xLeft - 2mm, \yMid)
7      node [left] {\tiny H$_2$N};
8    \draw (\xRight, \yMid) --
9      (\xRight + 2mm, \yMid)
10     node [right] {\tiny COOH};
11 }
12 \begin{pmbdomains}%
13     [show name=false, enlarge left=-0.8cm, enlarge right=1.2cm]{100}
14   \setfeaturestyle{other/main chain}{{draw=black,fill=black!20}}
15   \addfeature[description=1]{domain}{10}{25}
16   \addfeature[description=2]{domain}{30}{55}
17 \end{pmbdomains}
```

Example 3.16
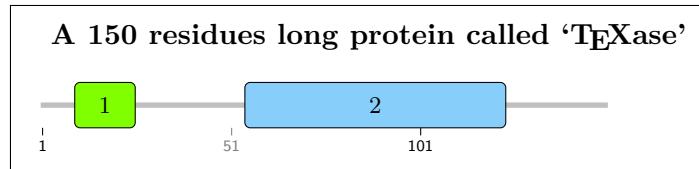

```
1  \setfeatureshape{other/name}{%
2    \node [/pgfmolbio/domains/current style]
3      at (\xLeft, \pmbdomvalueof{baseline skip}
4        * \pmbdomvalueof{y unit} / 2)
5      {A \pmbdomvalueof{sequence length} residues long protein
6        called `\pmbdomvalueof{name}'};
7  }
8  \begin{pmbdomains}[name=\TeX ase]{150}
9    \setfeaturestyle{other/name}{{font=\bfseries, right}}
10   \addfeature[description=1]{domain}{10}{25}
11   \addfeature[description=2]{domain}{55}{123}
12 \end{pmbdomains}
```

Default: `.2`

The space (as a multiple of y-units) between levels (see the figure below).
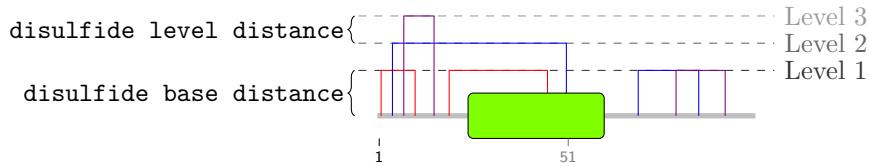
Example 3.17
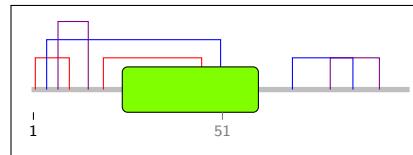

```
1  \begin{pmbdomains}[show name=false,
2      disulfide base distance=.7,
3      disulfide level distance=.4]{100}
4  \setfeaturestyle{disulfide}{draw=red, draw=blue, draw=violet}
5  \addfeature{disulfide}{2}{10}
6  \addfeature{disulfide}{5}{50}
7  \addfeature{disulfide}{8}{15}
8  \addfeature{disulfide}{20}{45}
9  \addfeature[level=1]{disulfide}{70}{85}
10  \addfeature[level=1]{disulfide}{80}{92}
11  \addfeature{domain}{25}{60}
12  \end{pmbdomains}
```

\setdisulfidefeatures{⟨*key list*⟩}
\adddisulfidefeatures{⟨*key list*⟩}
\removedisulfidefeatures{⟨*key list*⟩}

These macros edit the list of "disulfide-like" features, i. e. those subject to the automatic stacking mechanism. **\setdisulfidefeatures** renews this list, replacing any previous contents. **\adddisulfidefeatures** adds the features in its ⟨*key list*⟩ to an existing list, while **\removedisulfidefeatures** removes selected features. By default, there are three disulfide-like features: `disulfide`, `DISULFID` and `range`. Note that **\setfeaturealias** and its relatives do not influence the list.

32

Feature `range`   (*no alias*)

Indicates a range of residues. `range` features are disulfide-like in order to prevent them from overlapping.

/pgfmolbio/domains/**range font** =⟨*font commands*⟩

Default: `\sffamily\scriptsize`

Changes the font for the range label. The last command may take a single argument (Example 3.18).

Example 3.18
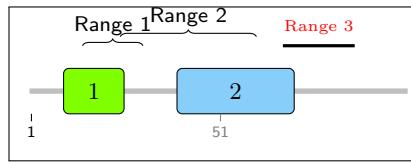

```
1 \begin{pmbdomains}[show name=false]{100}
2   \addfeature[description=1]{domain}{10}{25}
3   \addfeature[description=2]{domain}{40}{70}
4   \addfeature[description=Range 1]{range}{15}{30}
5   \addfeature[description=Range 2]{range}{25}{60}
6   \addfeature[description=Range 3,%
7     style={very thick, draw=black},%
8     range font=\tiny\textcolor{red}]{range}{68}{86}
9 \end{pmbdomains}
```

3.7 Ruler

Feature `other/ruler`   (*no alias*)

This feature is automatically added to the feature list at the end of each `pmbdomains` environment. It draws a ruler below the main chain, which indicates the residue numbers (Example 3.19). The following auxiliary commands are available for the feature style TikZ code: `\xMid`, `\yMid`, `\residueNumber` and `current style`.

/pgfmolbio/domains/**show ruler** =⟨*boolean*⟩

Default: `true`

Determines whether the rule is drawn.

/pgfmolbio/domains/**ruler range** =⟨*ruler range list*⟩

Default: `auto-auto`

The complete syntax for `ruler range` is

> ⟨*ruler range list*⟩ := {⟨*ruler range*⟩[,⟨*ruler range*⟩,...]}
> ⟨*ruler range*⟩ := ⟨*lower*⟩-⟨*upper*⟩[step ⟨*interval*⟩]
> ⟨*lower*⟩ := `auto` | ⟨*number*⟩[⟨*letter*⟩] | (⟨*number*⟩)
> ⟨*upper*⟩ := `auto` | ⟨*number*⟩[⟨*letter*⟩] | (⟨*number*⟩)
> ⟨*interval*⟩ := ⟨*number*⟩

Each ⟨*ruler range*⟩ tells the package to mark every ⟨*interval*⟩th residue from ⟨*lower*⟩ to ⟨*upper*⟩ by an `other/ruler` feature; the `step` part is optional. Possible values for ⟨*lower*⟩ and ⟨*upper*⟩ are:

- `auto`, which indicates the leftmost or rightmost residue shown, respectively;

- a plain number (with an optional letter), which denotes a residue in the *relative* numbering scheme set by `residue numbering`;

- a parenthesized number, which denotes a residue in the *absolute* numbering scheme.

/pgfmolbio/domains/**default ruler step size =**⟨*number*⟩

Default: `50`

Step size for a ⟨*ruler range*⟩ that lacks the optional `step` part.

/pgfmolbio/domains/**ruler distance =**⟨*factor*⟩

Default: `-.5`

Separation (multiples of the *y*-unit) between ruler and main chain (Example 3.19).

3.8 Sequences

/pgfmolbio/domains/**sequence =**⟨*sequence*⟩

Default: empty

Sets the amino acid ⟨*sequence*⟩ of a protein (single-letter abbreviations).

Feature `other/sequence`   (*no alias*)
Displays a sequence which is vertically centered at the main chain. Since a residue is only 0.5 mm wide by default, you should increase the `x unit` when showing `sequence` features (Example 3.20).

Example 3.19
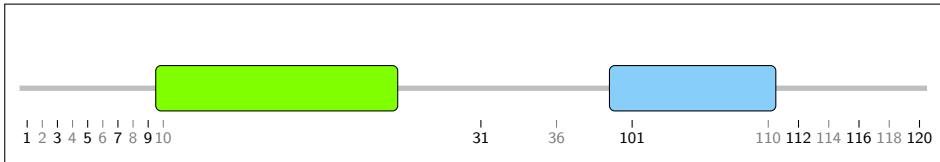

```
1  \begin{pmbdomains}[x unit=2mm,
2      show name=false,
3      residue numbering={1-40,101-120},
4      ruler range={auto-10 step 1, 31-(41), 110-120 step 2},
5      default ruler step size=5,
6      ruler distance=-.7]{60}
7    \addfeature{domain}{10}{25}
8    \addfeature{domain}{40}{(50)}
9  \end{pmbdomains}
```

Example 3.20
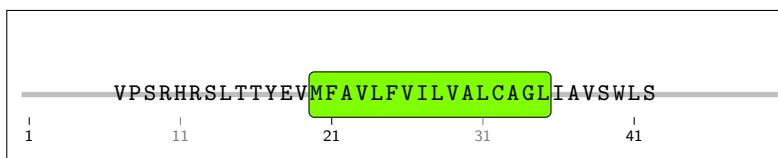

```
1  \begin{pmbdomains}[%
2      sequence=MGSKRSVPSRHRSLTTYEVMFAVLFVILV%
3        ALCAGLIAVSWLSIQGSVKDAAFGKSHEARGTL,
4      residues per line=50,
5      x unit=2mm, show name=false,
6      ruler range=auto-auto step 10]{50}
7    \setfeaturestyle{other/sequence}{font=\ttfamily\footnotesize}
8    \addfeature{domain}{20}{35}
9    \addfeature{other/sequence}{7}{42}
10 \end{pmbdomains}
```

```
\setfeatureprintfunction{⟨key list⟩}{⟨Lua function⟩}
\removefeatureprintfunction{⟨key list⟩}
\pmbdomdrawfeature{⟨type⟩}
```

Some features require sophisticated coordinate calculations. Hence, you might ocasionally want to call a Lua function as "preprocessor" before executing the ⟨*TikZ code*⟩ of `\setfeatureshape`. For this purpose, `\setfeatureprintfunction` registers such a ⟨*Lua function*⟩ and `\removefeatureprintfunction` deletes the preprocessing function(s) for all features in the ⟨*key list*⟩.

A suitable Lua function

- receives up to six arguments in the following order (see also section 5.6.1):

    1. A table describing the feature (see section 5.6.3 for its fields);

    2. the left x-coordinate of the feature (an integer);

    3. its right x-coordinate (an integer);

    4. the y-coordinate of the current line (an integer);

    5. the dimension stored in `x unit`, converted to scaled points (an integer);

    6. the dimension stored in `y unit`, converted to scaled points (an integer);

- performs all necessary calculations and defines all TeX macros required by `\setfeatureshape`;

- may execute `\pmbdomdrawfeature` with the appropriate feature ⟨*type*⟩ to draw the feature.

Example 3.21 devises a new print function, `printFunnySequence` (lines 2–17). It is similar to the default print function for `other/sequence` features, but adds random values to the y-coordinate of the individual letters.

`printFunnySequence` is a function with six arguments (line 2). We add the width of half a residue to the left x-coordinate, `xLeft` (line 3), since each letter should be horizontally centered. We iterate over each letter in the `sequence` field of the `feature` table (lines 4–16). In each loop, calculated coordinates are stored in the TeX macros `\xMid` (lines 5–7) and `\yMid` (lines 8–10). The construction `\string\\...` is expanded to `\\...` when `tex.sprint` passes its argument back to TeX. `pgfmolbio.dimToString` converts a number representing a dimension in scaled points to a string (e.g., 65536 to "1pt", see section 5.2). The letter of the current residue is stored in `\currentResidue` (lines 11–13). Finally, each letter is drawn by calling `\pmbdomdrawfeature{other/sequence}` (line 14), and the x-coordinate increases by one (line 15). Line 25 registers `printFunnySequence` for `other/sequence` features.

Example 3.21
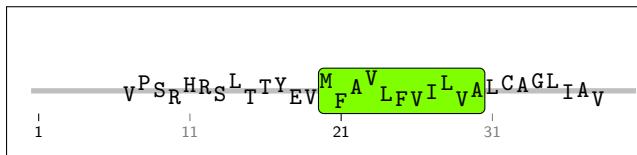

```latex
\directlua{
  function printFunnySequence (feature, xLeft, xRight, yMid, xUnit, yUnit)
    xLeft = xLeft + 0.5
    for currResidue in feature.sequence:gmatch(".") do
      tex.sprint("\string\\def\string\\xMid{" ..
        pgfmolbio.dimToString(xLeft * xUnit) ..
        "}")
      tex.sprint("\string\\def\string\\yMid{" ..
        pgfmolbio.dimToString((yMid + math.random(-5, 5) / 20) * yUnit) ..
        "}")
      tex.sprint("\string\\def\string\\currentResidue{" ..
        currResidue ..
        "}")
      tex.sprint("\string\\pmbdomdrawfeature{other/sequence}")
      xLeft = xLeft + 1
    end
  end
}

\begin{pmbdomains}[%
    sequence=MGSKRSVPSRHRSLTTYEVMFAVLFVILVALCAGLIAVSWLSIQGSVKDAAF,
    x unit=2mm, show name=false,
    ruler range=auto-auto step 10]{40}
  \setfeaturestyle{other/sequence}{font=\ttfamily\footnotesize}
  \setfeatureprintfunction{other/sequence}{printFunnySequence}
  \addfeature{domain}{20}{30}
  \addfeature{other/sequence}{7}{38}
\end{pmbdomains}
```

Feature `other/magnified sequence above` (*no alias*)
Displays its sequence as a single string above the main chain, with dashed lines indicating the sequence start and stop on the backbone. This feature allows you to show sequences without the need to increase the `x unit`.

Feature `other/magnified sequence below` (*no alias*)
Displays the sequence *below* the backbone.

/pgfmolbio/domains/**magnified sequence font** =⟨*font commands*⟩

Default: `\ttfamily\footnotesize`

The font used for a magnified sequence (Example 3.22).

Example 3.22
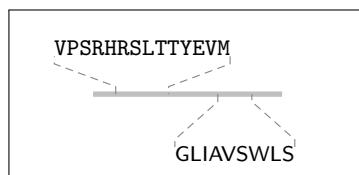

```
1 \begin{pmbdomains}[%
2     sequence=MGSKRSVPSRHRSLTTYEVMFAVLFVIL%
3       VALCAGLIAVSWLSIQGSVKDAAFGKSHEARGTL,
4     enlarge left=-1cm, enlarge right=1cm, enlarge bottom=-1cm,
5     show name=false, show ruler=false]{50}
6   \addfeature{other/magnified sequence above}{7}{20}
7   \addfeature[magnified sequence font=\scriptsize\sffamily]%
8     {other/magnified sequence below}{34}{42}
9 \end{pmbdomains}
```

3.9 Secondary Structure

/pgfmolbio/domains/**show secondary structure** =⟨*boolean*⟩

Default: `false`

Determines whether the secondary structure is shown.

/pgfmolbio/domains/**secondary structure distance** =⟨*factor*⟩

Default: `1`

Secondary structures appear along a thin line $\langle factor \rangle$ times the value of `y unit` above the main chain. In accordance with the categories established by the Dictionary of Protein Secondary Structure[5], pgfmolbio provides seven features for displaying secondary structure types (Example 3.23):

Example 3.23
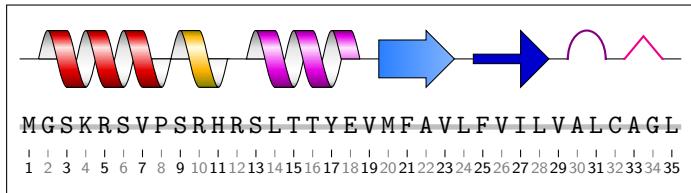

```
1  \begin{pmbdomains}[%
2      show name=false,
3      sequence=MGSKRSVPSRHRSLTTYEVMFAVLFVILVALCAGL,
4      x unit=2.5mm,
5      enlarge top=1.5cm,
6      ruler range=auto-auto step 1,
7      show secondary structure=true,
8      secondary structure distance=1.5
9  ]{35}
10  \setfeaturestyle{other/sequence}{{font=\ttfamily\small}}
11  \addfeature{alpha helix}{2}{8}
12  \addfeature{pi helix}{9}{11}
13  \addfeature{310 helix}{13}{18}
14  \addfeature{beta strand}{20}{23}
15  \addfeature{beta bridge}{25}{28}
16  \addfeature{beta turn}{30}{31}
17  \addfeature{bend}{33}{34}
18  \addfeature{other/sequence}{1}{35}
19  \end{pmbdomains}
```

Feature `alpha helix`   (*alias* `HELIX`)
Shows an α-helix.

Feature `pi helix`   (*no alias*)
Shows a π-helix.

Feature `310 helix`   (*no alias*)
Shows a 3_{10}-helix.

---

[5]Kabsch, W. and Sander, C. (1983). Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers* **22**(12), 2577–2637.

Figure 3.1: Shading colors of helix features.

Name	*xcolor definition*		
	α-helix	π-helix	3_{10}-helix
`helix back border color`		white!50!black ■	
`helix back main color`		white!90!black ▪	
`helix back middle color`		white	
`helix front border color`	red!50!black ■	yellow!50!black ■	magenta!50!black ■
`helix front main color`	red!90!black ■	yellow!70!red ■	magenta!90!black ■
`helix front middle color`	red!10!white ▪	yellow!10!white ▪	magenta!10!white ▪

```
helix back border color ⟶                    ⟵ helix front border color
                                              ⟵ helix front main color
   helix back main color ⟶                    ⟵ helix front middle color

 helix back middle color ⟶                    ⟵ helix front main color
   helix back main color ⟶
 helix back border color ⟶                    ⟵ helix front border color
```
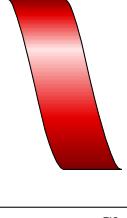
Shading `helix full back` Shading `helix full front`

Feature `beta strand`   (*alias* `STRAND`)
Shows a β-strand.

Feature `beta turn`   (*alias* `TURN`)
Shows a β-turn.

Feature `beta bridge`   (*no alias*)
Shows a β-bridge.

Feature `bend`   (*no alias*)
Shows a bend.

While changing the appearance of nonhelical secondary structure elements is simple, the complex helical features employ the print function `printHelixFeature` (section 5.6.1). However, their appearance can be customized on several levels:

1. The elements of a helical feature are drawn by five "subfeatures", which are called by `printHelixFeature` (Table 3.1a).

2. For each subfeature, there is a corresponding shading (Table 3.1b; see section 5.5.3 and section 83 of the TikZ manual for their definitions).

3. These shadings use six colors in total, three for front and three for back shadings (Figure 3.1). For each color, there is a key of the same name, so you can change helix colors in feature style lists (Example 3.24).

Table 3.1: Customizing helices in the domains module.

(a) Subfeatures	(b) Corresponding shadings	(c) Coordinates	
helix/half upper back	helix half upper back	\xLeft	\yMid
helix/half lower back	helix half lower back	\xRight	\yMid
helix/full back	helix full back	\xMid	\yLower
helix/half upper front	helix half upper front	\xRight	\yMid
helix/full front	helix full front	\xMid	\yLower

Example 3.24
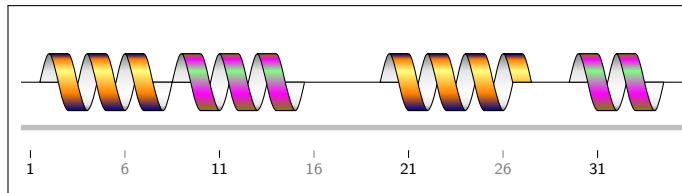

```
1  \begin{pmbdomains}[%
2      show name=false,
3      x unit=2.5mm,
4      enlarge top=1.5cm,
5      ruler range=auto-auto step 5,
6      show secondary structure
7  ]{35}
8  \setfeaturestyle{alpha helix}{%
9      *1{helix front border color=blue!50!black,%
10     helix front main color=orange,%
11     helix front middle color=yellow!50},%
12     *1{helix front border color=olive,%
13     helix front main color=magenta,%
14     helix front middle color=green!50}%
15  }
16  \addfeature{alpha helix}{2}{8}
17  \addfeature{alpha helix}{9}{15}
18  \addfeature{alpha helix}{20}{27}
19  \addfeature{alpha helix}{30}{34}
20 \end{pmbdomains}
```

Example 3.25

