Supplementary Information

The AnnotationSketch genome annotation drawing library

Sascha Steinbiss, Gordon Gremme, Christin Schärfer, Malte Mader and Stefan Kurtz

August 23, 2012

Contents

1 AnnotationSketch

AnnotationSketch is a versatile and efficient C-based drawing library for GFF3-compatible genomic annotations. It is included in the *GenomeTools* distribution. In addition to the native C interface, bindings to the Lua, Python and Ruby programming languages are provided.

1.1 Overview

AnnotationSketch consists of several classes, which take part in three visualization *phases* (see Fig. 1.1).

1.1.1 Phase 1: Feature selection

The GFF3 input data are parsed into a directed acyclic graph (*annotation graph*, see Fig. 1.2 for an example) whose nodes correspond to single features (i.e. lines from the GFF3 file). Consequently, edges in the graph represent the *part-of* relationships between groups of genomic features according to the Sequence Ontology hierarchy. Note that GFF3 input files *must* be valid according to the GFF3 specification to ensure that they can be read for *AnnotationSketch* drawing or any other kind of manipulation using *GenomeTools*. A validating GFF3 parser is available in *GenomeTools* (and can be run using `gt gff3validator`).
Each top-level node (which is a node without a parent) is then registered into a persistent *FeatureIndex* object. The *FeatureIndex* holds a collection of the top-level nodes of all features in each sequence region in an interval tree data structure that can be efficiently queried for features in a genomic region of interest. All child nodes of the top-level node are then available by the use of traversal functions. Alternatively, annotation graphs can be built by the user by creating each node explicitly and then connecting the nodes in a way such that the relationships are reflected in the graph structure (see examples section for example annotation graph building code).

1.1.2 Phase 2: Layout

The next step consists of processing the features (given via a *FeatureIndex* or a simple array of top level nodes) into a *Diagram* object which represents a single view of the annotations of a genomic region. First, semantic units are formed from the annotation subgraphs. This is done by building *blocks* from connected features by grouping and overlaying them according to several user-defined collapsing options (see "Collapsing"). By default, a separate *track* is then created for each Sequence Ontology feature type. Alternatively, if more granularity in track assignment is desired, *track selector* functions can be used to create tracks and assign blocks to them based on arbitrary feature characteristics. This is simply done by creating a unique identifier string per track. The *Diagram* object can also be used to hold one or more *custom tracks*, which allow