```
1  \pgfmathsetmacro\yShift{%
2    \pmbdomvalueof{secondary structure distance}
3      * \pmbdomvalueof{y unit}%
4  }
5
6  \setfeatureshape{helix/half upper back}{%
7    \draw [shading=helix half upper back]
8      (\xLeft, \yMid + \yShift pt) --
9      (\xLeft + .5 * \pmbdomvalueof{x unit},
10       \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
11     (\xLeft + 1.5 * \pmbdomvalueof{x unit},
12       \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
13     (\xLeft + \pmbdomvalueof{x unit}, \yMid + \yShift pt) --
14     cycle;
15 }
16
17 \setfeatureshape{helix/half lower back}{%
18   \draw [shading=helix half lower back]
19     (\xRight, \yMid + \yShift pt) --
20     (\xRight - .5 * \pmbdomvalueof{x unit},
21       \yMid - 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
22     (\xRight - 1.5 * \pmbdomvalueof{x unit},
23       \yMid - 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
24     (\xRight - \pmbdomvalueof{x unit}, \yMid + \yShift pt) --
25     cycle;
26 }
27
28 \setfeatureshape{helix/full back}{%
29   \draw [shading=helix full back]
30     (\xMid, \yLower + \yShift pt) --
31     (\xMid - \pmbdomvalueof{x unit}, \yLower + \yShift pt) --
32     (\xMid, \yLower + 3 * \pmbdomvalueof{x unit} + \yShift pt) --
33     (\xMid + \pmbdomvalueof{x unit},
34       \yLower + 3 * \pmbdomvalueof{x unit} + \yShift pt) --
35     cycle;
36 }
37
38 \setfeatureshape{helix/half upper front}{%
39   \draw [shading=helix half upper front]
40     (\xRight, \yMid + \yShift pt) --
41     (\xRight - .5 * \pmbdomvalueof{x unit},
42       \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
43     (\xRight - 1.5 * \pmbdomvalueof{x unit},
44       \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
45     (\xRight - \pmbdomvalueof{x unit}, \yMid + \yShift pt) --
46     cycle;
47 }
48
49 \setfeatureshape{helix/full front}{%
50   \draw [shading=helix full front]
51     (\xMid, \yLower + \yShift pt) --
52     (\xMid + \pmbdomvalueof{x unit}, \yLower + \yShift pt) --
53     (\xMid, \yLower + 3 * \pmbdomvalueof{x unit} + \yShift pt) --
54     (\xMid - \pmbdomvalueof{x unit},
55       \yLower + 3 * \pmbdomvalueof{x unit} + \yShift pt) --
56     cycle;
57 }
58
59 \begin{pmbdomains}[%
60     show name=false, sequence=MGSKRSVPSR,
61     x unit=2.5mm, enlarge top=1.5cm,
62     ruler range=auto-auto step 1,
63     show secondary structure
64   ]{10}
65   \setfeaturestyle{other/sequence}{{font=\ttfamily\small}}
66   \addfeature{alpha helix}{2}{6}
67   \addfeature{alpha helix}{8}{9}
68   \addfeature{other/sequence}{1}{10}
69 \end{pmbdomains}
```

3.10 File Input

\inputuniprot{⟨*Uniprot file*⟩}
\inputgff{⟨*gff file*⟩}

Include the features defined in an ⟨*Uniprot file*⟩ or ⟨*gff file*⟩, respectively (Example 3.26). These macros are only defined in `pmbdomains`.

Example 3.26
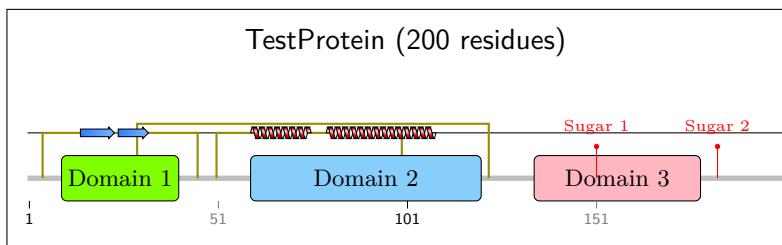

```
1 \begin{pmbdomains}[show secondary structure]{}
2   \setfeaturestyle{disulfide}{{draw=olive,thick}}
3   \inputuniprot{SampleUniprot.txt}
4 \end{pmbdomains}
```
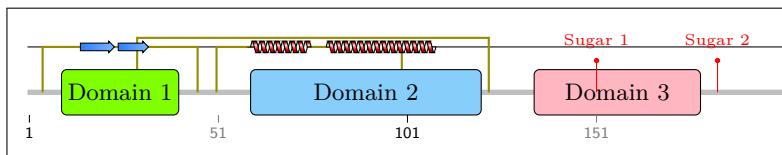


```
1 \begin{pmbdomains}[show name=false,show secondary structure]{200}
2   \setfeaturestyle{disulfide}{{draw=olive,thick}}
3   \inputgff{SampleGff.gff}
4 \end{pmbdomains}
```

`/pgfmolbio/domains/`**sequence length =**⟨*number*⟩

Default: (empty)

Note that in Example 3.26, we had to set a sequence length for the `pmbdomains` environment that contains the `\inputgff` macro. `gff` files lack a sequence length field. By contrast, `pgfmolbio` reads the sequence length from an Uniprot file, and thus the mandatory argument of `pmbdomains` may remain empty. In general, the sequence length is stored in the key of the same name.

4 The **convert** module

4.1 Overview

The convert module supports users who wish to include `pgfmolbio` graphs, but who do not want to typeset their documents with a TeX engine that implements Lua. To this end, the convert workflow comprises two steps: (1) Running LuaLaTeX on an input file that contains at least one `\pmbchromatogram` or similar macros/environments. This will generate one `tex` file per graph macro/environment that contains only TikZ commands. (2) Including this file in another TeX document (via `\input`) which is then processed by any TeX engine that supports TikZ.

4.2 Converting Chromatograms

In order to create the external TikZ file, run an input file like the one below through LuaLaTeX:

```
1  \documentclass{article}
2  \usepackage[chromatogram,convert]{pgfmolbio}
3
4  \begin{document}
5    \pmbchromatogram[sample range=base 50-base 60]{SampleScf.scf}
6    \pmbchromatogram[/pgfmolbio/convert/output file name=mytikzfile]%
7      {SampleScf.scf}
8    \pmbchromatogram[sample range=base 60-base 70]{SampleScf.scf}
9  \end{document}
```

The convert module disables `pdf` output and introduces the following keys:

/pgfmolbio/convert/**output file name** =⟨*text*⟩

   Default: (auto)

/pgfmolbio/convert/**output file extension** =⟨*text*⟩

   Default: `tex`

With the default value for `output file name` ("(auto)"), `pgfmolbio` creates files that are named `pmbconverted` and numbered consecutively (`pmbconverted0.tex`, `pmbconverted1.tex` etc.). Both keys can be changed locally (e.g., in the optional argument of `\pmbchromatogram`), but this turns off automatic numbering.

The code above produces the files `pmbconverted0.tex`, `mytikzfile.tex` and `pmbconverted2.tex`. Below is an annotated excerpt from `pmbconverted0.tex`:

```
1  \begin{tikzpicture}
2    [canvas section]
3    \draw [/pgfmolbio/chromatogram/canvas style] (0mm, -0mm) rectangle (25mm, 20mm);
4    [traces section]
5    \draw [/pgfmolbio/chromatogram/trace A style] (0mm, 6.37mm) -- (0.2mm, 6.66mm) -- [many
        coordinates] -- (25mm, 0mm);
6    \draw [/pgfmolbio/chromatogram/trace C style] (0mm, 0.06mm) -- (0.2mm, 0.05mm) -- [...] --
        (25mm, 6.27mm);
7    \draw [/pgfmolbio/chromatogram/trace G style] (0mm, 0.01mm) -- (0.2mm, 0.01mm) -- [...] --
        (25mm, 0.05mm);
8    \draw [/pgfmolbio/chromatogram/trace T style] (0mm, 0mm) -- (0.2mm, 0mm) -- [...] -- (25mm,
        0.06mm);
9    [ticks/base labels/probabilities section]
10   \draw [/pgfmolbio/chromatogram/tick A style] (0mm, -0mm) -- (0mm, -1mm) node [/pgfmolbio/
        chromatogram/base label A style] {\pgfkeysvalueof{/pgfmolbio/chromatogram/base label A
        text}} node [/pgfmolbio/chromatogram/base number style] {\strut 50};
11   \draw [ultra thick, pmbTraceGreen] (0mm, -8mm) -- (0.9mm, -8mm);
12   \draw [/pgfmolbio/chromatogram/tick T style] (1.8mm, -0mm) -- (1.8mm, -1mm) node [/
        pgfmolbio/chromatogram/base label T style] {\pgfkeysvalueof{/pgfmolbio/chromatogram/base
        label T text}};
13   \draw [ultra thick, pmbTraceGreen] (0.9mm, -8mm) -- (3mm, -8mm);
14   \draw [/pgfmolbio/chromatogram/tick A style] (4.2mm, -0mm) -- (4.2mm, -1mm) node [/
        pgfmolbio/chromatogram/base label A style] {\pgfkeysvalueof{/pgfmolbio/chromatogram/base
        label A text}};
15   \draw [ultra thick, pmbTraceGreen] (3mm, -8mm) -- (5.4mm, -8mm);
16   [...]
17   [more ticks, base labels and probability rules]
18 \end{tikzpicture}
```

You can change the format of the coordinates by the following keys:

/pgfmolbio/**coordinate unit** =⟨*unit*⟩

Default: `mm`

/pgfmolbio/**coordinate format string** =⟨*format string*⟩

Default: `%s%s`

`pgfmolbio` internally calculates dimensions in scaled points, but usually converts them before returning them to TeX. To this end, it selects the ⟨*unit*⟩ stored in `coordinate unit` (any of the standard TeX units of measurement: `bp`, `cc`, `cm`, `dd`, `in`, `mm`, `pc`, `pt` or `sp`). In addition, the package formats the dimension according to the ⟨*format string*⟩ given by `coordinate format string`. This string basically follows the syntax of C's `printf` function, as described in the Lua reference manual.

Depending on the values of `coordinate unit` and `coordinate format string`, dimensions will be printed in different ways (Table 4.1).

The output files can be included in a file which is processed by pdfLaTeX:

```
1  \documentclass{article}
2  \usepackage[chromatogram]{pgfmolbio}
```

Table 4.1: Effects of `coordinate unit` and `coordinate format string` when converting an internal pgfmolbio dimension of 200000 [sp].

	Values	Output	Notes
sp	%s%s	200000sp	simple conversion
mm	%s%s	1.072570201154mm	default settings, may lead to a large number of decimal places
mm	%.3f%s	1.073mm	round to three decimal places
cm	%.3f	0.107	don't print any unit, i. e. use TikZ's xyz coordinate system

```
3
4  \begin{document}
5    \input{pmbconverted.tex}
6  \end{document}
```

Several keys of the chromatogram module must contain their final values before conversion, while others can be changed afterwards, i. e., before the generated file is loaded with \input (Table 4.2).

Table 4.2: Keys of the chromatogram module that require final values prior to conversion.

Required		Not required
base labels drawn	sample range	base label style
base number range	samples per line	base label X style
baseline skip	show base numbers	base label X text
bases drawn	tick length	base number style
canvas height	ticks drawn	canvas style
probabilities drawn	traces drawn	tick style
probability distance	x unit	tick X style
probability style function	y unit	trace style
		trace X style

4.3 Converting Domain Diagrams

/pgfmolbio/convert/**output code** =pgfmolbio | tikz

Default: `tikz`

In principle, domain diagrams are converted like sequencing chromatograms (section 4.2). However, `output code` lets you choose the kind of code **convert** writes to the output file: **pgfmolbio** generates a pmbdomains environment containing \addfeature commands, `tikz` produces TikZ code.

"Converting" one `pmbdomains` environment in the input file to another one in the output file might seem pointless. Nonetheless, this conversion mechanism can be highly useful for extracting features from a Uniprot or `gff` file. For example, consider the following input file:

```
1  \documentclass{article}
2  \usepackage[domains,convert]{pgfmolbio}
3
4  \begin{document}
5    \pgfmolbioset[convert]{output code=pgfmolbio}
6    \begin{pmbdomains}{}
7      \inputuniprot{SampleUniprot.txt}
8    \end{pmbdomains}
9  \end{document}
```

The corresponding output is

```
1  \begin{pmbdomains}
2      [name={TestProtein},
3      sequence=MGSKRSVPSRHRSL[...]PLATPGNVSIECP]{200}
4    \addfeature[description={Disulfide 1}]{DISULFID}{5}{45}
5    \addfeature[description={Disulfide 2}]{DISULFID}{30}{122}
6    \addfeature[description={Disulfide 3}]{DISULFID}{51}{99}
7    \addfeature[description={Domain 1}]{DOMAIN}{10}{40}
8    \addfeature[description={Domain 2}]{DOMAIN}{60}{120}
9    \addfeature[description={Domain 3}]{DOMAIN}{135}{178}
10   \addfeature[description={Strand 1}]{STRAND}{15}{23}
11   \addfeature[description={Strand 2}]{STRAND}{25}{32}
12   \addfeature[description={Helix 1}]{HELIX}{60}{75}
13   \addfeature[description={Helix 2}]{HELIX}{80}{108}
14   \addfeature[description={Sugar 1}]{CARBOHYD}{151}{151}
15   \addfeature[description={Sugar 2}]{CARBOHYD}{183}{183}
16  \end{pmbdomains}
```

Obviously, this method is particularly suitable for Uniprot files containing many features.

/pgfmolbio/convert/`include description` =⟨*boolean*⟩

Default: `true`

Decides whether the feature description obtained from the input should appear in the output. Since the description field in FT entries of Uniprot files can be quite long, you may not wish to show it in the output. For example, the output of the example above with `include description`=`false` looks like

```
1  \begin{pmbdomains}
2      [name={TestProtein},
3      sequence=MGSKRSVPSRHRSL[...]PLATPGNVSIECP]{200}
4    \addfeature{DISULFID}{5}{45}
5    \addfeature{DISULFID}{30}{122}
```

```
6    \addfeature{DISULFID}{51}{99}
7    [...]
8  \end{pmbdomains}
```

With `output code=tikz`, we obtain the following (annotated) output file:

```
1   [set relevant keys]
2   \pgfmolbioset[domains]{name={TestProtein},sequence={MGSKRS[...]VSIECP},sequence length=200}
3   [the actual TikZ picture]
4   \begin{tikzpicture}
5     [each feature appears within its own scope]
6     \begin{scope}\begin{pgfinterruptboundingbox}
7       \def\xLeft{0mm}
8       \def\xMid{50mm}
9       \def\xRight{100mm}
10      \def\yMid{-0mm}
11      \def\featureSequence{MGSKRS[...]VSIECP}
12      \clip (-50mm, \yMid + 100mm) rectangle (150mm, \yMid - 100mm);
13      \pgfmolbioset[domains]{style={{draw, line width=2pt, black!25}},@layer=1}
14      \pmbdomdrawfeature{other/main chain}
15    \end{pgfinterruptboundingbox}\end{scope}
16    [more features]
17    [...]
18    [helix features require additional drawing commands]
19    \begin{scope}\begin{pgfinterruptboundingbox}
20      \def\xLeft{29.5mm}
21      \def\xMid{33.5mm}
22      \def\xRight{37.5mm}
23      \def\yMid{-0mm}
24      \def\featureSequence{GTLKIISGATYNPHLQ}
25      \clip (-50mm, \yMid + 100mm) rectangle (87.5mm, \yMid - 100mm);
26      \pgfmolbioset[domains]{style={{helix front border color=red!50!black,helix front main
         color=red!90!black,helix front middle color=red!10!white}},description={Helix 1}}
27      \pgfmolbioset[domains]{current style}
28      \def\xLeft{29.5mm}
29      \def\yMid{-0mm}
30      \pmbdomdrawfeature{helix/half upper back}
31      \def\xMid{30.75mm}
32      \def\yLower{-0.75mm}
33      \pmbdomdrawfeature{helix/full back}
34      [more helix parts]
35    \end{pgfinterruptboundingbox}\end{scope}
36    [...]
37    [ruler section]
38    \begin{scope}
39      \pgfmolbioset[domains]{current style/.style={black}}
40        \def\xMid{0.25mm}
41        \let\xLeft\xMid\let\xRight\xMid
42        \def\yMid{-0mm}
43        \def\residueNumber{1}
44        \pmbdomdrawfeature{other/ruler}
45      \pgfmolbioset[domains]{current style/.style={black!50}}
46        \def\xMid{25.25mm}
47        \let\xLeft\xMid\let\xRight\xMid
48        \def\yMid{-0mm}
49        \def\residueNumber{51}
50        \pmbdomdrawfeature{other/ruler}
51      [more ruler numbers]
52      [...]
53    \end{scope}
54    [name section]
```

```
55    \begin{scope}
56      \pgfmolbioset[domains]{current style/.style={font=\sffamily }}
57      \def\xLeft{0mm}
58      \def\xMid{50mm}
59      \def\xRight{100mm}
60      \def\yMid{0mm}
61      \pmbdomdrawfeature{other/name}
62    \end{scope}
63    [adjust picture size]
64    \pmbprotocolsizes{\pmbdomvalueof{enlarge left}}{\pmbdomvalueof{enlarge top}}
65    \pmbprotocolsizes{100mm + \pmbdomvalueof{enlarge right}}{-0mm + \pmbdomvalueof{enlarge
        bottom}}
66  \end{tikzpicture}
```

Several keys of the domains module must contain their final values before conversion, and some macros can't be used afterwards (Table 4.3).

Table 4.3: Keys and macros of the domain module that require final values prior to conversion or can't be used afterwards, respectively.

Required		*Not required*
baseline skip	ruler distance	domain font
default ruler step size	ruler range	enlarge bottom
description	secondary structure distance	enlarge left
disulfide base distance	sequence	enlarge right
disulfide level distance	sequence length	enlarge top
level	show ruler	magnified sequence font
name	style	range font
residue numbering	x unit	show secondary structure
residue range	y unit	
residues per line		
\adddisulfidefeatures	\setfeatureprintfunction	\setfeaturealias
\removedisulfidefeatures	\setfeaturestyle	\setfeatureshape
\removefeatureprintfunction	\setfeaturestylealias	\setfeatureshapealias
\setdisulfidefeatures		

5 Implementation

5.1 `pgfmolbio.sty`

The options for the main style file determine which module(s) should be loaded.

```
1.67  \newif\ifpmb@loadmodule@chromatogram
1.68  \newif\ifpmb@loadmodule@domains
1.69  \newif\ifpmb@loadmodule@convert
1.70
1.71  \DeclareOption{chromatogram}{%
1.72     \pmb@loadmodule@chromatogramtrue%
1.73  }
1.74  \DeclareOption{domains}{%
1.75     \pmb@loadmodule@domainstrue%
1.76  }
1.77  \DeclareOption{convert}{%
1.78     \pmb@loadmodule@converttrue%
1.79  }
1.80
1.81  \ProcessOptions
1.82
```

The main style file also loads the following packages and TikZ libraries.

```
1.83  \RequirePackage{ifluatex}
1.84  \ifluatex
1.85     \RequirePackage{luatexbase-modutils}
1.86        \RequireLuaModule{lualibs}
1.87        \RequireLuaModule{pgfmolbio}
1.88  \fi
1.89  \RequirePackage[svgnames,dvipsnames]{xcolor}
1.90  \RequirePackage{tikz}
1.91     \usetikzlibrary{positioning,svg.path}
1.92
```

`\pgfmolbioset`

#1: The ⟨*module*⟩ to which the options apply.
#2: A ⟨*key-value list*⟩ which configures the graphs.

```
1.93  \newcommand\pgfmolbioset[2][]{%
1.94     \def\@tempa{#1}%
1.95     \ifx\@tempa\@empty%
1.96        \pgfqkeys{/pgfmolbio}{#2}%
1.97     \else%
1.98        \pgfqkeys{/pgfmolbio/#1}{#2}%
1.99     \fi%
1.100 }
1.101
```

We introduce two package-wide keys.

```
1.102 \pgfkeyssetvalue{/pgfmolbio/coordinate unit}{mm}
1.103 \pgfkeyssetvalue{/pgfmolbio/coordinate format string}{\%s\%s}
1.104
```

Furthermore, we define two scratch token registers. Strictly speaking, the two conditionals belong to the convert module, but all modules need to know them.

```
1.105 \newtoks\@pmb@toksa
1.106 \newtoks\@pmb@toksb
1.107 \newif\ifpmb@con@includedescription
1.108 \newif\ifpmb@con@outputtikzcode
1.109
```

\pmbprotocolsizes

#1: x-coordinate.

#2: y-coordinate.

An improved version of \pgf@protocolsizes that accepts coordinate calculations.

```
1.110 \def\pmbprotocolsizes#1#2{%
1.111    \pgfpoint{#1}{#2}%
1.112    \pgf@protocolsizes{\pgf@x}{\pgf@y}%
1.113 }
1.114
```

Finally, we load the modules requested by the user.

```
1.115 \ifpmb@loadmodule@chromatogram
1.116    \input{pgfmolbio.chromatogram.tex}
1.117 \fi
1.118 \ifpmb@loadmodule@domains
1.119    \input{pgfmolbio.domains.tex}
1.120 \fi
1.121 \ifpmb@loadmodule@convert
1.122    \input{pgfmolbio.convert.tex}
1.123 \fi
```

5.2 `pgfmolbio.lua`

Identification of the Lua module.

```
2.1   if luatexbase then
2.2     luatexbase.provides_module({
2.3       name           = "pgfmolbio",
2.4       version        = 0.2,
2.5       date           = "2012/10/01",
2.6       description    = "Molecular biology graphs wit LuaLaTeX",
2.7       author         = "Wolfgang Skala",
2.8       copyright      = "Wolfgang Skala",
2.9       license        = "LPPL",
2.10    })
2.11  end
2.12
```

setCoordinateFormat sets the output format of dimToString (see below). Both its parameters `unit` and `fmtString` are strings, which correspond to the values of coordinate unit and coordinate format string.

```
2.13  local coordUnit, coordFmtStr
2.14
2.15  function setCoordinateFormat(unit, fmtString)
2.16    coordUnit = unit
2.17    coordFmtStr = fmtString
2.18  end
2.19
```

stringToDim converts a string describing a TeX dimension to a number corresponding to scaled points. dimToString converts a dimension in scaled points to a string, formatting it according to the values of the local variables `coordUnit` and `coordFmtString`.

```
2.20  function stringToDim(x)
2.21    if type(x) == "string" then
2.22      return dimen(x)[1]
2.23    end
2.24  end
2.25
2.26  function dimToString(x)
2.27    return number.todimen(x, coordUnit, coordFmtStr)
2.28  end
2.29
```

getRange extracts a variable number of strings from `rangeInput` by applying the regular expressions in its arg table. `rangeInput` contains the values of any of the ... range keys.

```
2.30  function getRange(rangeInput, ...)
2.31    if type(rangeInput) ~= "string" then return end
2.32    local result = {}
2.33    for _, v in ipairs(arg) do
2.34      if type(v) ~= "string" then return end
2.35      table.insert(result, rangeInput:match(v))
2.36    end
2.37    return unpack(result)
2.38  end
2.39
```

packageWarning and packageError throw TEX warnings and errors, respectively. packageError also sets the global variable errorCatched to true. Some Lua functions check the value of this variable and terminate if an error has occurred.

```
2.40  function packageWarning(message)
2.41    tex.sprint("\\PackageWarning{pgfmolbio}{" .. message .. "}")
2.42  end
2.43
2.44  function packageError(message)
2.45    tex.error("Package pgfmolbio Error: " .. message)
2.46    errorCatched = true
2.47  end
2.48
2.49  errorCatched = false
2.50
```

We extend the string table by the function string.trim, which removes leading and trailing spaces.

```
2.51  if not string.trim then
2.52    string.trim = function(self)
2.53      return self:match("^%s*(.-)%s*$")
2.54    end
2.55  end
2.56
```

outputFileId is a counter to enumerate several output files by the convert module.

```
2.57  outputFileId = 0
```

5.3 `pgfmolbio.chromatogram.tex`

Since the Lua script of the chromatogram module does the bulk of the work, we can keep the TEX file relatively short.

```
3.1  \ifluatex
3.2    \RequireLuaModule{pgfmolbio.chromatogram}
3.3  \fi
3.4
```

We define five custom colors for the traces and probability indicators (see Table 2.1).

```
3.5   \definecolor{pmbTraceGreen}{RGB}{34,114,46}
3.6   \definecolor{pmbTraceBlue}{RGB}{48,37,199}
3.7   \definecolor{pmbTraceBlack}{RGB}{0,0,0}
3.8   \definecolor{pmbTraceRed}{RGB}{191,27,27}
3.9   \definecolor{pmbTraceYellow}{RGB}{233,230,0}
3.10
```

\@pmb@chr@keydef

#1: ⟨key⟩ name
#2: default ⟨value⟩

Most of the keys simply store their value. \@pmb@chr@keydef simplifies the declaration of such keys by calling \pgfkeyssetvalue with the appropriate path, ⟨key⟩ and ⟨value⟩.

```
3.11  \def\@pmb@chr@keydef#1#2{%
3.12    \pgfkeyssetvalue{/pgfmolbio/chromatogram/#1}{#2}%
3.13  }
```

\@pmb@chr@stylekeydef

#1: ⟨key⟩ name
#2: default ⟨value⟩

This macro initializes a style ⟨key⟩ with a ⟨value⟩.

```
3.14  \def\@pmb@chr@stylekeydef#1#2{%
3.15    \pgfkeys{/pgfmolbio/chromatogram/#1/.style={#2}}%
3.16  }
```

\@pmb@chr@getkey

#1: ⟨key⟩ name

\@pmb@chr@getkey retrieves the value stored by the ⟨key⟩.

```
3.17  \def\@pmb@chr@getkey#1{%
3.18    \pgfkeysvalueof{/pgfmolbio/chromatogram/#1}%
3.19  }
3.20
```

After providing these auxiliary macros, we define all keys of the chromatogram module.

```
3.21  \@pmb@chr@keydef{sample range}{1-500 step 1}
3.22
3.23  \@pmb@chr@keydef{x unit}{0.2mm}
3.24  \@pmb@chr@keydef{y unit}{0.01mm}
3.25  \@pmb@chr@keydef{samples per line}{500}
3.26  \@pmb@chr@keydef{baseline skip}{3cm}
3.27  \@pmb@chr@stylekeydef{canvas style}{draw=none, fill=none}
3.28  \@pmb@chr@keydef{canvas height}{2cm}
3.29
3.30  \@pmb@chr@stylekeydef{trace A style}{pmbTraceGreen}
3.31  \@pmb@chr@stylekeydef{trace C style}{pmbTraceBlue}
3.32  \@pmb@chr@stylekeydef{trace G style}{pmbTraceBlack}
3.33  \@pmb@chr@stylekeydef{trace T style}{pmbTraceRed}
3.34  \pgfmolbioset[chromatogram]{%
3.35    trace style/.code=\pgfkeysalso{
3.36      trace A style/.style={#1},
3.37      trace C style/.style={#1},
3.38      trace G style/.style={#1},
3.39      trace T style/.style={#1}
3.40    }%
3.41  }
3.42  \@pmb@chr@keydef{traces drawn}{}
3.43
3.44  \@pmb@chr@stylekeydef{tick A style}{thin, pmbTraceGreen}
3.45  \@pmb@chr@stylekeydef{tick C style}{thin, pmbTraceBlue}
3.46  \@pmb@chr@stylekeydef{tick G style}{thin, pmbTraceBlack}
3.47  \@pmb@chr@stylekeydef{tick T style}{thin, pmbTraceRed}
3.48  \pgfmolbioset[chromatogram]{%
3.49    tick style/.code=\pgfkeysalso{
3.50      tick A style/.style={#1},
3.51      tick C style/.style={#1},
3.52      tick G style/.style={#1},
3.53      tick T style/.style={#1}
3.54    }%
3.55  }
3.56  \@pmb@chr@keydef{tick length}{1mm}
3.57  \@pmb@chr@keydef{ticks drawn}{}
3.58
3.59  \@pmb@chr@keydef{base label A text}{\strut A}
3.60  \@pmb@chr@keydef{base label C text}{\strut C}
3.61  \@pmb@chr@keydef{base label G text}{\strut G}
```

```
3.62  \@pmb@chr@keydef{base label T text}{\strut T}
3.63  \@pmb@chr@stylekeydef{base label A style}%
3.64    {below=4pt, font=\ttfamily\footnotesize, pmbTraceGreen}
3.65  \@pmb@chr@stylekeydef{base label C style}%
3.66    {below=4pt, font=\ttfamily\footnotesize, pmbTraceBlue}
3.67  \@pmb@chr@stylekeydef{base label G style}%
3.68    {below=4pt, font=\ttfamily\footnotesize, pmbTraceBlack}
3.69  \@pmb@chr@stylekeydef{base label T style}%
3.70    {below=4pt, font=\ttfamily\footnotesize, pmbTraceRed}
3.71  \pgfmolbioset[chromatogram]{%
3.72    base label style/.code=\pgfkeysalso{
3.73      base label A style/.style={#1},
3.74      base label C style/.style={#1},
3.75      base label G style/.style={#1},
3.76      base label T style/.style={#1}
3.77    }%
3.78  }
3.79  \@pmb@chr@keydef{base labels drawn}{}
3.80
3.81  \newif\ifpmb@chr@showbasenumbers
3.82  \pgfmolbioset[chromatogram]{%
3.83    show base numbers/.is if=pmb@chr@showbasenumbers,
3.84    show base numbers
3.85  }
3.86  \@pmb@chr@stylekeydef{base number style}%
3.87    {pmbTraceBlack, below=-3pt, font=\sffamily\tiny}
3.88  \@pmb@chr@keydef{base number range}{auto-auto step 10}
3.89
3.90  \@pmb@chr@keydef{probability distance}{0.8cm}
3.91  \@pmb@chr@keydef{probabilities drawn}{}
3.92  \@pmb@chr@keydef{probability style function}{nil}
3.93
3.94  \pgfmolbioset[chromatogram]{
3.95    bases drawn/.code=\pgfkeysalso{
3.96      traces drawn=#1,
3.97      ticks drawn=#1,
3.98      base labels drawn=#1,
3.99      probabilities drawn=#1
3.100   },
3.101   bases drawn=ACGT
3.102 }
3.103
```

If `pgfmolbio` is used with a TEX engine that does not support Lua, the package
ends here.

```
3.104 \ifluatex\else\expandafter\endinput\fi
3.105
```

#1: A ⟨*key-value list*⟩ that configures the chromatogram.

#2: The name of an ⟨*scf file*⟩.

If `\pmbchromatogram` appears outside of a `tikzpicture`, we implicitly start this environment, otherwise we begin a new group. "Within a `tikzpicture`" means that `\useasboundingbox` is defined.

```
3.106  \newif\ifpmb@chr@tikzpicture
3.107
3.108  \newcommand\pmbchromatogram[2][]{%
3.109    \@ifundefined{useasboundingbox}%
3.110      {\pmb@chr@tikzpicturefalse\begin{tikzpicture}}%
3.111      {\pmb@chr@tikzpicturetrue\begingroup}%
```

Of course, we consider the ⟨*key-value list*⟩ before drawing the chromatogram.

```
3.112    \pgfmolbioset[chromatogram]{#1}%
```

We generate a new `Chromatogram` object and invoke several Lua functions: (1) `readScfFile` reads the given ⟨*scf file*⟩ (section 5.4.3). (2) `setParameters` passes the values stored by the keys to the Lua script. Note that this function is called twice, since `baseNumberRange` requires that `sampleRange` has been already set, and the implementation of `setParameters` does not ensure this (section 5.4.4). (3) `pgfmolbio.setCoordinateFormat` sets the coordinate output format (section 5.2).

```
3.113    \directlua{
3.114      pmbChromatogram = pgfmolbio.chromatogram.Chromatogram:new()
3.115      pmbChromatogram:readScfFile("#2")
3.116      pmbChromatogram:setParameters{
3.117        sampleRange = "\@pmb@chr@getkey{sample range}",
3.118        xUnit = "\@pmb@chr@getkey{x unit}",
3.119        yUnit = "\@pmb@chr@getkey{y unit}",
3.120        samplesPerLine = "\@pmb@chr@getkey{samples per line}",
3.121        baselineSkip = "\@pmb@chr@getkey{baseline skip}",
3.122        canvasHeight = "\@pmb@chr@getkey{canvas height}",
3.123        tracesDrawn = "\@pmb@chr@getkey{traces drawn}",
3.124        tickLength = "\@pmb@chr@getkey{tick length}",
3.125        ticksDrawn = "\@pmb@chr@getkey{ticks drawn}",
3.126        baseLabelsDrawn = "\@pmb@chr@getkey{base labels drawn}",
3.127        showBaseNumbers = "\ifpmb@chr@showbasenumbers true\else false\fi",
3.128        probDistance = "\@pmb@chr@getkey{probability distance}",
3.129        probabilitiesDrawn = "\@pmb@chr@getkey{probabilities drawn}",
3.130        probStyle = \@pmb@chr@getkey{probability style function}
3.131      }
3.132      pmbChromatogram:setParameters{
3.133        baseNumberRange = "\@pmb@chr@getkey{base number range}",
3.134      }
3.135      pgfmolbio.setCoordinateFormat(
```

```
3.136        "\pgfkeysvalueof{/pgfmolbio/coordinate unit}",
3.137        "\pgfkeysvalueof{/pgfmolbio/coordinate format string}"
3.138      )
```

If the **convert** module is loaded, we open the appropriate output file, change `tex.sprint` so that the function writes to this file and then call `printTikzChromatogram`. Without the **convert** module, `printTikzChromatogram` simply returns the drawing commands for the chromatogram to the TeX input stream (section 5.4.5).

```
3.139      \ifpmb@loadmodule@convert
3.140        local filename =
3.141          "\pgfkeysvalueof{/pgfmolbio/convert/output file name}"
3.142        if filename == "(auto)" then
3.143          filename = "pmbconverted" .. pgfmolbio.outputFileId
3.144        end
3.145        filename = filename ..
3.146          ".\pgfkeysvalueof{/pgfmolbio/convert/output file extension}"
3.147        outputFile, ioError = io.open(filename, "w")
3.148        if ioError then
3.149          tex.error(ioError)
3.150        end
3.151        tex.sprint = function (a) outputFile:write(a) end
3.152        tex.sprint("\string\\begin{tikzpicture}")
3.153        pmbChromatogram:printTikzChromatogram()
3.154        tex.sprint("\string\n\string\\end{tikzpicture}")
3.155        outputFile:close()
3.156        pgfmolbio.outputFileId = pgfmolbio.outputFileId + 1
3.157      \else
3.158        pmbChromatogram:printTikzChromatogram()
3.159      \fi
3.160    }%
```

At the end of `\pmbchromatogram`, we either close the `tikzpicture` or the group, depending on how we started.

```
3.161    \ifpmb@chr@tikzpicture\endgroup\else\end{tikzpicture}\fi%
3.162  }
```

5.4 `pgfmolbio.chromatogram.lua`

This Lua script is the true workhorse of the **chromatogram** module. Remember that the documentation for the Staden package[1] is the definite source for information on the `scf` file format.

```
4.1  if luatexbase then
4.2    luatexbase.provides_module{
```

---
[1] http://staden.sourceforge.net/

58

```
4.3      name           = "pgfmolbio.chromatogram",
4.4      version        = 0.2,
4.5      date           = "2012/10/01",
4.6      description    = "DNA sequencing chromatograms",
4.7      author         = "Wolfgang Skala",
4.8      copyright      = "Wolfgang Skala",
4.9      license        = "LPPL",
4.10   }
4.11 end
4.12
```

5.4.1 Module-Wide Variables and Auxiliary Functions

- `ALL_BASES`: A table of four indexed string fields, which represent the nucleotide single-letter abbreviations.

- `PGFKEYS_PATH`: A string that contains the `pgfkeys` path for `chromatogram` keys.

```
4.13 local ALL_BASES = {"A", "C", "G", "T"}
4.14 local PGFKEYS_PATH = "/pgfmolbio/chromatogram/"
4.15
```

These local functions point to functions in `pgfmolbio.lua` (section 5.2).

```
4.16 local stringToDim = pgfmolbio.stringToDim
4.17 local dimToString = pgfmolbio.dimToString
4.18 local packageError = pgfmolbio.packageError
4.19 local packageWarning = pgfmolbio.packageWarning
4.20 local getRange = pgfmolbio.getRange
4.21
```

`stdProbStyle` is the default `probability style function`. It returns a string representing an optional argument of `\draw`. Depending on the value of `prob`, the probability rule thus drawn is colored black, red, yellow or green for quality scores < 10, < 20, < 30 or ≥ 30, respectively (see also section 2.9).

```
4.22 local function stdProbStyle(prob)
4.23   local color = ""
4.24   if prob >= 0 and prob < 10 then
4.25     color = "black"
4.26   elseif prob >= 10 and prob < 20 then
4.27     color = "pmbTraceRed"
4.28   elseif prob >= 20 and prob < 30 then
4.29     color = "pmbTraceYellow"
4.30   else
4.31     color = "pmbTraceGreen"
```

```
4.32      end
4.33      return "ultra thick, " .. color
4.34  end
4.35
```

`findBasesInStr` searches for nucleotide single-letter abbreviations in its string argument. It returns a table of zero to four indexed string fields (one field per character found, which contains that letter).

```
4.36  local function findBasesInStr(target)
4.37      if not target then return end
4.38      local result = {}
4.39      for _, v in ipairs(ALL_BASES) do
4.40          if target:upper():find(v) then
4.41              table.insert(result, v)
4.42          end
4.43      end
4.44      return result
4.45  end
4.46
```

`readInt` reads `n` bytes from a `file`, starting at `offset` or at the current position if `offset` is `nil`. By assuming big-endian byte order, the byte sequence is converted to a number and returned.

```
4.47  local function readInt(file, n, offset)
4.48      if offset then file:seek("set", offset) end
4.49      local result = 0
4.50      for i = 1, n do
4.51          result = result * 0x100 + file:read(1):byte()
4.52      end
4.53      return result
4.54  end
4.55
```

5.4.2 The `Chromatogram` Class

The `Chromatogram` class (table) represents a single `scf` chromatogram. The constructor `Chromatogram:new` returns a new instance and initializes its variables, which store the values of **chromatogram** keys. Most variables are self-explanatory, since their name is similar to their corresponding key.

```
4.56  Chromatogram = {}
4.57
4.58  function Chromatogram:new()
4.59      newChromatogram = {
4.60          sampleMin = 1,
```

```
4.61        sampleMax = 500,
4.62        sampleStep = 1,
4.63        peakMin = -1,
4.64        peakMax = -1,
4.65        xUnit = stringToDim("0.2mm"),
4.66        yUnit = stringToDim("0.01mm"),
4.67        samplesPerLine = 500,
4.68        baselineSkip = stringToDim("3cm"),
4.69        canvasHeight = stringToDim("2cm"),
4.70        traceStyle = {
4.71          A = PGFKEYS_PATH .. "trace A style",
4.72          C = PGFKEYS_PATH .. "trace C style",
4.73          G = PGFKEYS_PATH .. "trace G style",
4.74          T = PGFKEYS_PATH .. "trace T style"
4.75        },
4.76        tickStyle = {
4.77          A = PGFKEYS_PATH .. "tick A style",
4.78          C = PGFKEYS_PATH .. "tick C style",
4.79          G = PGFKEYS_PATH .. "tick G style",
4.80          T = PGFKEYS_PATH .. "tick T style"
4.81        },
4.82        tickLength = stringToDim("1mm"),
4.83        baseLabelText = {
4.84          A = "\\pgfkeysvalueof{" .. PGFKEYS_PATH .. "base label A text}",
4.85          C = "\\pgfkeysvalueof{" .. PGFKEYS_PATH .. "base label C text}",
4.86          G = "\\pgfkeysvalueof{" .. PGFKEYS_PATH .. "base label G text}",
4.87          T = "\\pgfkeysvalueof{" .. PGFKEYS_PATH .. "base label T text}"
4.88        },
4.89        baseLabelStyle = {
4.90          A = PGFKEYS_PATH .. "base label A style",
4.91          C = PGFKEYS_PATH .. "base label C style",
4.92          G = PGFKEYS_PATH .. "base label G style",
4.93          T = PGFKEYS_PATH .. "base label T style"
4.94        },
4.95        showBaseNumbers = true,
4.96        baseNumberMin = -1,
4.97        baseNumberMax = -1,
4.98        baseNumberStep = 10,
4.99        probDistance = stringToDim("0.8cm"),
4.100       probStyle = stdProbStyle,
4.101       tracesDrawn = ALL_BASES,
4.102       ticksDrawn = "ACGT",
4.103       baseLabelsDrawn = "ACGT",
4.104       probabilitiesDrawn = "ACGT",
4.105     }
4.106     setmetatable(newChromatogram, self)
4.107     self.__index = self
4.108     return newChromatogram
4.109 end
```

`getMinMaxProbability` returns the minimum and maximum probability value in the current chromatogram.

```lua
function Chromatogram:getMinMaxProbability()
  local minProb = 0
  local maxProb = 0
  for _, currPeak in ipairs(self.selectedPeaks) do
    for __, currProb in pairs(currPeak.prob) do
      if currProb > maxProb then maxProb = currProb end
      if currProb < minProb then minProb = currProb end
    end
  end
  return minProb, maxProb
end
```

`getSampleAndPeakIndex` returns the sample (`sampleId`) and peak index (`peakId`) that correspond to `baseIndex`. If `baseIndex` is a number, the function simply returns it as sample index. However, if `baseIndex` is a string of the form `"base` ⟨*number*⟩`"` (as in a valid value for the `sample range` key), the function returns the offset of the ⟨*number*⟩-th peak. `isLowerLimit` must be `true` if the function should return the indices of the lower end of a range.

```lua
function Chromatogram:getSampleAndPeakIndex(baseIndex, isLowerLimit)
  local sampleId, peakId

  sampleId = tonumber(baseIndex)
  if sampleId then
    for i, v in ipairs(self.peaks) do
      if isLowerLimit then
        if v.offset >= sampleId then
          peakId = i
          break
        end
      else
        if v.offset == sampleId then
          peakId = i
          break
        elseif v.offset > sampleId then
          peakId = i - 1
          break
        end
      end
    end
  else
    peakId = tonumber(baseIndex:match("base%s*(%d+)"))
```

```
4.146        if peakId then
4.147          sampleId = self.peaks[peakId].offset
4.148        end
4.149      end
4.150      return sampleId, peakId
4.151  end
4.152
```

5.4.3 Read the `scf` File

`Chromatogram:readScfFile` introduces three further fields to `Chromatogram`:

- `header`: A table of 14 named number fields that save the information in the `scf` header.

- `samples`: A table of four named subtables `A`, `C`, `G`, `T`. Each subtable contains `header.samplesNumber` indexed number fields that represent the fluorescence intensities along a trace.

- `peaks`: A table of `header.basesNumber` indexed subtables which in turn contain three named fields:

    - `offset`: A number indicating the offset of the current peak.
    - `prob`: A table of four named number fields `A`, `C`, `G`, `T`. These numbers store the probability that the current peak is one of the four bases.
    - `base`: A string that states the base represented by the current peak.

`Chromatogram:readScfFile` checks whether the requested `scf` file "`filename`" corresponds to the most recently opened one (via `lastScfFile`). In this case, the variables `peaks` and `samples` already contain the relevant data, so we can refrain from re-reading the file. Otherwise, the program tries to open and evaluate the specified file, raising an error on failure.