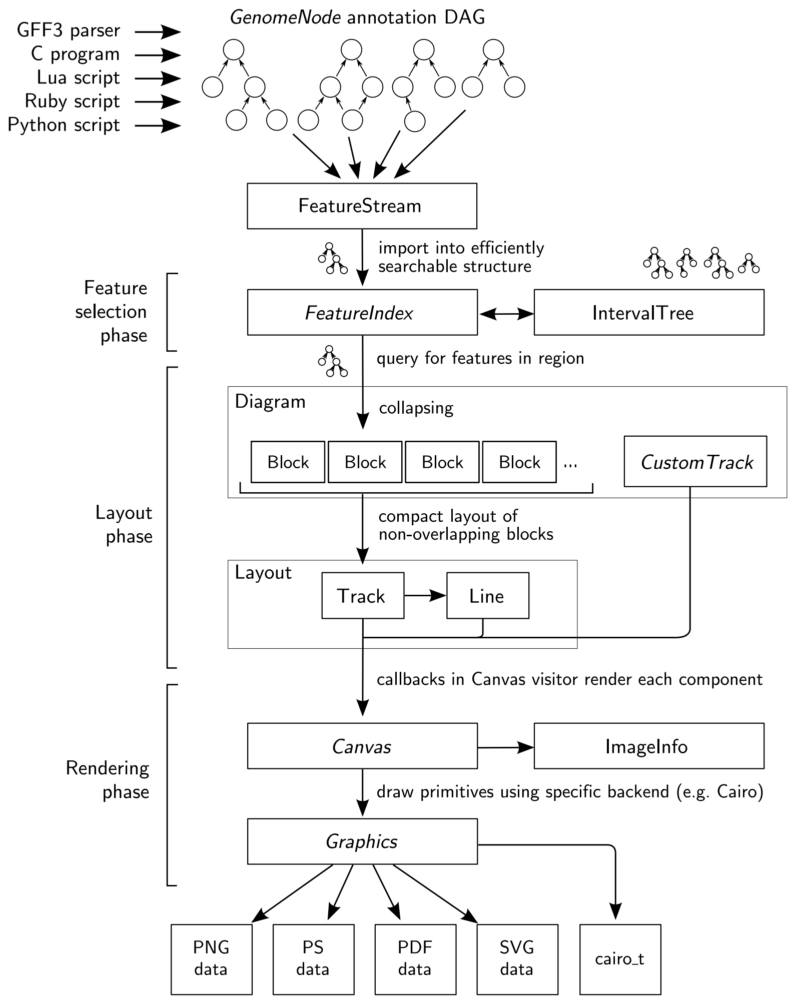


Figure 1.1: Schematic of the data flow through the classes involved in image creation.

users to develop their own graphical representations as plugins. The *Diagram* is then prepared for image output by calculating a compact *Layout* in which the *Block* objects in a track are distributed into *Line* objects, each containing non-overlapping blocks (see Fig. 1.3). The overall layout calculated this way tries to keep lines as compact as possible, minimising the amount of vertical space used. How new *Lines* are created depends on the chosen implementation of the *LineBreaker* interface, by default a *Block* is pushed into a new *Line* when either the *Block* or its caption overlaps with another one.

1.1.3 Phase 3: Rendering

In the final phase, the *Layout* object is used as a blueprint to create an image of a given type and size, considering user-defined options. The rendering process is invoked by calling the `sketch()` method of a *Layout* object. All rendering logic is implemented in classes implement-

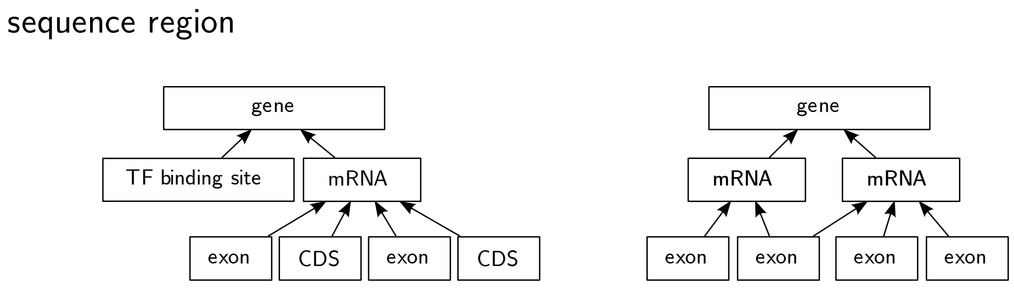


Figure 1.2: Example sequence region containing two genes in an annotation graph depicting the *part-of* relationships between their components.

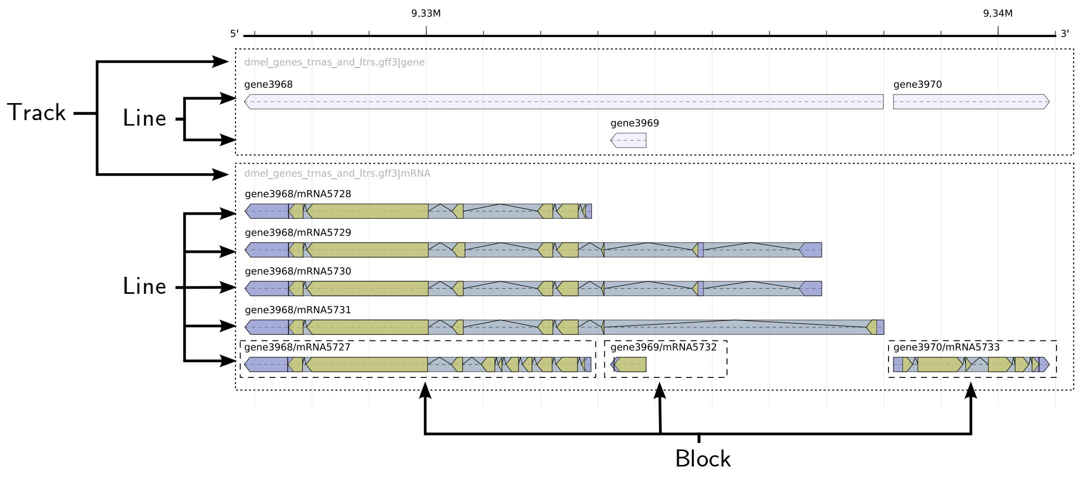


Figure 1.3: The components of the *Layout* class reflect sections of the resulting image.