```
4.153  function Chromatogram:readScfFile(filename)
4.154    if filename ~= self.lastScfFile then
4.155      self.lastScfFile = filename
4.156      local scfFile, errorMsg = io.open(filename, "rb")
4.157      if not scfFile then packageError(errorMsg) end
4.158
4.159      self.samples = {A = {}, C = {}, G = {}, T = {}}
4.160      self.peaks = {}
```

The function collects the relevant data from the file. *Firstly*, `header` saves the information in the file header:

- `magicNumber`: Each `scf` file must start with the four bytes 2E736366, which is the string ".scf". If this sequence is absent, the **chromatogram** module raises an error.

- `samplesNumber`: The number of sample points.

- `samplesOffset`: The offset of the sample data start.

- `basesNumber`: The number of recognized bases.

- `version`: Since the chromatogram module currently only supports scf version 3.00 (the string "3.00" equals 332E3030), TeX stops with an error message if the file version is different.

- `sampleSize`: The size of each sample point in bytes.

```
4.161   self.header = {
4.162     magicNumber = readInt(scfFile, 4, 0),
4.163     samplesNumber = readInt(scfFile, 4),
4.164     samplesOffset = readInt(scfFile, 4),
4.165     basesNumber = readInt(scfFile, 4),
4.166     leftClip = readInt(scfFile, 4),
4.167     rightClip = readInt(scfFile, 4),
4.168     basesOffset = readInt(scfFile, 4),
4.169     comments = readInt(scfFile, 4),
4.170     commentsOffset = readInt(scfFile, 4),
4.171     version = readInt(scfFile, 4),
4.172     sampleSize = readInt(scfFile, 4),
4.173     codeSet = readInt(scfFile, 4),
4.174     privateSize = readInt(scfFile, 4),
4.175     privateOffset = readInt(scfFile, 4)
4.176   }
4.177   if self.header.magicNumber ~= 0x2E736366 then
4.178     packageError(
4.179       "Magic number in scf scfFile '" ..
4.180       self.lastScfFile ..
4.181       "' corrupt!"
4.182     )
4.183   end
4.184   if self.header.version ~= 0x332E3030 then
4.185     packageError(
4.186       "Scf scfFile '" ..
4.187       self.lastScfFile ..
4.188       "' is not version 3.00!"
4.189     )
4.190   end
```

Secondly, `samples` receives the samples data from the file. Note that the values of the sample points are stored as unsigned integers representing second derivatives (i. e., differences between differences between two consecutive sample points). Hence, we convert them back to signed, absolute values.

64

```
4.191    scfFile:seek("set", self.header.samplesOffset)
4.192    for baseIndex, baseName in ipairs(ALL_BASES) do
4.193      for i = 1, self.header.samplesNumber do
4.194        self.samples[baseName][i] =
4.195          readInt(scfFile, self.header.sampleSize)
4.196      end
4.197
4.198      for _ = 1, 2 do
4.199        local preValue = 0
4.200        for i = 1, self.header.samplesNumber do
4.201          self.samples[baseName][i] = self.samples[baseName][i] + preValue
4.202          if self.samples[baseName][i] > 0xFFFF then
4.203            self.samples[baseName][i] = self.samples[baseName][i] - 0x10000
4.204          end
4.205          preValue = self.samples[baseName][i]
4.206        end
4.207      end
4.208    end
```

Finally, we store the peak information in `peaks`.

```
4.209    for i = 1, self.header.basesNumber do
4.210      self.peaks[i] = {
4.211        offset = readInt(scfFile, 4),
4.212        prob = {A, C, G, T},
4.213        base
4.214      }
4.215    end
4.216
4.217    for i = 1, self.header.basesNumber do
4.218      self.peaks[i].prob.A = readInt(scfFile, 1)
4.219    end
4.220
4.221    for i = 1, self.header.basesNumber do
4.222      self.peaks[i].prob.C = readInt(scfFile, 1)
4.223    end
4.224
4.225    for i = 1, self.header.basesNumber do
4.226      self.peaks[i].prob.G = readInt(scfFile, 1)
4.227    end
4.228
4.229    for i = 1, self.header.basesNumber do
4.230      self.peaks[i].prob.T = readInt(scfFile, 1)
4.231    end
4.232
4.233    for i = 1, self.header.basesNumber do
4.234      self.peaks[i].base = string.char(readInt(scfFile, 1))
4.235    end
4.236
```

```
4.237      scfFile:close()
4.238    end
4.239 end
4.240
```

5.4.4 Set Chromatogram Parameters

`Chromatogram:setParameters` passes options from the **chromatogram** module to the
Lua script. Each field of the table `keyHash` is named after a `Chromatogram` attribute
and represents a function that receives one string parameter (the value of a LaTeX
key). For instance, `keyHash.sampleRange` extracts the range and step values from
the value stored in the `sample range` key.

```
4.241 function Chromatogram:setParameters(newParms)
4.242   local keyHash = {
4.243     sampleRange = function(v)
4.244       local sampleRangeMin, sampleRangeMax, sampleRangeStep =
4.245         getRange(
4.246           v:trim(),
4.247           "^([base]*%s*%d+)%s*%-",
4.248           "%-%s*([base]*%s*%d+)",
4.249           "step%s*(%d+)$"
4.250         )
4.251       self.sampleMin, self.peakMin =
4.252         self:getSampleAndPeakIndex(sampleRangeMin, true)
4.253       self.sampleMax, self.peakMax =
4.254         self:getSampleAndPeakIndex(sampleRangeMax, false)
4.255       if self.sampleMin >= self.sampleMax then
4.256         packageError("Sample range is smaller than 1.")
4.257       end
4.258       self.sampleStep = sampleRangeStep or self.sampleStep
4.259     end,
4.260     xUnit = stringToDim,
4.261     yUnit = stringToDim,
4.262     samplesPerLine = tonumber,
4.263     baselineSkip = stringToDim,
4.264     canvasHeight = stringToDim,
4.265     tickLength = stringToDim,
4.266     showBaseNumbers = function(v)
4.267       if v == "true" then return true else return false end
4.268     end,
4.269     baseNumberRange = function(v)
4.270       local baseNumberRangeMin, baseNumberRangeMax, baseNumberRangeStep =
4.271         getRange(
4.272           v:trim(),
4.273           "^([auto%d]*)%s+%-",
4.274           "%-%s+([auto%d]*$)"
4.275         )
```

```
4.276        if tonumber(baseNumberRangeMin) then
4.277            self.baseNumberMin = tonumber(baseNumberRangeMin)
4.278        else
4.279            self.baseNumberMin = self.peakMin
4.280        end
4.281        if tonumber(baseNumberRangeMax) then
4.282            self.baseNumberMax = tonumber(baseNumberRangeMax)
4.283        else
4.284            self.baseNumberMax = self.peakMax
4.285        end
4.286        if self.baseNumberMin >= self.baseNumberMax then
4.287            packageError("Base number range is smaller than 1.")
4.288        end
4.289        if self.baseNumberMin < self.peakMin then
4.290            self.baseNumberMin = self.peakMin
4.291            packageWarning("Lower base number range is smaller than lower
        sample range. It was adjusted to " .. self.baseNumberMin .. ".")
4.292        end
4.293        if self.baseNumberMax > self.peakMax then
4.294            self.baseNumberMax = self.peakMax
4.295            packageWarning("Upper base number range exceeds upper sample range.
         It was adjusted to " .. self.baseNumberMax .. ".")
4.296        end
4.297        self.baseNumberStep = tonumber(baseNumberRangeStep)
4.298            or self.baseNumberStep
4.299    end,
4.300    probDistance = stringToDim,
4.301    probStyle = function(v) return v end,
4.302    tracesDrawn = findBasesInStr,
4.303    ticksDrawn = function(v) return v end,
4.304    baseLabelsDrawn = function(v) return v end,
4.305    probabilitiesDrawn = function(v) return v end,
4.306    probStyle = function(v) return v end
4.307 }
```

We iterate over all fields in the argument of setParameters. If a field of the same
name exists in keyHash, we call this field with the value of the corresponding field
in newParms as parameter.

```
4.308 for key, value in pairs(newParms) do
4.309    if keyHash[key] then
4.310        self[key] = keyHash[key](value)
4.311    end
4.312 end
4.313 end
4.314
```

5.4.5 Print the Chromatogram

`Chromatogram:printTikzChromatogram` writes all commands that draw the chromatogram to the TeX input stream (via `tex.sprint`), but only if no error has occurred previously.

```
4.315  function Chromatogram:printTikzChromatogram()
4.316    if pgfmolbio.errorCatched then return end
```

(1) Select peaks to draw In order to simplify the drawing operations, we select the peaks that appear in the final output and store information on them in `selectedPeaks`. `selectedPeaks` is a table of zero to `header.basesNumber` indexed subtables. It is similar to `peaks` but only describes the peaks in the displayed part of the chromatogram, which is selected by the `samples range` key. Each subtable of `selectedPeaks` consists of the following five named fields:

- `offset`: A number indicating the offset of the current peak in "transformed" coordinates (i. e., the x-coordinate of the first sample point shown equals 1).

- `base`: See `peaks.base` (section 5.4.3).

- `prob`: See `peaks.prob` (section 5.4.3).

- `baseIndex`: A number that stores the index of the current peak. The first detected peak in the chromatogram has index 1.

- `probXRight`: A number corresponding to the right x-coordinate of the probability indicator.

```
4.317    self.selectedPeaks = {}
4.318    local tIndex = 1
4.319    for rPeakIndex, currPeak in ipairs(self.peaks) do
4.320      if currPeak.offset >= self.sampleMin
4.321          and currPeak.offset <= self.sampleMax then
4.322        self.selectedPeaks[tIndex] = {
4.323          offset = currPeak.offset + 1 - self.sampleMin,
4.324          base = currPeak.base,
4.325          prob = currPeak.prob,
4.326          baseIndex = rPeakIndex,
4.327          probXRight = self.sampleMax + 1 - self.sampleMin
4.328        }
```

The right x-coordinate of the probability indicator (`probXRight`) is the mean between the offsets of the adjacent peaks. For the last peak, `probXRight` equals the largest transformed x-coordinate.

```
4.329        if tIndex > 1 then
4.330          self.selectedPeaks[tIndex-1].probXRight =
4.331            (self.selectedPeaks[tIndex-1].offset
4.332             + self.selectedPeaks[tIndex].offset) / 2
4.333        end
4.334        tIndex = tIndex + 1
4.335      end
4.336    end
4.337
```

Furthermore, we adjust `baseNumberMin` and `baseNumberMax` if any peak was detected in the displayed part of the chromatogram. The value `-1`, which indicates the keyword `auto`, is replaced by the index of the first or last peak, respectively.

```
4.338    if tIndex > 1 then
4.339      if self.baseNumberMin == -1 then
4.340        self.baseNumberMin = self.selectedPeaks[1].baseIndex
4.341      end
4.342      if self.baseNumberMax == -1 then
4.343        self.baseNumberMax = self.selectedPeaks[tIndex-1].baseIndex
4.344      end
4.345    end
4.346
```

(2) Canvas For each line, we draw a rectangle in `canvas style` whose left border coincides with the y-axis.
`yLower`, `yUpper`, `xRight`: rectangle coordinates;
`currLine`: current line, starting from 0;
`samplesLeft`: sample points left to draw after the end of the current line.

```
4.347    local samplesLeft = self.sampleMax - self.sampleMin + 1
4.348    local currLine = 0
4.349    while samplesLeft > 0 do
4.350      local yLower = -currLine * self.baselineSkip
4.351      local yUpper = -currLine * self.baselineSkip + self.canvasHeight
4.352      local xRight =
4.353        (math.min(self.samplesPerLine, samplesLeft) - 1) * self.xUnit
4.354      tex.sprint(
4.355        "\n\t\\draw [" .. PGFKEYS_PATH .. "canvas style] (" ..
4.356        dimToString(0) ..
4.357        ", " ..
4.358        dimToString(yLower) ..
4.359        ") rectangle (" ..
4.360        dimToString(xRight) ..
4.361        ", " ..
4.362        dimToString(yUpper) ..
```

```
4.363          ");"
4.364        )
4.365        samplesLeft = samplesLeft - self.samplesPerLine
4.366        currLine = currLine + 1
4.367      end
4.368
```

(3) Traces   The traces in `tracesDrawn` are drawn sequentially.
`currSampleIndex`: original x-coordinate of a sample point;
`sampleX`: transformed x-coordinate of a sample point, starting at 1;
`x` and `y`: "real" coordinates (in scaled points) of a sample point;
`currLine`: current line, starting at 0;
`firstPointInLine`: boolean that indicates if the current sample point is the first in the line.

```
4.369    for _, baseName in ipairs(self.tracesDrawn) do
4.370      tex.sprint("\n\t\\draw [" .. self.traceStyle[baseName] .. "] ")
4.371      local currSampleIndex = self.sampleMin
4.372      local sampleX = 1
4.373      local x = 0
4.374      local y = 0
4.375      local currLine = 0
4.376      local firstPointInLine = true
4.377
```

We iterate over each sample point. As long as the current sample point is within the selected range, we calculate the real coordinates of the sample point; add the lineto operator `--` if at least one sample point has already appeared in the current line; and write the point to the TeX input stream.

```
4.378      while currSampleIndex <= self.sampleMax do
4.379        x = ((sampleX - 1) % self.samplesPerLine) * self.xUnit
4.380        y = self.samples[baseName][currSampleIndex] * self.yUnit
4.381          - currLine * self.baselineSkip
4.382        if sampleX % self.sampleStep == 0 then
4.383          if not firstPointInLine then
4.384            tex.sprint(" -- ")
4.385          else
4.386            firstPointInLine = false
4.387          end
4.388          tex.sprint(
4.389            "(" ..
4.390            dimToString(x) ..
4.391            ", " ..
4.392            dimToString(y) ..
4.393            ")"
4.394          )
```

```
4.395          end
```

Besides, we add line breaks at the appropriate positions.

```
4.396          if sampleX ~= self.sampleMax + 1 - self.sampleMin then
4.397            if sampleX >= (currLine + 1) * self.samplesPerLine then
4.398              currLine = currLine + 1
4.399              tex.sprint(";\n\t\\draw [" .. self.traceStyle[baseName] .. "] ")
4.400              firstPointInLine = true
4.401            end
4.402          else
4.403            tex.sprint(";")
4.404          end
4.405        sampleX = sampleX + 1
4.406        currSampleIndex = currSampleIndex + 1
4.407        end
4.408      end
4.409
```

(4) Annotations   We iterate over each selected peak and start by finding the line in which the first peak resides.

`currLine`: current line, starting at 0;

`lastProbX`: right x-coordinate of the probability rule of the last peak;

`probRemainder`: string that draws the remainder of a probability indicator following a line break;

`x, yUpper, yLower`: "real" tick coordinates;

`tickOperation`: string that equals either TikZ's moveto or lineto operation, depending on whether the current peak should be marked with a tick.

```
4.410    local currLine = 0
4.411    local lastProbX = 1
4.412    local probRemainder = false
4.413
4.414    for _, currPeak in ipairs(self.selectedPeaks) do
4.415      while currPeak.offset > (currLine + 1) * self.samplesPerLine do
4.416        currLine = currLine + 1
4.417      end
4.418
4.419      local x = ((currPeak.offset - 1) % self.samplesPerLine) * self.xUnit
4.420      local yUpper = -currLine * self.baselineSkip
4.421      local yLower = -currLine * self.baselineSkip - self.tickLength
4.422      local tickOperation = ""
4.423      if self.ticksDrawn:upper():find(currPeak.base) then
4.424        tickOperation = "--"
4.425      end
4.426
```

(4a) Ticks and labels Having calculated all coordinates, we draw the tick and the base label, given the latter has been specified by `base labels drawn`.

```lua
4.427    tex.sprint(
4.428      "\n\t\\draw [" ..
4.429      self.tickStyle[currPeak.base] ..
4.430      "] (" ..
4.431      dimToString(x) ..
4.432      ", " ..
4.433      dimToString(yUpper) ..
4.434      ") " ..
4.435      tickOperation ..
4.436      " (" ..
4.437      dimToString(x) ..
4.438      ", " ..
4.439      dimToString(yLower) ..
4.440      ")"
4.441    )
4.442    if self.baseLabelsDrawn:upper():find(currPeak.base) then
4.443      tex.sprint(
4.444        " node [" ..
4.445        self.baseLabelStyle[currPeak.base] ..
4.446        "] {" ..
4.447        self.baseLabelText[currPeak.base] ..
4.448        "}"
4.449      )
4.450    end
4.451
```

(4b) Base numbers If `show base numbers` is true and the current base number is within the interval given by `base number range`, a base number is printed.

```lua
4.452    if self.showBaseNumbers
4.453        and currPeak.baseIndex >= self.baseNumberMin
4.454        and currPeak.baseIndex <= self.baseNumberMax
4.455        and (currPeak.baseIndex - self.baseNumberMin)
4.456          % self.baseNumberStep == 0 then
4.457      tex.sprint(
4.458        " node [" ..
4.459        PGFKEYS_PATH ..
4.460        "base number style] {\\strut " ..
4.461        currPeak.baseIndex ..
4.462        "}"
4.463      )
4.464    end
4.465    tex.sprint(";")
4.466
```

72

(4c) Probabilities   First, we draw the remainder of the last probability rule. Such a remainder has been stored in `probRemainder` if the last rule had protruded into the right margin (see below). Furthermore, we determine if a probability rule should appear beneath the current peak.

```lua
4.467   if probRemainder then
4.468     tex.sprint(probRemainder)
4.469     probRemainder = false
4.470   end
4.471   local drawCurrProb =
4.472     self.probabilitiesDrawn:upper():find(currPeak.base)
```

Now comes the tricky part. Whenever we choose to paint a probability rule, we may envision three scenarios. *Firstly*, the probability rule starts in the left margin of the current line (i. e., `xLeft` is negative). This means that the part protruding into the left margin must instead appear at the end of the last line. Therefore, we calculate the coordinates of this part (storing them in `xLeftPrev`, `xRightPrev` and `yPrev`) and draw the segment. Since the remainder of the rule necessarily starts at the left border of the current line, we set `xLeft` to zero.

```lua
4.473   local xLeft = lastProbX - 1 - currLine * self.samplesPerLine
4.474   if xLeft < 0 then
4.475     local xLeftPrev = (self.samplesPerLine + xLeft) * self.xUnit
4.476     local xRightPrev = (self.samplesPerLine - 1) * self.xUnit
4.477     local yPrev = -(currLine-1) * self.baselineSkip - self.probDistance
4.478     if drawCurrProb then
4.479       tex.sprint(
4.480         "\n\t\\draw [" ..
4.481         self.probStyle(currPeak.prob[currPeak.base]) ..
4.482         "] (" ..
4.483         dimToString(xLeftPrev) ..
4.484         ", " ..
4.485         dimToString(yPrev) ..
4.486         ") -- (" ..
4.487         dimToString(xRightPrev) ..
4.488         ", " ..
4.489         dimToString(yPrev) ..
4.490         ");"
4.491       )
4.492     end
4.493     xLeft = 0
4.494   else
4.495     xLeft = xLeft * self.xUnit
4.496   end
4.497
```

Secondly, the probability rule ends in the right margin of the current line (i. e., `xRight` at least equals `samplesPerLine`). This means that the part protruding into

the right margin must instead appear at the start of the following line. Therefore, we calculate the coordinates of this part (storing them in `xRightNext` and `yNext`) and save the drawing command in `probRemainder` (whose contents were printed above). Since the remainder of the rule necessarily ends at the right border of the current line, we set `xRight` to this coordinate.

```
4.498        local xRight = currPeak.probXRight - 1 - currLine * self.samplesPerLine
4.499        if xRight >= self.samplesPerLine then
4.500          if drawCurrProb then
4.501            local xRightNext = (xRight - self.samplesPerLine) * self.xUnit
4.502            local yNext = -(currLine+1) * self.baselineSkip - self.probDistance
4.503            probRemainder =
4.504              "\n\t\\draw [" ..
4.505              self.probStyle(currPeak.prob[currPeak.base]) ..
4.506              "] (" ..
4.507              dimToString(0) ..
4.508              ", " ..
4.509              dimToString(yNext) ..
4.510              ") -- (" ..
4.511              dimToString(xRightNext) ..
4.512              ", " ..
4.513              dimToString(yNext) ..
4.514              ");"
4.515          end
4.516          xRight = (self.samplesPerLine - 1) * self.xUnit
4.517        else
4.518          xRight = xRight * self.xUnit
4.519        end
4.520
```

Thirdly, the probability rule starts and ends within the boundaries of the current line. In this lucky case, the *y*-coordinate is the only one missing, since we previously calculated `xLeft` (case 1) and `xRight` (case 2). Drawing of the probability rule proceeds as usual.

```
4.521        local y = -currLine * self.baselineSkip - self.probDistance
4.522        if drawCurrProb then
4.523          tex.sprint(
4.524            "\n\t\\draw [" ..
4.525            self.probStyle(currPeak.prob[currPeak.base]) ..
4.526            "] (" ..
4.527            dimToString(xLeft) ..
4.528            ", " ..
4.529            dimToString(y) ..
4.530            ") -- (" ..
4.531            dimToString(xRight) ..
4.532            ", " ..
4.533            dimToString(y) ..
4.534            ");"
```

```
4.535           )
4.536         end
4.537       lastProbX = currPeak.probXRight
4.538     end
4.539 end
```

5.5 `pgfmolbio.domains.tex`

```
5.1 \ProvidesFile{pgfmolbio.domains.tex}[2012/10/01 v0.2 Protein Domains]
5.2
```

If the domains module is requested by LuaTEX it loads the corresponding Lua module and generates a new `SpecialKeys` object, which will store all feature styles, disulfide keys and print functions (section 5.6.2).

```
5.3 \ifluatex
5.4   \RequireLuaModule{pgfmolbio.domains}
5.5   \directlua{pmbSpecialKeys = pgfmolbio.domains.SpecialKeys:new()}
5.6 \fi
5.7
```

5.5.1 Keys

`\@pmb@dom@keydef`

#1: ⟨*key*⟩ name
#2: default ⟨*value*⟩
`\@pmb@dom@keydef` declares a ⟨*key*⟩ in path `/pgfmolbio/domains` and assigns a default ⟨*value*⟩.

```
5.8 \def\@pmb@dom@keydef#1#2{%
5.9   \pgfkeyssetvalue{/pgfmolbio/domains/#1}{#2}%
5.10 }
5.11
```

`\pmbdomvalueof`

#1: ⟨*key*⟩ name
`\pmbdomvalueof` retrieves the value of a ⟨*key*⟩ in path `/pgfmolbio/domains`. Note that the control word lacks an `@` and is thus freely accessible within a LATEX document (see section 3.4).

```
5.12  \def\pmbdomvalueof#1{%
5.13    \pgfkeysvalueof{/pgfmolbio/domains/#1}%
5.14  }
5.15
```

Aided by these auxiliary macros, we define all keys of the domains module.

```
5.16  \@pmb@dom@keydef{name}{Protein}
5.17  \newif\ifpmb@dom@showname
5.18  \pgfmolbioset[domains]{%
5.19    show name/.is if=pmb@dom@showname,
5.20    show name
5.21  }
5.22  \@pmb@dom@keydef{description}{}
5.23
5.24  \@pmb@dom@keydef{x unit}{.5mm}
5.25  \@pmb@dom@keydef{y unit}{6mm}
5.26  \@pmb@dom@keydef{residues per line}{200}
5.27  \@pmb@dom@keydef{baseline skip}{3}
5.28  \@pmb@dom@keydef{residue numbering}{auto}
5.29  \@pmb@dom@keydef{residue range}{auto-auto}
5.30  \@pmb@dom@keydef{enlarge left}{0cm}
5.31  \@pmb@dom@keydef{enlarge right}{0cm}
5.32  \@pmb@dom@keydef{enlarge top}{1cm}
5.33  \@pmb@dom@keydef{enlarge bottom}{0cm}
5.34
5.35  \pgfmolbioset[domains]{%
5.36    style/.code=\pgfmolbioset[domains]{current style/.style={#1}}
5.37  }
5.38
5.39  \@pmb@dom@keydef{domain font}{\footnotesize}
5.40
5.41  \@pmb@dom@keydef{level}{}
5.42  \@pmb@dom@keydef{disulfide base distance}{1}
5.43  \@pmb@dom@keydef{disulfide level distance}{.2}
5.44  \@pmb@dom@keydef{range font}{\sffamily\scriptsize}
5.45
5.46  \newif\ifpmb@dom@showruler
5.47  \pgfmolbioset[domains]{%
5.48    show ruler/.is if=pmb@dom@showruler,
5.49    show ruler
5.50  }
5.51  \@pmb@dom@keydef{ruler range}{auto-auto}
5.52  \@pmb@dom@keydef{default ruler step size}{50}
5.53  \@pmb@dom@keydef{ruler distance}{-.5}
5.54
5.55  \@pmb@dom@keydef{sequence}{}
5.56  \@pmb@dom@keydef{magnified sequence font}{\ttfamily\footnotesize}
5.57
```

```
5.58
5.59  \newif\ifpmb@dom@showsecstructure
5.60  \pgfmolbioset[domains]{%
5.61    show secondary structure/.is if=pmb@dom@showsecstructure,
5.62    show secondary structure=false
5.63  }
5.64  \@pmb@dom@keydef{secondary structure distance}{1}
5.65  \pgfmolbioset[domains]{%
5.66    helix back border color/.code=\colorlet{helix back border color}{#1},
5.67    helix back main color/.code=\colorlet{helix back main color}{#1},
5.68    helix back middle color/.code=\colorlet{helix back middle color}{#1},
5.69    helix front border color/.code=\colorlet{helix front border color}{#1},
5.70    helix front main color/.code=\colorlet{helix front main color}{#1},
5.71    helix front middle color/.code=\colorlet{helix front middle color}{#1},
5.72    helix back border color=white!50!black,
5.73    helix back main color=white!90!black,
5.74    helix back middle color=white,
5.75    helix front border color=red!50!black,
5.76    helix front main color=red!90!black,
5.77    helix front middle color=red!10!white
5.78  }
5.79
5.80  \@pmb@dom@keydef{sequence length}{}
5.81
5.82  \@pmb@dom@keydef{@layer}{}
5.83
```

5.5.2 Feature Shapes

\setfeatureshape

#1: Shape ⟨*name*⟩.
#2: Ti*k*Z ⟨*code*⟩.
Stores the ⟨*code*⟩ for a shape in the macro **\@pmb@dom@feature@**⟨*name*⟩**@shape**.

```
5.84  \newcommand\setfeatureshape[2]{%
5.85    \expandafter\def\csname @pmb@dom@feature@#1@shape\endcsname{#2}%
5.86  }
5.87
```

\setfeatureshapealias

#1: New shape.
#2: Existing shape.
Links a new shape to an existing one.

```
5.88  \newcommand\setfeatureshapealias[2]{%
5.89    \expandafter\def\csname @pmb@dom@feature@#1@shape\endcsname{%
5.90      \@nameuse{@pmb@dom@feature@#2@shape}%
5.91    }%
5.92  }
5.93
```

\setfeaturestylealias

#1: New style.

#2: Existing style.

This macro and the next one are only defined in LuaTeXepending on whether \setfeaturestylealias occurs within a pmbdomains environment, it either sets the feature styles of the SpecialKeys object in the current Protein (pmbProtein. specialKeys) or of the global SpecialKeys object (pmbSpecialKeys).

\setfeaturealias

#1: New feature.

#2: Existing feature.

Calls \setfeatureshapealias and possibly \setfeaturestylealias.

```
5.94   \ifluatex
5.95     \newcommand\setfeaturestylealias[2]{%
5.96       \directlua{
5.97         if pmbProtein then
5.98           pmbProtein.specialKeys:aliasFeatureStyle("#1", "#2")
5.99         else
5.100          pmbSpecialKeys:aliasFeatureStyle("#1", "#2")
5.101        end
5.102      }%
5.103    }
5.104    \newcommand\setfeaturealias[2]{%
5.105      \setfeatureshapealias{#1}{#2}%
5.106      \setfeaturestylealias{#1}{#2}%
5.107    }
5.108  \else
5.109    \let\setfeaturealias\setfeatureshapealias%
5.110  \fi
5.111
```

\pmbdomdrawfeature

#1: The feature ⟨*type*⟩ that should be drawn.

If a feature $\langle type \rangle$ (i. e., the corresponding macro) is undefined, we issue a warning and draw feature `default`.

```
5.112  \newcommand\pmbdomdrawfeature[1]{%
5.113    \@ifundefined{@pmb@dom@feature@#1@shape}{%
5.114      \PackageWarning{pgfmolbio}%
5.115        {Feature shape `#1' unknown, using `default'.}%
5.116      \@pmb@dom@feature@default@shape%
5.117    }{%
5.118      \@nameuse{@pmb@dom@feature@#1@shape}%
5.119    }%
5.120  }
5.121
```

Definitions of standard features and their aliases.

```
5.122  \setfeatureshape{default}{%
5.123    \path [/pgfmolbio/domains/current style]
5.124      (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
5.125      (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
5.126  }
5.127
5.128  \setfeatureshape{domain}{
5.129    \draw [/pgfmolbio/domains/current style, rounded corners=2pt]
5.130      (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
5.131      (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
5.132    \node at (\xMid, \yMid)
5.133      {\pmbdomvalueof{domain font}{\pmbdomvalueof{description}}};
5.134  }
5.135  \setfeaturealias{DOMAIN}{domain}
5.136
5.137  \setfeatureshape{signal peptide}{%
5.138    \path [/pgfmolbio/domains/current style]
5.139      (\xLeft, \yMid + \pmbdomvalueof{y unit} / 5) rectangle
5.140      (\xRight, \yMid - \pmbdomvalueof{y unit} / 5);
5.141  }
5.142  \setfeaturealias{SIGNAL}{signal peptide}
5.143
5.144  \setfeatureshape{propeptide}{%
5.145    \path [/pgfmolbio/domains/current style]
5.146      (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
5.147      (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
5.148  }
5.149  \setfeaturealias{PROPEP}{propeptide}
5.150
5.151  \setfeatureshape{carbohydrate}{%
5.152    \draw [/pgfmolbio/domains/current style]
5.153      (\xMid, \yMid) --
5.154      (\xMid, \yMid + .7 * \pmbdomvalueof{y unit})
5.155      node [above] {\tiny\strut\pmbdomvalueof{description}};
```

79

```
5.156    \fill [/pgfmolbio/domains/current style]
5.157       (\xMid, \yMid + .7 * \pmbdomvalueof{y unit}) circle [radius=1pt];
5.158  }
5.159  \setfeaturealias{CARBOHYD}{carbohydrate}
5.160
5.161  \setfeatureshape{other/main chain}{%
5.162    \ifpmb@dom@showsecstructure%
5.163       \pgfmathsetmacro\yUpper{%
5.164          \yMid + \pmbdomvalueof{secondary structure distance}
5.165             * \pmbdomvalueof{y unit}%
5.166       }
5.167       \draw [thin]
5.168          (\xLeft, \yUpper pt) --
5.169          (\xRight, \yUpper pt);%
5.170    \fi%
5.171    \path [/pgfmolbio/domains/current style]
5.172       (\xLeft, \yMid) --
5.173       (\xRight, \yMid);%
5.174  }
5.175
5.176  \setfeatureshape{other/name}{%
5.177    \ifpmb@dom@showname%
5.178       \node [/pgfmolbio/domains/current style]
5.179          at (\xMid, \pmbdomvalueof{baseline skip} * \pmbdomvalueof{y unit})
5.180          {\pmbdomvalueof{name} (\pmbdomvalueof{sequence length} residues)};
5.181    \fi%
5.182  }
5.183
5.184  \setfeatureshape{disulfide}{%
5.185    \pgfmathsetmacro\yUpper{%
5.186       \yMid + (
5.187          \pmbdomvalueof{disulfide base distance} +
5.188          (\pmbdomvalueof{level} - 1) *
5.189          \pmbdomvalueof{disulfide level distance}
5.190       ) * \pmbdomvalueof{y unit}
5.191    }
5.192    \path [/pgfmolbio/domains/current style]
5.193       (\xLeft, \yMid) --
5.194       (\xLeft, \yUpper pt) --
5.195       (\xRight, \yUpper pt) --
5.196       (\xRight, \yMid);
5.197  }
5.198  \setfeaturealias{DISULFID}{disulfide}
5.199
5.200  \setfeatureshape{range}{%
5.201    \pgfmathsetmacro\yUpper{%
5.202       \yMid + (
5.203          \pmbdomvalueof{disulfide base distance} +
5.204          (\pmbdomvalueof{level} - 1) *
```

```
5.205        \pmbdomvalueof{disulfide level distance}
5.206      ) * \pmbdomvalueof{y unit}
5.207    }
5.208    \path [/pgfmolbio/domains/current style]
5.209      (\xLeft, \yUpper pt) --
5.210      (\xRight, \yUpper pt)
5.211      node [pos=.5, above]
5.212        {\pmbdomvalueof{range font}{\pmbdomvalueof{description}}};
5.213 }
5.214
5.215 \setfeatureshape{other/ruler}{%
5.216    \draw [/pgfmolbio/domains/current style]
5.217      (\xMid,
5.218        \yMid + \pmbdomvalueof{ruler distance} *
5.219              \pmbdomvalueof{y unit}) --
5.220      (\xMid,
5.221        \yMid + \pmbdomvalueof{ruler distance} *
5.222              \pmbdomvalueof{y unit} - 1mm)
5.223      node [below=-1mm] {\tiny\sffamily\strut\residueNumber};
5.224 }
5.225
5.226
5.227 \setfeatureshape{other/sequence}{%
5.228    \node [/pgfmolbio/domains/current style]
5.229      at (\xMid, \yMid) {\strut\currentResidue};
5.230 }
5.231
5.232 \newlength\pmb@magnifiedsequence@width
5.233
5.234 \setfeatureshape{other/magnified sequence above}{%
5.235    \settowidth\pmb@magnifiedsequence@width{%
5.236      \begin{pgfinterruptpicture}%
5.237        \pmbdomvalueof{magnified sequence font}%
5.238        \featureSequence%
5.239      \end{pgfinterruptpicture}%
5.240    }%
5.241    \pgfmathsetmacro\xUpperLeft{\xMid - \pmb@magnifiedsequence@width / 2}
5.242    \pgfmathsetmacro\xUpperRight{\xMid + \pmb@magnifiedsequence@width / 2}
5.243
5.244    \draw [/pgfmolbio/domains/current style]
5.245      (\xLeft, \yMid) --
5.246      (\xLeft, \yMid + \pmbdomvalueof{y unit} / 6) --
5.247      (\xUpperLeft pt, \yMid + \pmbdomvalueof{y unit} * 4/6) --
5.248      (\xUpperLeft pt, \yMid + \pmbdomvalueof{y unit} * 5/6)
5.249      (\xUpperRight pt, \yMid + \pmbdomvalueof{y unit} * 5/6) --
5.250      (\xUpperRight pt, \yMid + \pmbdomvalueof{y unit} * 4/6) --
5.251      (\xRight, \yMid + \pmbdomvalueof{y unit} / 6) --
5.252      (\xRight, \yMid);
5.253    \node [anchor=mid]
```

```
5.254        at (\xMid, \yMid + \pmbdomvalueof{y unit})
5.255        {\pmbdomvalueof{magnified sequence font}\featureSequence};
5.256  }
5.257
5.258  \setfeatureshape{other/magnified sequence below}{%
5.259     \settowidth\pmb@magnifiedsequence@width{%
5.260        \begin{pgfinterruptpicture}%
5.261           \pmbdomvalueof{magnified sequence font}%
5.262           \featureSequence%
5.263        \end{pgfinterruptpicture}%
5.264     }%
5.265     \pgfmathsetmacro\xLowerLeft{\xMid - \pmb@magnifiedsequence@width / 2}
5.266     \pgfmathsetmacro\xLowerRight{\xMid + \pmb@magnifiedsequence@width / 2}
5.267
5.268     \draw [/pgfmolbio/domains/current style]
5.269        (\xLeft, \yMid) --
5.270        (\xLeft, \yMid - \pmbdomvalueof{y unit} / 6) --
5.271        (\xLowerLeft pt, \yMid - \pmbdomvalueof{y unit}) --
5.272        (\xLowerLeft pt, \yMid - \pmbdomvalueof{y unit} * 7/6)
5.273        (\xLowerRight pt, \yMid - \pmbdomvalueof{y unit} * 7/6) --
5.274        (\xLowerRight pt, \yMid - \pmbdomvalueof{y unit}) --
5.275        (\xRight, \yMid - \pmbdomvalueof{y unit} / 6) --
5.276        (\xRight, \yMid);
5.277     \node [anchor=mid]
5.278        at (\xMid, \yMid - \pmbdomvalueof{y unit} * 8/6)
5.279        {\pmbdomvalueof{magnified sequence font}\featureSequence};
5.280  }
5.281
5.282
```

5.5.3 Secondary Structure Elements

<div>

`\@pmb@dom@helixsegment`

</div>

#1: Scale factor for Ti*k*Z's `svg` action.

Draws a full helix segment at the current canvas position. We use the (unusual) `svg` syntax since the helix segment was designed in Inkscape, and the `svg` commands were copied from the resulting vector graphics file.

```
5.283  \newcommand\@pmb@dom@helixsegment[1]{%
5.284     svg [scale=#1] "%
5.285        c  0.30427  0
5.286           0.62523  0.59174
5.287           0.79543  0.96646
5.288        c  0.97673  2.15039
5.289           1.34005  4.49858
```

```
5.290          1.84538  6.6178
5.291       c  0.56155  2.35498
5.292          0.99602  4.514
5.293          1.82948  6.72355
5.294       c  0.11069  0.29346
5.295          0.23841  0.69219
5.296          0.56172  0.69219
5.297       l -5        0
5.298       c -0.27235  0.0237
5.299         -0.55793 -0.51373
5.300         -0.65225 -0.76773
5.301       c -0.98048 -2.64055
5.302         -1.40233 -5.46534
5.303         -2.06809 -8.00784
5.304       c -0.50047 -1.91127
5.305         -0.94696 -3.73368
5.306         -1.68631 -5.43929
5.307       c -0.14066 -0.3245
5.308         -0.34516 -0.78514
5.309         -0.69997 -0.78514
5.310       z"
5.311 }
5.312
```

\@pmb@dom@helixhalfsegment

#1: Scale factor for Ti*k*Z's `svg` action.

Draws a half helix segment.

```
5.313 \newcommand\@pmb@dom@helixhalfsegment[1]{%
5.314   svg [scale=#1] "%
5.315     c  0.50663  2.18926
5.316        0.96294  4.51494
5.317        1.78125  6.71875
5.318     c  0.09432  0.254
5.319        0.35265  0.80495
5.320        0.625    0.78125
5.321     l  5        0
5.322     c -0.32331  0
5.323       -0.45181 -0.42529
5.324       -0.5625  -0.71875
5.325     c -0.83346 -2.20955
5.326       -1.2822  -4.36377
5.327       -1.84375 -6.78125
5.328     l -5        0
5.329     z"
5.330 }
5.331
```

Shadings for helix segments.

```
5.332  \pgfdeclareverticalshading[%
5.333      helix back border color,%
5.334      helix back main color,%
5.335      helix back middle color%
5.336  ]{helix half upper back}{100bp}{
5.337  color(0bp)=(helix back middle color);
5.338  color(5bp)=(helix back middle color);
5.339  color(45bp)=(helix back main color);
5.340  color(75bp)=(helix back border color);
5.341  color(100bp)=(helix back border color)
5.342  }
5.343
5.344  \pgfdeclareverticalshading[%
5.345      helix back border color,%
5.346      helix back main color,%
5.347      helix back middle color%
5.348  ]{helix half lower back}{100bp}{
5.349  color(0bp)=(helix back border color);
5.350  color(25bp)=(helix back border color);
5.351  color(35bp)=(helix back main color);
5.352  color(55bp)=(helix back middle color);
5.353  color(95bp)=(helix back main color);
5.354  color(100bp)=(helix back main color)
5.355  }
5.356
5.357  \pgfdeclareverticalshading[%
5.358      helix back border color,%
5.359      helix back main color,%
5.360      helix back middle color%
5.361  ]{helix full back}{100bp}{
5.362  color(0bp)=(helix back border color);
5.363  color(25bp)=(helix back border color);
5.364  color(30bp)=(helix back main color);
5.365  color(40bp)=(helix back middle color);
5.366  color(60bp)=(helix back main color);
5.367  color(75bp)=(helix back border color);
5.368  color(100bp)=(helix back border color)
5.369  }
5.370
5.371  \pgfdeclareverticalshading[%
5.372      helix front border color,%
5.373      helix front main color,%
5.374      helix front middle color%
5.375  ]{helix half upper front}{100bp}{
5.376  color(0bp)=(helix front main color);
5.377  color(5bp)=(helix front main color);
5.378  color(45bp)=(helix front middle color);
```

```
5.379    color(65bp)=(helix front main color);
5.380    color(75bp)=(helix front border color);
5.381    color(100bp)=(helix front border color)
5.382  }
5.383
5.384  \pgfdeclareverticalshading[%
5.385      helix front border color,%
5.386      helix front main color,%
5.387      helix front middle color%
5.388    ]{helix full front}{100bp}{
5.389    color(0bp)=(helix front border color);
5.390    color(25bp)=(helix front border color);
5.391    color(40bp)=(helix front main color);
5.392    color(60bp)=(helix front middle color);
5.393    color(70bp)=(helix front main color);
5.394    color(75bp)=(helix front border color);
5.395    color(100bp)=(helix front border color)
5.396  }
5.397
```

The following features print single helical turns. They are drawn with appropriate coordinates by `printHelixFeature` (section 5.6.1).