ing the *Canvas* interface, whose methods are called during traversal of the *Layout* members. It encapsulates the state of a drawing and works independently of the chosen rendering back-end. Instead, rendering backend-dependent subclasses of *Canvas* are closely tied to a specific implementation of the *Graphics* interface, which provides methods to draw a number of primitives to a drawing surface abstraction. It wraps around the respective low-level graphics engine and allows for its easy extension or replacement. Currently, there is a *Graphics* implementation for the Cairo 2D graphics library (*GraphicsCairo*) and two *Canvas* subclasses providing access to the image file formats supported by Cairo (*CanvasCairoFile*) and to arbitrary Cairo contexts (*CanvasCairoContext*, which directly accesses a `cairo_t`). This class can be used, for example, to directly draw *AnnotationSketch* output in any graphical environment which is supported by Cairo (`http://www.cairographics.org/manual/cairo-surfaces.html`).

1.1.4 Collapsing

By default, *Lines* are grouped by the Sequence Ontology type associated with the top-level elements of their *Blocks*, resulting in one track per type. To obtain a shorter yet concise output,

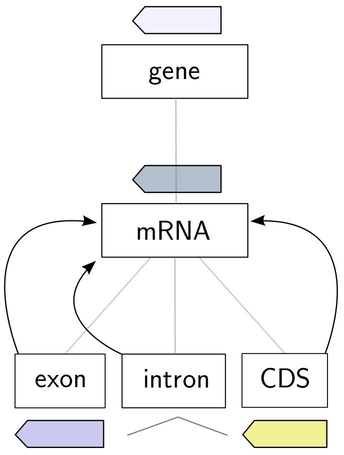


Figure 1.4: Schematic of the relationships between the *gene*, *mRNA*, *exon*, *intron* and *CDS* types and the colors of their representations in a diagram. The arrows illustrate how the relationships influence the collapsing process if collapsing is enabled for the *exon*, *intron* and *CDS* types. In this example, they will be drawn on top of their parent *mRNA* features.

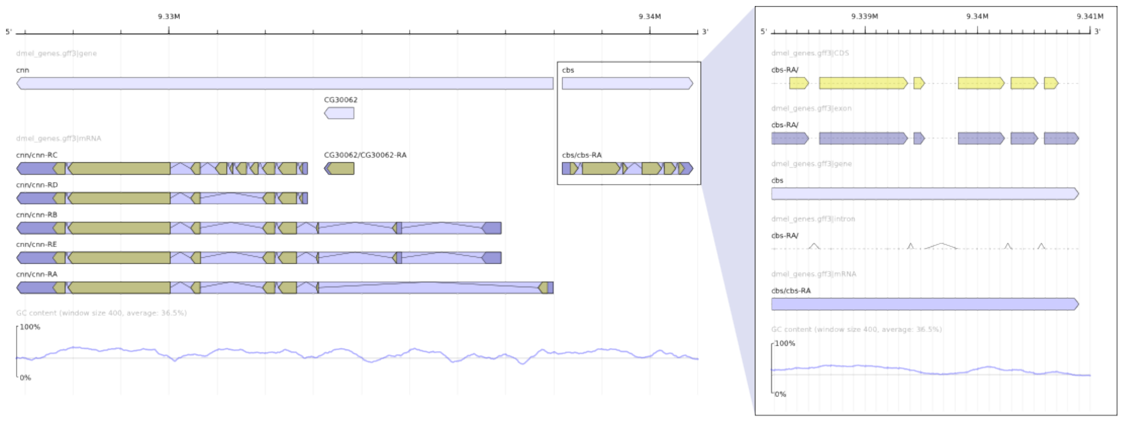
tracks for parent types in the feature graph can be enabled to contain all the features of their child types. The features with the given type are then drawn on top of their parent features (e.g. all *exon* and *intron* features are placed into their parent *mRNA* or *gene* track). This process is called *collapsing*. Collapsing can be enabled by setting the `collapse_to_parent` option for the respective child type to `true`, e.g. the following options:

```
config = {
  exon = {
    ...,
    collapse_to_parent = true,
    ...,
  },
  intron = {
    ...,
    collapse_to_parent = true,
    ...,
  },
  CDS = {
    ...,
    collapse_to_parent = true,
    ...,
  },
}
```