```
5.398  \setfeatureshape{helix/half upper back}{%
5.399    \ifpmb@dom@showsecstructure%
5.400      \pgfmathsetmacro\yShift{%
5.401        \pmbdomvalueof{secondary structure distance} *
5.402        \pmbdomvalueof{y unit}%
5.403      }
5.404      \draw [shading=helix half upper back]
5.405        (\xLeft, \yMid + \yShift pt)
5.406        \@pmb@dom@helixhalfsegment{\pmbdomvalueof{x unit} / 5};
5.407    \fi%
5.408  }
5.409
5.410  \setfeatureshape{helix/half lower back}{%
5.411    \ifpmb@dom@showsecstructure%
5.412      \pgfmathsetmacro\yShift{%
5.413        \pmbdomvalueof{secondary structure distance} *
5.414        \pmbdomvalueof{y unit}%
5.415      }
5.416      \draw [shading=helix half lower back]
5.417        (\xRight, \yMid + \yShift pt) [rotate=180]
5.418        \@pmb@dom@helixhalfsegment{\pmbdomvalueof{x unit} / 5};
5.419    \fi%
5.420  }
5.421
5.422  \setfeatureshape{helix/full back}{%
5.423    \ifpmb@dom@showsecstructure%
5.424      \pgfmathsetmacro\yShift{%
```

```
5.425            \pmbdomvalueof{secondary structure distance} *
5.426            \pmbdomvalueof{y unit}%
5.427        }
5.428      \draw [shading=helix full back]
5.429          (\xMid, \yLower + \yShift pt)
5.430          \@pmb@dom@helixsegment{\pmbdomvalueof{x unit} / 5};
5.431    \fi%
5.432 }
5.433
5.434 \setfeatureshape{helix/half upper front}{%
5.435    \ifpmb@dom@showsecstructure%
5.436        \pgfmathsetmacro\yShift{%
5.437            \pmbdomvalueof{secondary structure distance} *
5.438            \pmbdomvalueof{y unit}%
5.439        }
5.440      \draw [shading=helix half upper front]
5.441          (\xRight, \yMid + \yShift pt) [xscale=-1]
5.442          \@pmb@dom@helixhalfsegment{\pmbdomvalueof{x unit} / 5};
5.443    \fi%
5.444 }
5.445
5.446 \setfeatureshape{helix/full front}{%
5.447    \ifpmb@dom@showsecstructure%
5.448        \pgfmathsetmacro\yShift{%
5.449            \pmbdomvalueof{secondary structure distance} *
5.450            \pmbdomvalueof{y unit}%
5.451        }
5.452      \draw [shading=helix full front]
5.453          (\xMid, \yLower + \yShift pt) [xscale=-1]
5.454          \@pmb@dom@helixsegment{\pmbdomvalueof{x unit} / 5};
5.455    \fi%
5.456 }
5.457
```

Definitions of the remaining secondary structure features.

```
5.458 \definecolor{strand left color}{RGB}{42,127,255}
5.459 \definecolor{strand right color}{RGB}{128,179,255}
5.460
5.461 \setfeatureshape{beta strand}{%
5.462    \ifpmb@dom@showsecstructure%
5.463        \pgfmathsetmacro\yShift{%
5.464            \pmbdomvalueof{secondary structure distance} *
5.465            \pmbdomvalueof{y unit}%
5.466        }
5.467      \draw [/pgfmolbio/domains/current style]
5.468          (\xLeft, \yMid + \pmbdomvalueof{x unit} + \yShift pt) --
5.469          (\xRight - 1.5 * \pmbdomvalueof{x unit},
5.470            \yMid + \pmbdomvalueof{x unit} + \yShift pt) --
5.471          (\xRight - 1.5 * \pmbdomvalueof{x unit},
```

```
5.472          \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
5.473        (\xRight, \yMid + \yShift pt) --
5.474        (\xRight - 1.5 * \pmbdomvalueof{x unit},
5.475          \yMid - 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
5.476        (\xRight - 1.5 * \pmbdomvalueof{x unit},
5.477          \yMid - \pmbdomvalueof{x unit} + \yShift pt) --
5.478        (\xLeft, \yMid - \pmbdomvalueof{x unit} + \yShift pt) --
5.479        cycle;%
5.480    \fi%
5.481 }
5.482 \setfeaturealias{STRAND}{beta strand}
5.483
5.484 \setfeatureshape{beta turn}{%
5.485    \ifpmb@dom@showsecstructure%
5.486      \pgfmathsetmacro\yShift{%
5.487        \pmbdomvalueof{secondary structure distance} *
5.488        \pmbdomvalueof{y unit}%
5.489      }
5.490      \pgfmathsetmacro\turnXradius{(\xRight - \xLeft) / 2}%
5.491      \pgfmathsetmacro\turnYradius{\pmbdomvalueof{x unit} * 1.5}%
5.492      \fill [white]
5.493        (\xLeft, \yMid + 1mm + \yShift pt) rectangle
5.494        (\xRight, \yMid - 1mm + \yShift pt);%
5.495      \draw [/pgfmolbio/domains/current style]
5.496        (\xLeft - .5pt, \yMid + \yShift pt) --
5.497        (\xLeft, \yMid + \yShift pt) arc
5.498          [start angle=180, end angle=0,
5.499          x radius=\turnXradius pt, y radius=\turnYradius pt] --
5.500        (\xRight + .5pt, \yMid + \yShift pt);%
5.501    \fi%
5.502 }
5.503 \setfeaturealias{TURN}{beta turn}
5.504
5.505 \setfeatureshape{beta bridge}{%
5.506    \ifpmb@dom@showsecstructure%
5.507      \pgfmathsetmacro\yShift{%
5.508        \pmbdomvalueof{secondary structure distance} *
5.509        \pmbdomvalueof{y unit}%
5.510      }
5.511      \draw [/pgfmolbio/domains/current style]
5.512        (\xLeft, \yMid + .25 * \pmbdomvalueof{x unit} + \yShift pt) --
5.513        (\xRight - 1.5 * \pmbdomvalueof{x unit},
5.514          \yMid + .25 * \pmbdomvalueof{x unit} + \yShift pt) --
5.515        (\xRight - 1.5 * \pmbdomvalueof{x unit},
5.516          \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
5.517        (\xRight, \yMid + \yShift pt) --
5.518        (\xRight - 1.5 * \pmbdomvalueof{x unit},
5.519          \yMid - 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
5.520        (\xRight - 1.5 * \pmbdomvalueof{x unit},
```

```
5.521        \yMid - .25 * \pmbdomvalueof{x unit} + \yShift pt) --
5.522      (\xLeft, \yMid - .25 * \pmbdomvalueof{x unit} + \yShift pt) --
5.523      cycle;%
5.524    \fi%
5.525 }
5.526
5.527 \setfeatureshape{bend}{%
5.528    \ifpmb@dom@showsecstructure%
5.529      \pgfmathsetmacro\yShift{%
5.530        \pmbdomvalueof{secondary structure distance} *
5.531        \pmbdomvalueof{y unit}%
5.532      }
5.533      \fill [white]
5.534        (\xLeft, \yMid + 1mm + \yShift pt) rectangle
5.535        (\xRight, \yMid - 1mm + \yShift pt);%
5.536      \draw [/pgfmolbio/domains/current style]
5.537        (\xLeft - .5pt, \yMid + \yShift pt) --
5.538        (\xLeft, \yMid + \yShift pt) --
5.539        (\xMid, \yMid + .5 * \pmbdomvalueof{y unit} + \yShift pt) --
5.540        (\xRight, \yMid + \yShift pt) --
5.541        (\xRight + .5pt, \yMid + \yShift pt);%
5.542    \fi%
5.543 }
5.544
```

This concludes the part of the package that is always loaded. The remaining code is only executed within LuaTeX.

```
5.545 \ifluatex\else\expandafter\endinput\fi
5.546
5.547
```

5.5.4 Adding Features

\pmb@dom@inputuniprot

#1: The ⟨*name*⟩ of a Uniprot file.
\pmb@dom@inputuniprot reads some attributes and all features from a Uniprot file (readUniprotFile, section 5.6.4). It then updates some keys of the domains module (getParameters, section 5.6.5) and then passes the value of residue numbering to the pmbProtein object.

```
5.548 \newcommand\pmb@dom@inputuniprot[1]{%
5.549    \directlua{
5.550      pmbProtein:readUniprotFile("#1")
5.551      pmbProtein:getParameters()
5.552      pmbProtein:setParameters{
```

88

```
5.553        residueNumbering = "\pmbdomvalueof{residue numbering}"
5.554      }
5.555    }%
5.556 }
5.557
```

#1: The ⟨*name*⟩ of a General Feature Format (`gff`) file.

This macro reads all features from a `gff` file (`readGffFile`, section 5.6.4). It then passes the value of residue numbering to `pmbProtein`.

```
5.558 \newcommand\pmb@dom@inputgff[1]{%
5.559   \directlua{
5.560     pmbProtein:readGffFile("#1")
5.561     pmbProtein:setParameters{
5.562       residueNumbering = "\pmbdomvalueof{residue numbering}"
5.563     }
5.564   }%
5.565 }
5.566
```

#1: A ⟨*key-value list*⟩ that is locally applied to the feature.

#2: The feature ⟨*key*⟩.

#3: The ⟨*first*⟩ …

#4: and ⟨*last*⟩ residue covered by the feature.

This macro adds a feature to `pmbProtein` by calling its `addFeature` method. The ⟨*key-value list*⟩ should be stored without any expansion in the `kvList` field of `addFeature`'s single argument table. To this end, we first store the ⟨*key-value list*⟩ in the token register `\@pmb@toksa` and then access its contents by the construction `\directlua{[...]\the\@pmb@toksa[...]}`. This code behaves similarly to `\the` inside an `\edef`, i. e. the contents of the token register are not further expanded.

```
5.567 \newcommand\pmb@dom@addfeature[4][]{%
5.568   \begingroup%
5.569   \pgfmolbioset[domains]{#1}%
5.570   \@pmb@toksa{#1}%
5.571   \directlua{
5.572     pmbProtein:addFeature{
5.573       key = "#2",
5.574       start = "#3",
5.575       stop = "#4",
5.576       kvList = "\luaescapestring{\the\@pmb@toksa}",
```

```
5.577        level = tonumber("\pmbdomvalueof{level}"),
5.578        layer = tonumber("\pmbdomvalueof{@layer}")
5.579      }
5.580    }%
5.581    \endgroup%
5.582 }
5.583
```

5.5.5 The Main Environment

pmbdomains

> #1: A ⟨*key-value list*⟩ that configures the domain diagram.
> #2: The ⟨*sequence length*⟩.

If **pmbdomains** appears outside of a **tikzpicture**, we implicitly start this environment, otherwise we begin a new group. "Within a **tikzpicture**" means that \useasboundingbox is defined. The ⟨*key-value list*⟩ is processed.

```
5.584 \newif\ifpmb@dom@tikzpicture
5.585
5.586 \newenvironment{pmbdomains}[2][]{%
5.587   \@ifundefined{useasboundingbox}%
5.588     {\pmb@dom@tikzpicturefalse\begin{tikzpicture}}%
5.589     {\pmb@dom@tikzpicturetrue}%
5.590   \pgfmolbioset[domains]{sequence length=#2, #1}%
```

The macros \inputuniprot, \inputgff and \addfeature only point to their respective internal macros (section 5.5.4) within **pmbdomains**.

```
5.591     \let\inputuniprot\pmb@dom@inputuniprot%
5.592     \let\inputgff\pmb@dom@inputgff%
5.593     \let\addfeature\pmb@dom@addfeature%
```

pmbProtein is a new `Protein` object whose `specialKeys` attribute is initialized with the values from the package-wide `SpecialKeys` object. Since `pmbProtein` must know the sequence length and residue numbering before the environment's body is processed, we call `setParameters` twice to ensure that `sequenceLength` is set prior to `residueNumbering`.

```
5.594     \directlua{
5.595       pmbProtein = pgfmolbio.domains.Protein:new()
5.596       pmbProtein.specialKeys =
5.597         pgfmolbio.domains.SpecialKeys:new(pmbSpecialKeys)
5.598       pmbProtein:setParameters{
5.599         sequenceLength = "\pmbdomvalueof{sequence length}"
5.600       }
5.601       pmbProtein:setParameters{
```

```
5.602          residueNumbering = "\pmbdomvalueof{residue numbering}"
5.603        }
5.604      }%
5.605 }{%
```

At the end of `pmbdomains`'s body, `pmbProtein` stores all features that have been defined there. We add one more feature, `other/main chain`, which spans the whole protein and occupies the lowermost layer (this is the only instance where we need the `@layer` key).

```
5.606      \pmb@dom@addfeature[@layer=1]{other/main chain}%
5.607        {(1)}{(\pmbdomvalueof{sequence length})}%
```

The following syntactical gem ensures that the token register `\@pmb@toksa` contains the value of the `name` key without expansion of any macros within the value.

```
5.608      \@pmb@toksa=%
5.609        \expandafter\expandafter\expandafter\expandafter%
5.610        \expandafter\expandafter\expandafter{%
5.611          \pgfkeysvalueof{/pgfmolbio/domains/name}%
5.612        }%
```

Set the remaining attributes of `pmbProtein`.

```
5.613      \directlua{
5.614        pmbProtein:setParameters{
5.615          residueRange = "\pmbdomvalueof{residue range}",
5.616          defaultRulerStepSize = "\pmbdomvalueof{default ruler step size}"
5.617        }
5.618        pmbProtein:setParameters{
5.619          name = "\luaescapestring{\the\@pmb@toksa}",
5.620          xUnit = "\pmbdomvalueof{x unit}",
5.621          yUnit = "\pmbdomvalueof{y unit}",
5.622          residuesPerLine = "\pmbdomvalueof{residues per line}",
5.623          baselineSkip = "\pmbdomvalueof{baseline skip}",
5.624          showRuler = "\ifpmb@dom@showruler true\else false\fi",
5.625          rulerRange = "\pmbdomvalueof{ruler range}",
5.626          sequence = "\pmbdomvalueof{sequence}"
5.627        }
```

Calculate he appropriate levels of disulfide-like features (section 5.6.7). `pgfmolbio .setCoordinateFormat` sets the coordinate output format (section 5.2).

```
5.628        pmbProtein:calculateDisulfideLevels()
5.629        pgfmolbio.setCoordinateFormat(
5.630          "\pgfkeysvalueof{/pgfmolbio/coordinate unit}",
5.631          "\pgfkeysvalueof{/pgfmolbio/coordinate format string}"
5.632        )
```

If the **convert** module is loaded, we open the appropriate output file. If we wish to output final TikZ code, we change `tex.sprint` so that the function writes to

91

this file and then call `printTikzDomains`. Otherwise, we write a string representation of `pmbProtein` to the file (section 5.6.9). Without the convert module, `printTikzDomains` simply returns the drawing commands for the chromatogram to the TeX input stream (section 5.6.8).

```
5.633    \ifpmb@loadmodule@convert
5.634      local filename =
5.635        "\pgfkeysvalueof{/pgfmolbio/convert/output file name}"
5.636      if filename == "(auto)" then
5.637        filename = "pmbconverted" .. pgfmolbio.outputFileId
5.638      end
5.639      filename = filename ..
5.640        ".\pgfkeysvalueof{/pgfmolbio/convert/output file extension}"
5.641      outputFile, ioError = io.open(filename, "w")
5.642      if ioError then
5.643        tex.error(ioError)
5.644      end
5.645      \ifpmb@con@outputtikzcode
5.646        tex.sprint = function(a) outputFile:write(a) end
5.647        pmbProtein:getParameters()
5.648        tex.sprint("\string\n\string\\begin{tikzpicture}")
5.649        pmbProtein:printTikzDomains()
5.650        tex.sprint("\string\n\string\\end{tikzpicture}")
5.651      \else
5.652        \ifpmb@con@includedescription
5.653          pmbProtein.includeDescription = true
5.654        \fi
5.655        outputFile:write(tostring(pmbProtein))
5.656      \fi
5.657      outputFile:close()
5.658      pgfmolbio.outputFileId = pgfmolbio.outputFileId + 1
5.659    \else
5.660      pmbProtein:printTikzDomains()
5.661    \fi
5.662    pmbProtein = nil
5.663  }%
```

At the end of `pmbdomains`, we close an implicitly added `tikzpicture`.

```
5.664    \ifpmb@dom@tikzpicture\else\end{tikzpicture}\fi%
5.665  }
5.666
```

5.5.6 Feature Styles

```
\setdisulfidefeatures
```

#1: A list of ⟨*features*⟩.

Clears the list of disulfide-like features and adds the ⟨*features*⟩ to the empty list. Disulfide-like features are arranged in non-overlapping layers (section 3.6). Depending on whether this macro appears inside a `pmbdomains` environment or not, the appropriate methods of either `pmbProtein.specialKeys` or `pmbSpecialKeys` are called, respectively.

```
5.667  \newcommand\setdisulfidefeatures[1]{%
5.668    \directlua{
5.669      if pmbProtein then
5.670        pmbProtein.specialKeys:clearKeys("disulfideKeys")
5.671        pmbProtein.specialKeys:setKeys("disulfideKeys", "#1", true)
5.672      else
5.673        pmbSpecialKeys:clearKeys("disulfideKeys")
5.674        pmbSpecialKeys:setKeys("disulfideKeys", "#1", true)
5.675      end
5.676    }%
5.677  }
5.678
```

\adddisulfidefeatures

#1: A list of ⟨*features*⟩.

Adds the ⟨*features*⟩ to the list of disulfide-like features without overwriting the current list.

```
5.679  \newcommand\adddisulfidefeatures[1]{%
5.680    \directlua{
5.681      if pmbProtein then
5.682        pmbProtein.specialKeys:setKeys("disulfideKeys", "#1", true)
5.683      else
5.684        pmbSpecialKeys:setKeys("disulfideKeys", "#1", true)
5.685      end
5.686    }%
5.687  }
5.688
```

\removedisulfidefeatures

#1: A list of ⟨*features*⟩.

Removes the ⟨*features*⟩ from the list of disulfide-like features.

```
5.689  \newcommand\removedisulfidefeatures[1]{%
5.690    \directlua{
5.691      if pmbProtein then
5.692        pmbProtein.specialKeys:setKeys("disulfideKeys", "#1", nil)
```

```
5.693        else
5.694          pmbSpecialKeys:setKeys("disulfideKeys", "#1", nil)
5.695        end
5.696    }%
5.697  }
5.698
```

Declare the default disulfide-like features.

```
5.699  \setdisulfidefeatures{DISULFID, disulfide, range}
5.700
```

#1: A ⟨*list*⟩ of features.

#2: Name of a Lua ⟨*function*⟩.

Assigns a feature print ⟨*function*⟩ to each feature in the ⟨*list*⟩. Feature print functions are preprocessors which, for instance, calculate coordinates for features (section 3.8).

```
5.701  \newcommand\setfeatureprintfunction[2]{%
5.702    \directlua{
5.703      if pmbProtein then
5.704        pmbProtein.specialKeys:setKeys("printFunctions", "#1", #2)
5.705      else
5.706        pmbSpecialKeys:setKeys("printFunctions", "#1", #2)
5.707      end
5.708    }%
5.709  }
5.710
```

#1: A ⟨*list*⟩ of features.

Removes any feature print function from the features in the ⟨*list*⟩.

```
5.711  \newcommand\removefeatureprintfunction[1]{%
5.712    \directlua{
5.713      if pmbProtein then
5.714        pmbProtein.specialKeys:setKeys("printFunctions", "#1", nil)
5.715      else
5.716        pmbSpecialKeys:setKeys("printFunctions", "#1", nil)
5.717      end
5.718    }%
5.719  }
5.720
```

Assign default feature print functions.

```
5.721  \setfeatureprintfunction{other/sequence}%
5.722    {pgfmolbio.domains.printSequenceFeature}
5.723  \setfeatureprintfunction{alpha helix, pi helix, 310 helix, HELIX}%
5.724    {pgfmolbio.domains.printHelixFeature}
5.725
```

#1: A ⟨*feature*⟩ name.

#2: A ⟨*style list*⟩.

Sets the style of a ⟨*feature*⟩ to the style described in the ⟨*style list*⟩. Note that the contents of ⟨*style list*⟩ are passed to the Lua function without expansion (via the token register \@pmb@toksa).

```
5.726  \newcommand\setfeaturestyle[2]{%
5.727    \@pmb@toksa{#2}%
5.728    \directlua{
5.729      if pmbProtein then
5.730        pmbProtein.specialKeys:setFeatureStyle(
5.731          "#1", "\luaescapestring{\the\@pmb@toksa}"
5.732        )
5.733      else
5.734        pmbSpecialKeys:setFeatureStyle(
5.735          "#1", "\luaescapestring{\the\@pmb@toksa}"
5.736        )
5.737      end
5.738    }%
5.739  }
5.740
```

Declare default feature styles.

```
5.741  \setfeaturestyle{default}{draw}
5.742  \setfeaturestyle{domain}%
5.743    {fill=Chartreuse,fill=LightSkyBlue,fill=LightPink,fill=Gold!50}
5.744  \setfeaturestyle{signal peptide}{fill=black}
5.745  \setfeaturestyle{propeptide}%
5.746    {*1{fill=Gold, opacity=.5, rounded corners=4pt}}
5.747  \setfeaturestyle{carbohydrate}{red}
5.748  \setfeaturestyle{other/main chain}{*1{draw, line width=2pt, black!25}}
5.749  \setfeaturestyle{other/name}{font=\sffamily}
5.750  \setfeaturestyle{disulfide}{draw=olive}
5.751  \setfeaturestyle{range}{*1{draw,decorate,decoration=brace}}
5.752  \setfeaturestyle{other/ruler}{black, black!50}
5.753  \setfeaturestyle{other/sequence}{*1{font=\ttfamily\tiny}}%
5.754  \setfeaturestyle{other/magnified sequence above}%
5.755    {*1{draw=black!50, densely dashed}}
```

```
5.756  \setfeaturestylealias{other/magnified sequence below}%
5.757    {other/magnified sequence above}
5.758  \setfeaturestyle{alpha helix}{%
5.759    *1{helix front border color=red!50!black,%
5.760    helix front main color=red!90!black,%
5.761    helix front middle color=red!10!white}%
5.762  }
5.763  \setfeaturestylealias{HELIX}{alpha helix}
5.764  \setfeaturestyle{pi helix}{%
5.765    *1{helix front border color=yellow!50!black,%
5.766    helix front main color=yellow!70!red,%
5.767    helix front middle color=yellow!10!white}%
5.768  }
5.769  \setfeaturestyle{310 helix}{%
5.770    *1{helix front border color=magenta!50!black,%
5.771    helix front main color=magenta!90!black,%
5.772    helix front middle color=magenta!10!white}%
5.773  }
5.774  \setfeaturestyle{beta strand}{%
5.775    *1{left color=strand left color, right color=strand right color}%
5.776  }
5.777  \setfeaturestyle{beta turn}{*1{draw=violet, thick}}
5.778  \setfeaturestyle{beta bridge}{*1{fill=MediumBlue}}
5.779  \setfeaturestyle{bend}{*1{draw=magenta, thick}}
```

5.6 `pgfmolbio.domains.lua`

```
6.1   if luatexbase then
6.2     luatexbase.provides_module({
6.3       name           = "pgfmolbio.domains",
6.4       version        = 0.2,
6.5       date           = "2012/10/01",
6.6       description    = "Domain graphs",
6.7       author         = "Wolfgang Skala",
6.8       copyright      = "Wolfgang Skala",
6.9       license        = "LPPL",
6.10    })
6.11  end
6.12
```

These local functions point to functions in `pgfmolbio.lua` (section 5.2).

```
6.13  local stringToDim = pgfmolbio.stringToDim
6.14  local dimToString = pgfmolbio.dimToString
6.15  local packageError = pgfmolbio.packageError
6.16  local packageWarning = pgfmolbio.packageWarning
6.17  local getRange = pgfmolbio.getRange
```

96

5.6.1 Predefined Feature Print Functions

`printSequenceFeature` prints the letters of a sequence between the x-coordinates `xLeft` and `xRight`.

```
6.19  function printSequenceFeature(feature, xLeft, xRight, yMid, xUnit, yUnit)
6.20    xLeft = xLeft + 0.5
6.21    for currResidue in feature.sequence:gmatch(".") do
6.22      tex.sprint("\n\t\t\\def\\xMid{" .. dimToString(xLeft * xUnit) .. "}")
6.23      tex.sprint("\n\t\t\\def\\yMid{" .. dimToString(yMid * yUnit) .. "}")
6.24      tex.sprint("\n\t\t\\def\\currentResidue{" .. currResidue .. "}")
6.25      tex.sprint("\n\t\t\\pmbdomdrawfeature{other/sequence}")
6.26      xLeft = xLeft + 1
6.27    end
6.28  end
6.29
```

`printHelixFeature` prints a helix feature between the x-coordinates `xLeft` and `xRight`.

```
6.30  function printHelixFeature(feature, xLeft, xRight, yMid, xUnit, yUnit)
6.31    local residuesLeft, currX
6.32    tex.sprint("\n\t\t\\pgfmolbioset[domains]{current style}")
6.33
```

Firstly, three different background parts are drawn: one `half upper back` at the left, zero or more `full back` in the middle and possibly one `half lower back` at the right.

```
6.34    residuesLeft = feature.stop - feature.start + 1
6.35    currX = xLeft
6.36    tex.sprint("\n\t\t\\def\\xLeft{" .. dimToString(currX * xUnit) .. "}")
6.37    tex.sprint("\n\t\t\\def\\yMid{" .. dimToString(yMid * yUnit) .. "}")
6.38    tex.sprint("\n\t\t\\pmbdomdrawfeature{helix/half upper back}")
6.39    residuesLeft = residuesLeft - 2
6.40    currX = currX + 2.5
6.41
6.42    while residuesLeft > 0 do
6.43      if residuesLeft == 1 then
6.44        tex.sprint(
6.45          "\n\t\t\\def\\xRight{" ..
6.46          dimToString((currX + 0.5) * xUnit) ..
6.47          "}"
6.48        )
6.49        tex.sprint("\n\t\t\\def\\yMid{" .. dimToString(yMid * yUnit) .. "}")
6.50        tex.sprint("\n\t\t\\pmbdomdrawfeature{helix/half lower back}")
```

97

```
6.51      else
6.52        tex.sprint("\n\t\t\\def\\xMid{" .. dimToString(currX * xUnit) .. "}")
6.53        tex.sprint(
6.54          "\n\t\t\\def\\yLower{" ..
6.55          dimToString(yMid * yUnit - 1.5 * xUnit) ..
6.56          "}"
6.57        )
6.58        tex.sprint("\n\t\t\\pmbdomdrawfeature{helix/full back}")
6.59      end
6.60      residuesLeft = residuesLeft - 2
6.61      currX = currX + 2
6.62    end
6.63
```

Secondly, two different foreground parts are drawn: at least one `full front` at
the left and in the middle, and possibly one `half upper front` at the right.

```
6.64    residuesLeft = feature.stop - feature.start
6.65    currX = xLeft + 1.5
6.66    while residuesLeft > 0 do
6.67      if residuesLeft == 1 then
6.68        tex.sprint(
6.69          "\n\t\t\\def\\xRight{" ..
6.70          dimToString((currX + 0.5) * xUnit) ..
6.71          "}"
6.72        )
6.73        tex.sprint("\n\t\t\\def\\yMid{" .. dimToString(yMid * yUnit) .. "}")
6.74        tex.sprint("\n\t\t\\pmbdomdrawfeature{helix/half upper front}")
6.75      else
6.76        tex.sprint("\n\t\t\\def\\xMid{" .. dimToString(currX * xUnit) .. "}")
6.77        tex.sprint(
6.78          "\n\t\t\\def\\yLower{" ..
6.79          dimToString(yMid * yUnit - 1.5 * xUnit) ..
6.80          "}"
6.81        )
6.82        tex.sprint("\n\t\t\\pmbdomdrawfeature{helix/full front}")
6.83      end
6.84      residuesLeft = residuesLeft - 2
6.85      currX = currX + 2
6.86    end
6.87  end
6.88
```

5.6.2 The `SpecialKeys` Class

The `SpecialKeys` class contains three member variables: `disulfideKeys` (a list of
keys that indicate disulfide-like features, like `disulfide`), `featureStyles` (a list of

feature styles) and `printFunctions` (a list of keys associated with a feature print function, like `alpha helix`). Furthermore, it provides methods to manipulate these fields.

The constructor `SpecialKeys:new` generates a new `SpecialKeys` object and initializes it with values from `parms`.

```
6.89   SpecialKeys = {}
6.90
6.91   function SpecialKeys:new(parms)
6.92     parms = parms or {}
6.93     local newSpecialKeys = {
6.94       disulfideKeys = {},
6.95       featureStyles = {},
6.96       printFunctions = {}
6.97     }
6.98
6.99     for keyList, listContents in pairs(parms) do
6.100      for key, value in pairs(listContents) do
6.101        newSpecialKeys[keyList][key] = value
6.102      end
6.103    end
6.104
6.105    setmetatable(newSpecialKeys, self)
6.106    self.__index = self
6.107    return newSpecialKeys
6.108  end
6.109
```

`SpecialKeys:setKeys` sets a `value` for a `key` in the `keylist`. Possible values for `keyList` are `"disulfideKeys"`, `"featureStyles"` or `"printFunctions"`.

```
6.110  function SpecialKeys:setKeys(keylist, keys, value)
6.111    for key in keys:gmatch("([^,]+)") do
6.112      key = key:trim()
6.113      self[keylist][key] = value
6.114    end
6.115  end
6.116
```

`SpecialKeys:setFeatureStyle` parses the style list `style` and associates it with a certain `key`. In Lua, a style list is an array of tables. Each table contains the fields `cycles` and `style`. `cycles` determines how often the `style` (a string suitable for the mandatory argument of `\pgfmolbioset`) is to be used. In addition, an optional field `alias` contains a reference to another key, if the current key is an alias of it (see below).

```
6.117  function SpecialKeys:setFeatureStyle(key, style)
6.118    local newStyleList, styleCycles, styleContents
```

```
6.119
6.120    newStyleList = {}
6.121    while style ~= "" do
6.122      styleCycles = 1
6.123      if style:sub(1,1) == "{" then
6.124        styleContents = style:match("%b{}")
6.125        style = style:match("%b{}(.*)")
6.126      elseif style:sub(1,1) == "*" then
6.127        styleCycles, styleContents = style:match("%*(%d*)(%b{})")
6.128        if styleCycles == "" then styleCycles = 1 end
6.129        style = style:match("%*%d*%b{}(.*)")
6.130      elseif style:sub(1,1) == "," or style:sub(1,1) == " " then
6.131        style = style:match("[,%s]+(.*)")
6.132        styleCycles, styleContents = nil, nil
6.133      else
6.134        styleContents = style:match("([^,]+),")
6.135        if not styleContents then
6.136          styleContents = style
6.137          style = ""
6.138        else
6.139          style = style:match("[^,]+,(.*)")
6.140        end
6.141      end
6.142      if styleCycles then
6.143        table.insert(
6.144          newStyleList,
6.145          {cycles = styleCycles, style = styleContents}
6.146        )
6.147      end
6.148    end
6.149    self.featureStyles[key] = newStyleList
6.150  end
6.151
```

`SpecialKeys:aliasFeatureStyle` sets the `alias` field of a style list so that feature `newKey` uses the same feature style as feature `oldKey`.

```
6.152  function SpecialKeys:aliasFeatureStyle(newKey, oldKey)
6.153    self.featureStyles[newKey] = {alias = oldKey}
6.154  end
6.155
```

`SpecialKeys:getBaseKey` returns either the name of `key` itself or of its parent key if `key` is an alias.

```
6.156  function SpecialKeys:getBaseKey(key)
6.157    if self.featureStyles[key] then
6.158      if self.featureStyles[key].alias then
6.159        return self.featureStyles[key].alias
```

```
6.160        end
6.161      end
6.162      return key
6.163   end
6.164
```

SpecialKeys:clearKeys clears a keylist.

```
6.165   function SpecialKeys:clearKeys(keylist)
6.166      self[keylist] = {}
6.167   end
6.168
```

SpecialKeys:selectStyleFromList returns the styleID-th style from the style
list associated with key. Firstly, the correct style list is selected.

```
6.169   function SpecialKeys:selectStyleFromList(key, styleID)
6.170      local styleList
6.171
6.172      if not self.featureStyles[key] then
6.173        packageWarning(
6.174          "Feature style `" ..
6.175          key ..
6.176          "' unknown, using `default'."
6.177          )
6.178        styleList = self.featureStyles.default
6.179      elseif self.featureStyles[key].alias then
6.180        styleList = self.featureStyles[self.featureStyles[key].alias]
6.181      else
6.182        styleList = self.featureStyles[key]
6.183      end
6.184
```

Secondly, the method choses the appropriate style in the list.

```
6.185      while true do
6.186        for _, v in ipairs(styleList) do
6.187          styleID = styleID - v.cycles
6.188          if styleID < 1 then
6.189            return v.style
6.190          end
6.191        end
6.192      end
6.193   end
6.194
```

101

5.6.3 The `Protein` Class

The `Protein` class represents a domain diagram in Lua. Its member variables largely correspond to the keys of the **domains** module. In detail:

- `sequenceLength`: A value of `-1` indicates that the sequence length has not been properly set.

- `ft` is the feature table, i. e. an array of tables with the following fields:
  - `key`: A string that equals the feature key.
  - `start`: The start …
  - `stop`: … and the end residue of the feature, both in *absolute* numbering. (For the difference between absolute and relative numbering, see section 3.3.)
  - `kvList`: A string containing comma-separated key-value pairs, which is passed to `\pgfmolbioset` immediately before the feature is drawn.
  - `level`: The level of the feature (only relevant for disulfide-like features).

- `residueNumbering`: An array of strings. The indices are absolute residue numbers, while the fields represent the corresponding relative residue numbers.

- `revResidueNumbering`: The inverse of `residueNumbering` (i. e., a table of numbers).

- `rulerRange`: An array of tables. Each table represents one mark of the ruler and has the fields `pos` (position in absolute residue numbers) and `number` (relative number of the marked residue).

- `currentStyle`: A table whose field names equal feature keys. Each field denotes the index of the style that was last selected from that feature's style list.

- `includeDescription`: This boolean field remains uninitialized. Instead, it is directly set in `pgfmolbio.domains.tex` if the `convert` module is loaded and the user requests a string representation of a `Protein` object (section 5.6.9).

The constructor `Protein:new` initializes the member variables with default values.

```
6.195  Protein = {}
6.196
6.197  function Protein:new()
6.198    local newProtein = {
6.199      name = "",
6.200      sequenceLength = -1,
6.201      ft = {},
6.202      sequence = "",
6.203      xUnit = stringToDim("0.5mm"),
```

```
6.204        yUnit = stringToDim("6mm"),
6.205        residuesPerLine = 250,
6.206        residueRangeMin = 1,
6.207        residueRangeMax = 100,
6.208        residueNumbering = {},
6.209        revResidueNumbering = {},
6.210        baselineSkip = 3,
6.211        rulerRange = {},
6.212        defaultRulerStepSize = 50,
6.213        showRuler = true,
6.214        currentStyle = {},
6.215        specialKeys = SpecialKeys:new()
6.216      }
6.217      setmetatable(newProtein, self)
6.218      self.__index = self
6.219      return newProtein
6.220 end
6.221
```

`Protein:toAbsoluteResidueNumber` converts a string that either contains an absolute or relative residue number to an absolute residue number.

```
6.222 function Protein:toAbsoluteResidueNumber(value)
6.223   local result = value:match("%b()")
6.224   if result then
6.225     result = tonumber(result:sub(2, -2))
6.226   else
6.227     result = self.revResidueNumbering[(value:gsub("[<>%?]", ""))]
6.228   end
6.229   if not result then
6.230     packageError("Bad or missing start/end point value: " .. value)
6.231   end
6.232   return result
6.233 end
6.234
```

5.6.4 Uniprot and GFF Files

`Protein:readUniprotFile` reads the relevant parts of Uniprot file `filename`[2].

```
6.235 function Protein:readUniprotFile(filename)
6.236   local uniprotFile, errorMsg = io.open(filename, "r")
6.237   if not uniprotFile then packageError(errorMsg) end
6.238
```

---

[2]For a detailed description of this format, see `http://web.expasy.org/docs/userman.html`.

Each line in a Uniprot file starts with a line code consisting of two letters. This code determines the syntax of the remainder of the line.

```lua
6.239  local sequence = {}
6.240  local inSequence = false
6.241  local featureTable = {}
6.242
6.243  for currLine in uniprotFile:lines() do
6.244    local lineCode = currLine:sub(1, 2)
6.245    local lineContents = currLine:sub(3)
```

The `ID` line is the first line in a Uniprot file. It provides two relevant properties of the protein, namely its name and ints sequence length. For example, in the file `SampleUniprot.txt` (see section 3.10), the `ID` line reads
`ID   TestProtein   Reviewed;        200 AA.`
which declares a protein with 200 residues called TestProtein.

```lua
6.246    if lineCode == "ID" then
6.247      local name, sequenceLength =
6.248        lineContents:match("%s*(%S+)%s*%a+;%s*(%d+)%s*AA%.")
6.249      self.name = name
6.250      self.sequenceLength = tonumber(sequenceLength)
6.251      self.residueRangeMax = self.sequenceLength
```

`FT` lines describe features of the protein (domains, disulfides, sugars etc.). The first line of a feature always contains its key (columns 6–13) and endpoints (columns 15–20 and 22–27, respectively). The description (columns 35–75) may span several lines, in which case the key columns of consecutive lines are empty. For instance,
`FT   DOMAIN      10      40        Domain 1`
declares a `DOMAIN` feature between residues 10 and 40 with description "Domain 1".

```lua
6.252    elseif lineCode == "FT" then
6.253      local key = currLine:sub(6, 13):trim()
6.254      local start, stop, description =
6.255        currLine:sub(15, 20), currLine:sub(22, 27), currLine:sub(35, 75)
6.256      if key ~= "" then
6.257        table.insert(featureTable, {
6.258          key = key,
6.259          start = "(" .. start .. ")",
6.260          stop = "(" .. stop .. ")",
6.261          description = description,
6.262          style = "",
6.263          kvList = ""
6.264        })
6.265      else
6.266        featureTable[#featureTable].description =
6.267          featureTable[#featureTable].description .. description
6.268      end
```

104

The `SQ` line starts the sequence block. Each of the following sequence data lines lacks a line code and shows the amino acid sequence in one letter code, e. g.

```
SQ    SEQUENCE   200 AA;   22041 MW;   00A52FE2EC5431D9 CRC64;
       MGSKRSVPSR HRSLTTYEVM FAVLFVILVA LCAGLIAVSW LSIQ [...]
```

```
6.269    elseif lineCode == "SQ" then
6.270      inSequence = true
6.271    elseif lineCode == "  " and inSequence then
6.272      table.insert(sequence, (lineContents:gsub("%s+", "")))
```

The `\\` line terminates the Uniprot file.

```
6.273    elseif lineCode == "\\\\" then
6.274      break
6.275    end
6.276  end
```

After closing the file, features are converted to the proper format (section 5.6.3).

```
6.277  uniprotFile:close()
6.278  if next(sequence) then self.sequence = table.concat(sequence) end
6.279  for _, v in ipairs(featureTable) do self:addFeature(v) end
6.280  end
6.281
```

`Protein:readGffFile` reads the relevant parts of General Feature Format file `filename`[3].

```
6.282  function Protein:readGffFile(filename)
6.283    local gffFile, errorMsg = io.open(filename, "r")
6.284    local lineContents, fields, lineNumber
6.285
6.286    if not gffFile then packageError(errorMsg) end
```

Each line in a `gff` file describes a feature and consists of up to 9 tabulator-separated fields, of which only fields 3 (key), 4 (start) and 5 (end) are required for the `domains` module. Everything following the comment sign (`#`) on a line is ignored.

```
6.287    lineNumber = 1
6.288    for currLine in gffFile:lines() do
6.289      lineContents = currLine:gsub("#.*$", "")
6.290      fields = {}
6.291      if lineContents ~= "" then
6.292        for currField in lineContents:gmatch("([^\t]+)") do
6.293          table.insert(fields, currField)
6.294        end
```

---

[3]For a detailed description of this format, see `http://http://www.sanger.ac.uk/resources/software/gff/spec.html`.

```
6.295        if not fields[5] then
6.296          packageError("Bad line (" .. lineNumber .. ") in gff file '" ..
6.297            filename .. "':\n" .. currLine)
6.298          break
6.299        end
6.300        self:addFeature{
6.301          key = fields[3],
6.302          start = "(" .. fields[4] .. ")",
6.303          stop = "(" .. fields[5] .. ")",
6.304          description = fields[9] or "",
6.305          style = "",
6.306          kvList = ""
6.307        }
6.308      end
6.309      lineNumber = lineNumber + 1
6.310    end
6.311    gffFile:close()
6.312 end
6.313
```

5.6.5 Getter and Setter Methods

`Protein:getParameters` informs TEX of the protein name, sequence and sequence length. This method is called after reading a Uniprot file (section 5.5.4).

```
6.314 function Protein:getParameters()
6.315    tex.sprint(
6.316      "\\pgfmolbioset[domains]{name={" ..
6.317      self.name ..
6.318      "},sequence={" ..
6.319      self.sequence ..
6.320      "},sequence length=" ..
6.321      self.sequenceLength ..
6.322      "}"
6.323    )
6.324 end
6.325
```

`Protein:setParameters` passes options from the domains module to the Lua script. Each field of the table `keyHash` is named after a `Protein` attribute and represents a function that receives one string parameter (the value of a LATEX key).

```
6.326 function Protein:setParameters(newParms)
6.327    local keyHash = {
```

`keyHash.sequenceLength` checks for an invalid sequence length.

```
6.328      sequenceLength = function(v)
6.329         v = tonumber(v)
6.330         if not v then return self.sequenceLength end
6.331         if v < 1 then
6.332           packageError("Sequence length must be larger than zero.")
6.333         end
6.334         return v
6.335      end,
```

keyHash.residueNumbering generates the residue numbering array and its inverse (described in section 5.6.3).

```
6.336      residueNumbering = function(v)
6.337         local ranges = {}
6.338         local start, startNumber, startLetter, stop
6.339         self.revResidueNumbering = {}
6.340         if v:trim() == "auto" then
6.341           for i = 1, self.sequenceLength do
6.342             table.insert(ranges, tostring(i))
6.343           end
6.344         else --example list: `1-4,5,6A-D'
6.345           for _, value in ipairs(v:explode(",+")) do
6.346             value = value:trim()
6.347             start, stop = value:match("(%w*)%s*%-%s*(%w*)$")
6.348             if not start then
6.349               start = value:match("(%w*)")
6.350             end
6.351             if not start or start == "" then --invalid range
6.352               packageError("Unknown residue numbering range: " .. value)
6.353             end
6.354             if stop then
6.355               if tonumber(start) and tonumber(stop) then
6.356                 --process range `1-4'
6.357                 for currNumber = tonumber(start), tonumber(stop) do
6.358                   table.insert(ranges, tostring(currNumber))
6.359                 end
6.360               else --process range `6A-D'
6.361                 startNumber, startLetter = start:match("(%d*)(%a)")
6.362                 stop = stop:match("(%a)")
6.363                 for currLetter = startLetter:byte(), stop:byte() do
6.364                   table.insert(ranges,
6.365                     startNumber .. string.char(currLetter))
6.366                 end
6.367               end
6.368             else --process range `5'
6.369               table.insert(ranges, start)
6.370             end
6.371           end
```

```
6.372          end
6.373          for i, value in ipairs(ranges) do
6.374            if self.revResidueNumbering[value] then
6.375              packageError("The range value " .. value ..
6.376                " appears more than once.")
6.377            else
6.378              self.revResidueNumbering[value] = i
6.379            end
6.380          end
6.381          return ranges
6.382        end,
```

keyHash.residueRange sets the residue range, treating possible errors.

```
6.383        residueRange = function(v)
6.384          local num
6.385          local residueRangeMin, residueRangeMax =
6.386            getRange(v:trim(), "^([%w%(%)]+)%s*%-", "%-%s*([%w%(%)]+)$")
6.387          if residueRangeMin == "auto" then
6.388            self.residueRangeMin = 1
6.389          else
6.390            num = residueRangeMin:match("%b()")
6.391            if num then
6.392              self.residueRangeMin = tonumber(num:sub(2, -2))
6.393            elseif self.revResidueNumbering[residueRangeMin] then
6.394              self.residueRangeMin = self.revResidueNumbering[residueRangeMin]
6.395            else
6.396              packageError("Invalid residue range: " .. residueRangeMin)
6.397            end
6.398          end
6.399
6.400          if residueRangeMax == "auto" then
6.401            self.residueRangeMax = self.sequenceLength
6.402          else
6.403            num = residueRangeMax:match("%b()")
6.404            if num then
6.405              self.residueRangeMax = tonumber(num:sub(2, -2))
6.406            elseif self.revResidueNumbering[residueRangeMax] then
6.407              self.residueRangeMax = self.revResidueNumbering[residueRangeMax]
6.408            else
6.409              packageError("Invalid residue range: " .. residueRangeMax)
6.410            end
6.411          end
6.412
6.413          if self.residueRangeMin >= self.residueRangeMax then
6.414            packageError("Residue range is smaller than 1.")
6.415          end
6.416        end,
```

The following fields map to functions already defined.

```
6.417    defaultRulerStepSize = tonumber,
6.418    name = tostring,
6.419    sequence = tostring,
6.420    xUnit = stringToDim,
6.421    yUnit = stringToDim,
6.422    residuesPerLine = tonumber,
6.423    baselineSkip = tonumber,
```

`keyHash.rulerRange` sets the ruler range, treating possible errors and inconsistencies (for example, if the upper ruler range exceeds the upper residue range).

```
6.424    rulerRange = function(v)
6.425      local num
6.426      local ranges = {}
6.427      local rulerRangeMin, rulerRangeMax, rulerRangeStep
6.428      for _, value in ipairs(v:explode(",+")) do
6.429        rulerRangeMin, rulerRangeMax, rulerRangeStep =
6.430          getRange(value:trim(), "^([%w%(%)]+)",
6.431            "%-%s*([%w%(%)]+)", "step%s*(%d+)$")
6.432
6.433        if rulerRangeMin == "auto" then
6.434          rulerRangeMin = self.residueRangeMin
6.435        else
6.436          num = rulerRangeMin:match("%b()")
6.437          if num then
6.438            rulerRangeMin = tonumber(num:sub(2, -2))
6.439          elseif self.revResidueNumbering[rulerRangeMin] then
6.440            rulerRangeMin = self.revResidueNumbering[rulerRangeMin]
6.441          else
6.442            packageError("Invalid lower ruler range: " .. rulerRangeMin)
6.443          end
6.444        end
6.445
6.446        if rulerRangeMax then
6.447          if rulerRangeMax == "auto" then
6.448            rulerRangeMax = self.residueRangeMax
6.449          else
6.450            num = rulerRangeMax:match("%b()")
6.451            if num then
6.452              rulerRangeMax = tonumber(num:sub(2, -2))
6.453            elseif self.revResidueNumbering[rulerRangeMax] then
6.454              rulerRangeMax = self.revResidueNumbering[rulerRangeMax]
6.455            else
6.456              packageError("Invalid upper ruler range: " .. rulerRangeMax)
6.457            end
6.458          end
6.459
```

```
6.460            if rulerRangeMin >= rulerRangeMax then
6.461              packageError("Ruler range is smaller than 1.")
6.462            end
6.463            if rulerRangeMin < self.residueRangeMin then
6.464              rulerRangeMin = self.residueRangeMin
6.465              packageWarning(
6.466                "Lower ruler range is smaller than" ..
6.467                "lower residue range. It was adjusted to " ..
6.468                rulerRangeMin .. "."
6.469              )
6.470            end
6.471            if rulerRangeMax > self.residueRangeMax then
6.472              rulerRangeMax = self.residueRangeMax
6.473              packageWarning(
6.474                "Upper ruler range exceeds" ..
6.475                "upper residue range. It was adjusted to " ..
6.476                rulerRangeMax .. "."
6.477              )
6.478            end
6.479          else
6.480            rulerRangeMax = rulerRangeMin
6.481          end
6.482          rulerRangeStep = tonumber(rulerRangeStep)
6.483            or self.defaultRulerStepSize
6.484
6.485          for i = rulerRangeMin, rulerRangeMax, rulerRangeStep do
6.486            table.insert(
6.487              ranges,
6.488              {pos = i, number = self.residueNumbering[i]}
6.489            )
6.490          end
6.491        end
6.492        return ranges
6.493      end,
```

`keyHash.showRuler` determines if the ruler is visible.

```
6.494      showRuler = function(v)
6.495        if v == "true" then return true else return false end
6.496      end
6.497    }
```

We iterate over all fields in the argument of `setParameters`. If a field of the same name exists in `keyHash`, we call this field with the value of the corresponding field in `newParms` as parameter.

```
6.498    for key, value in pairs(newParms) do
6.499      if keyHash[key] then
6.500        self[key] = keyHash[key](value)
```

```
6.501          if pgfmolbio.errorCatched then return end
6.502        end
6.503      end
6.504  end
6.505
```

5.6.6 Adding Feature

`Protein:addFeature` converts raw feature information to the format of `ft` fields
(described in section 5.6.3). Firstly, the method determines the index of the style
that should be used for the current feature.

```
6.506  function Protein:addFeature(newFeature)
6.507    local baseKey, ftEntry
6.508
6.509    baseKey = self.specialKeys:getBaseKey(newFeature.key)
6.510    if self.currentStyle[baseKey] then
6.511      self.currentStyle[baseKey] = self.currentStyle[baseKey] + 1
6.512    else
6.513      self.currentStyle[baseKey] = 1
6.514    end
6.515
```

Then, a new field for the feature table is set up.

```
6.516    ftEntry = {
6.517      key = newFeature.key,
6.518      start = self:toAbsoluteResidueNumber(newFeature.start),
6.519      stop = self:toAbsoluteResidueNumber(newFeature.stop),
6.520      kvList = "style={" ..
6.521        self.specialKeys:selectStyleFromList(baseKey,
6.522          self.currentStyle[baseKey]) .. "}",
6.523      level = newFeature.level or nil
6.524    }
```

Finally, the key-value list `kvList` is modified (if applicable) and the new field is
inserted into `ft`.

```
6.525    if newFeature.kvList ~= "" then
6.526      ftEntry.kvList = ftEntry.kvList .. "," .. newFeature.kvList
6.527    end
6.528    if newFeature.description then
6.529      ftEntry.kvList = ftEntry.kvList ..
6.530        ",description={" .. newFeature.description .. "}"
6.531      ftEntry.description = newFeature.description
6.532    end
6.533    table.insert(self.ft, newFeature.layer or #self.ft + 1, ftEntry)
6.534  end
```

5.6.7 Calculate Disulfide Levels

`Protein:calculateDisulfideLevels` arranges disulfide-like features in non-over-lapping levels.

```
6.536  function Protein:calculateDisulfideLevels()
6.537    if pgfmolbio.errorCatched then return end
6.538    local disulfideGrid, currLevel, levelFree
6.539    disulfideGrid = {}
6.540
6.541    for i, v in ipairs(self.ft) do
6.542      if self.specialKeys.disulfideKeys[v.key] then
```

If the `level` field of a disulfide-like feature is already specified, it overrides the automatic mechanism of level determination. This may lead to clashes.

```
6.543        if v.level then
6.544          if not disulfideGrid[v.level] then
6.545            disulfideGrid[v.level] = {}
6.546          end
6.547          for currPos = v.start, v.stop do
6.548            disulfideGrid[v.level][currPos] = true
6.549          end
```

Otherwise, the algorithm looks for the first free level (starting at level 1), i.e. the first level the feature may occupy without clashing with another one. (1) If the level currently checked already exists, it has been created by a previous disulfide-like feature. In this case, it is considered free if the previous feature does not overlap with the current one.

```
6.550        else
6.551          currLevel = 1
6.552          repeat
6.553            levelFree = true
6.554            if disulfideGrid[currLevel] then
6.555              for currPos = v.start, v.stop do
6.556                levelFree = levelFree
6.557                  and not disulfideGrid[currLevel][currPos]
6.558              end
6.559              if levelFree then
6.560                self.ft[i].level = currLevel
6.561                for currPos = v.start, v.stop do
6.562                  disulfideGrid[currLevel][currPos] = true
6.563                end
6.564              end
```

(2) If the level currently checked does not exist, it must be free.

```
6.565            else
6.566              self.ft[i].level = currLevel
6.567              disulfideGrid[currLevel] = {}
6.568              for currPos = v.start, v.stop do
6.569                disulfideGrid[currLevel][currPos] = true
6.570              end
6.571              levelFree = true
6.572            end
6.573            currLevel = currLevel + 1
6.574          until levelFree == true
6.575        end
6.576      end
6.577    end
6.578 end
6.579
```

5.6.8 Print Domains

`Protein:printTikzDomains` is the heart of the Lua script, since it converts a `Protein` object to TEX code.

```
6.580 function Protein:printTikzDomains()
6.581   if pgfmolbio.errorCatched then return end
6.582   local xLeft, xMid, xRight, yMid, xLeftClip, xRightClip,
6.583     currLine, residuesLeft, currStyle
6.584
```

(1) Features (excluding `other/ruler` and `other/name`) For each feature in the feature table, we first calculate its coordinates (`xLeft`, `xMid`, `xRight` and `yMid`) and clipped areas (`xLeftClip`, `xRightClip`).

```
6.585   for _, currFeature in ipairs(self.ft) do
6.586     currLine = 0
6.587     xLeft = currFeature.start - self.residueRangeMin -
6.588       currLine * self.residuesPerLine + 1
6.589     while xLeft > self.residuesPerLine do
6.590       xLeft = xLeft - self.residuesPerLine
6.591       currLine = currLine + 1
6.592     end
6.593     xLeft = xLeft - 1
6.594     xRight = currFeature.stop - self.residueRangeMin -
6.595       currLine * self.residuesPerLine + 1
6.596     residuesLeft = self.residueRangeMax - self.residueRangeMin -
6.597       currLine * self.residuesPerLine + 1
6.598     xLeftClip = stringToDim("-5cm")
```

113

```
6.599      xRightClip = self.residuesPerLine * self.xUnit
6.600
6.601      if currFeature.start <= self.residueRangeMax
6.602          and currFeature.stop >= self.residueRangeMin then
6.603        repeat
6.604          if residuesLeft <= self.residuesPerLine then
6.605            if residuesLeft < xRight then
6.606              xRightClip = residuesLeft * self.xUnit
6.607            else
6.608              xRightClip = xRight * self.xUnit + stringToDim("5cm")
6.609            end
6.610          else
6.611            if xRight <= self.residuesPerLine then
6.612              xRightClip = xRight * self.xUnit + stringToDim("5cm")
6.613            end
6.614          end
6.615          if xLeft < 0 then xLeftClip = stringToDim("0cm") end
6.616
6.617          xMid = (xLeft + xRight) / 2
6.618          yMid = -currLine * self.baselineSkip
```

The current feature is extended by any level and sequence information present.

```
6.619              if currFeature.level then
6.620                currFeature.kvList = currFeature.kvList ..
6.621                  ",level=" .. currFeature.level
6.622              end
6.623              currFeature.sequence =
6.624                self.sequence:sub(currFeature.start, currFeature.stop)
6.625
```

Each feature appears within its own `scope`. A `pgfinterruptboundingbox` ensures that the bounding box of the picture ignores the feature, since the `\clip` macro would enlarge it too much. Auxiliary macros for `\setfeatureshape` are defined (section 3.4).

```
6.626              tex.sprint("\n\t\\begin{scope}\\begin{pgfinterruptboundingbox}")
6.627              tex.sprint("\n\t\t\\def\\xLeft{" ..
6.628                dimToString(xLeft * self.xUnit) .. "}")
6.629              tex.sprint("\n\t\t\\def\\xMid{" ..
6.630                dimToString(xMid * self.xUnit) .. "}")
6.631              tex.sprint("\n\t\t\\def\\xRight{" ..
6.632                dimToString(xRight * self.xUnit) .. "}")
6.633              tex.sprint("\n\t\t\\def\\yMid{" ..
6.634                dimToString(yMid * self.yUnit) .. "}")
6.635              tex.sprint("\n\t\t\\def\\featureSequence{" ..
6.636                currFeature.sequence .. "}")
6.637              tex.sprint(
6.638                "\n\t\t\\clip (" ..
```

```
6.639          dimToString(xLeftClip) ..
6.640          ", \\yMid + " ..
6.641          dimToString(stringToDim("10cm")) ..
6.642          ") rectangle (" ..
6.643          dimToString(xRightClip) ..
6.644          ", \\yMid - " ..
6.645          dimToString(stringToDim("10cm")) ..
6.646          ");"
6.647        )
6.648        tex.sprint(
6.649          "\n\t\t\\pgfmolbioset[domains]{" ..
6.650          currFeature.kvList ..
6.651          "}"
6.652        )
```

We invoke either the print function associated with the current feature or directly
call \pmbdomdrawfeature. Afterwards, we close both surrounding environments.

```
6.653        if self.specialKeys.printFunctions[currFeature.key] then
6.654          self.specialKeys.printFunctions[currFeature.key](
6.655            currFeature, xLeft, xRight, yMid, self.xUnit, self.yUnit)
6.656        else
6.657          tex.sprint("\n\t\t\\pmbdomdrawfeature{" ..
6.658            currFeature.key .. "}")
6.659        end
6.660        tex.sprint("\n\t\\end{pgfinterruptboundingbox}\\end{scope}")
6.661
```

Calculate coordinates for the next line of the feature.

```
6.662        currLine = currLine + 1
6.663        xLeft = xLeft - self.residuesPerLine
6.664        xRight = xRight - self.residuesPerLine
6.665        residuesLeft = residuesLeft - self.residuesPerLine
6.666      until xRight < 1 or residuesLeft < 1
6.667    end
6.668  end
6.669
```

(2) Feature other/ruler   The ruler requires special treatment, buth the algo-
rithm is actually simple: For each marker, calculate its coordinates, select its style
and print it.

```
6.670  if self.showRuler then
6.671    currStyle = 1
6.672    tex.sprint("\n\t\\begin{scope}")
6.673    for _, currRuler in ipairs(self.rulerRange) do
```

```
6.674        currLine = 0
6.675        xMid = currRuler.pos - self.residueRangeMin -
6.676          currLine * self.residuesPerLine + 1
6.677        while xMid > self.residuesPerLine do
6.678          xMid = xMid - self.residuesPerLine
6.679          currLine = currLine + 1
6.680        end
6.681        xMid = xMid - 0.5
6.682        yMid = -currLine * self.baselineSkip
6.683        tex.sprint(
6.684          "\n\t\t\\pgfmolbioset[domains]{current style/.style={" ..
6.685          self.specialKeys:selectStyleFromList("other/ruler", currStyle) ..
6.686          "}}"
6.687        )
6.688        tex.sprint("\n\t\t\t\\def\\xMid{" ..
6.689          dimToString(xMid * self.xUnit) .. "}")
6.690        tex.sprint("\n\t\t\t\\let\\xLeft\\xMid\\let\\xRight\\xMid")
6.691        tex.sprint("\n\t\t\t\\def\\yMid{" ..
6.692          dimToString(yMid * self.yUnit) .. "}")
6.693        tex.sprint("\n\t\t\t\\def\\residueNumber{" ..
6.694          currRuler.number .. "}")
6.695        tex.sprint("\n\t\t\t\\pmbdomdrawfeature{other/ruler}")
6.696        currStyle = currStyle + 1
6.697      end
6.698      tex.sprint("\n\t\\end{scope}")
6.699    end
6.700
```

(3) Feature `other/name` Similarly, we calculate the coordinates of the name and print it.

```
6.701  xMid =
6.702    math.min(
6.703      self.residuesPerLine,
6.704      self.residueRangeMax - self.residueRangeMin + 1
6.705    ) / 2
6.706  tex.sprint("\n\t\\begin{scope}")
6.707  tex.sprint(
6.708    "\n\t\t\\pgfmolbioset[domains]{current style/.style={" ..
6.709    self.specialKeys:selectStyleFromList("other/name", 1) ..
6.710    "}}"
6.711  )
6.712  tex.sprint("\n\t\t\\def\\xLeft{0mm}")
6.713  tex.sprint("\n\t\t\\def\\xMid{" .. dimToString(xMid * self.xUnit) .. "}")
6.714  tex.sprint("\n\t\t\\def\\xRight{" ..
6.715    dimToString(self.residuesPerLine * self.xUnit) .. "}")
6.716  tex.sprint("\n\t\t\\def\\yMid{0mm}")
6.717  tex.sprint("\n\t\t\\pmbdomdrawfeature{other/name}")
```

116

```
6.718    tex.sprint("\n\t\\end{scope}")
6.719
```

(4) Set bounding box The bounding box is determined manually in order to prevent excessive enlargement due to clipping. The top left corner of the bounding box is the coordinate (`enlarge left`, `enlarge top`).

```
6.720    tex.sprint(
6.721       "\n\t\\pmbprotocolsizes{" ..
6.722       "\\pmbdomvalueof{enlarge left}}{\\pmbdomvalueof{enlarge top}}"
6.723    )
```

The *x*-coordinate of its right border is the largest line width plus the value of `enlarge right`. The *y*-coordinate of its bottom border is that of the lowermost line plus the value of `enlarge bottom`.

```
6.724    currLine =
6.725       math.ceil(
6.726          (self.residueRangeMax - self.residueRangeMin + 1) /
6.727             self.residuesPerLine
6.728       ) - 1
6.729    xRight =
6.730       math.min(
6.731          self.residuesPerLine,
6.732          self.residueRangeMax - self.residueRangeMin + 1
6.733       )
6.734    tex.sprint(
6.735       "\n\t\\pmbprotocolsizes{" ..
6.736       dimToString(xRight * self.xUnit) ..
6.737       " + \\pmbdomvalueof{enlarge right}}{" ..
6.738       dimToString(-currLine * self.baselineSkip * self.yUnit) ..
6.739       " + \\pmbdomvalueof{enlarge bottom}}"
6.740    )
6.741 end
6.742
```

5.6.9 Converting a `Protein` to a String

`Protein:__tostring` is required by the **convert** module and returns a `pmbdomains` environment that contains all the information stored in the `Protein` object (section 4.3). Firstly, we start the environment.

```
6.743 function Protein:__tostring()
6.744    local result = {}
6.745    local currLine
6.746
```

117

```
6.747    currLine = "\\begin{pmbdomains}\n\t\t[name={" ..
6.748      self.name ..
6.749      "}"
6.750    if self.sequence ~= "" then
6.751      currLine = currLine ..
6.752        ",\n\t\tsequence=" ..
6.753        self.sequence
6.754    end
6.755    currLine = currLine ..
6.756      "]{" ..
6.757      self.sequenceLength ..
6.758      "}"
6.759    table.insert(result, currLine)
6.760
```

Afterwards, each feature in the feature table is converted to an `\addfeature` macro. Note the use of the `includeDescription` field (described in section 5.6.3).

```
6.761    for i, v in ipairs(self.ft) do
6.762      if v.key ~= "other/main chain" then
6.763        currLine = "\t\\addfeature"
6.764        if self.includeDescription and v.description then
6.765          currLine =
6.766            currLine ..
6.767            "[description={" ..
6.768            v.description ..
6.769            "}]"
6.770        end
6.771        currLine =
6.772          currLine ..
6.773          "{" ..
6.774          v.key ..
6.775          "}{" ..
6.776          v.start ..
6.777          "}{" ..
6.778          v.stop ..
6.779          "}"
6.780        table.insert(result, currLine)
6.781      end
6.782    end
```

Finally, we close the environment.

```
6.783    table.insert(result,
6.784      "\\end{pmbdomains}"
6.785    )
6.786    return table.concat(result, "\n")
6.787 end
```

5.7 `pgfmolbio.convert.tex`

The code for the convert module is short: We only need to declare four options and set `\pdfdraftmode` to 1 in order to prevent pdfTEX from producing any `pdf` output.

```
7.1  \pdfdraftmode1
7.2
7.3  \pgfkeyssetvalue{/pgfmolbio/convert/output file name}{(auto)}
7.4  \pgfkeyssetvalue{/pgfmolbio/convert/output file extension}{tex}
7.5
7.6  \pgfmolbioset[convert]{%
7.7    output code/.is choice,
7.8    output code/tikz/.code=\pmb@con@outputtikzcodetrue,
7.9    output code/pgfmolbio/.code=\pmb@con@outputtikzcodefalse,
7.10   output code=tikz
7.11 }
7.12
7.13 \pgfmolbioset[convert]{%
7.14   include description/.is if=pmb@con@includedescription,
7.15   include description
7.16 }
```