would lead to all features of the *exon*, *intron* and *CDS* types collapsing into the *mRNA* track (see Fig. 1.4 and 1.5).

1.1.5 Styles

The Lua scripting language is used to provide user-defined settings. Settings can be imported from a script that is executed when loaded, thus eliminating the need for another parser. The Lua configuration data are made accessible to C via the *Style* class. Configurable options in-



(a) collapsed view (b) uncollapsed view

Figure 1.5: Example image of the *cnn* and *cbs* genes from *Drosophila melanogaster* (Ensembl release 51, positions 9326816–9341000 on chromosome arm 2R) as drawn by *AnnotationSketch*. At the bottom, the calculated GC content of the respective sequence is drawn via a custom track attached to the diagram. (a) shows a collapsed view in which all *exon*, *intron* and *CDS* types are collapsed into their parent type's track. In contrast, (b) shows the *cbs* gene with all collapsing options set to `false`, resulting in each type being drawn in its own track.

clude assignment of display styles to each feature type, spacer and margin sizes, and collapsing parameters.
Instead of giving direct values, callback Lua functions can be used in some options to generate feature-dependent configuration settings at run-time. During layout and/or rendering, the *GenomeNode* object for the feature to be rendered is passed to the callback function which can then be evaluated and the appropriate type can be returned.
For example, setting the following options in the style file (or via the Lua bindings):

```lua
config = {
  ...,
  mRNA = {
    block_caption       = function(gn)
                            rng = gn:get_range()
                            return string.format("%s/%s (%dbp, %d exons)",
                                  gn:get_attribute("Parent"),
                                  gn:get_attribute("ID"),
                                  rng:get_end() - rng:get_start() + 1,
                                  #(gn:get_exons()))
                          end,
    ...
  },

  exon = {
    -- Color definitions
    fill                = function(gn)
                            if gn:get_score() then
                              aval = gn:get_score()*1.0
                            else
                              aval = 0.0
                            end
                            return {red=1.0, green=0.0, blue=0.0, alpha=aval}
```

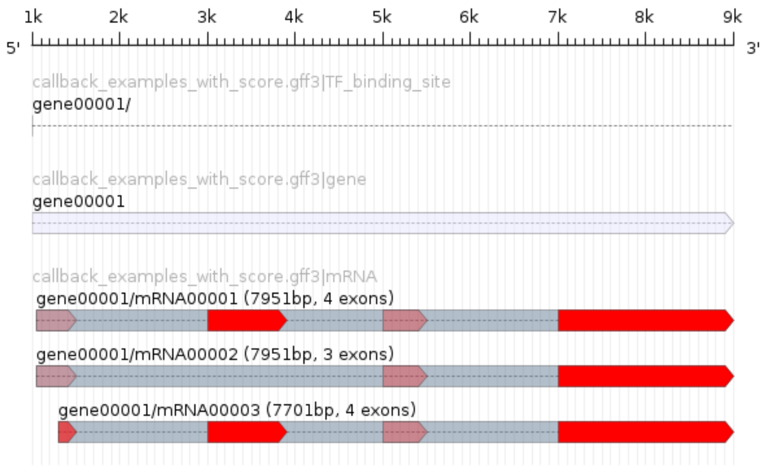

1k
2k
3k
4k
5k
6k
7k
8k
9k
5'
3'
callback_examples_with_score.gff3|TF_binding_site
gene00001/
callback_examples_with_score.gff3|gene
gene00001
callback_examples_with_score.gff3|mRNA
gene00001/mRNA00001 (7951bp, 4 exons)
gene00001/mRNA00002 (7951bp, 3 exons)
gene00001/mRNA00003 (7701bp, 4 exons)

Figure 1.6: Example rendering using callback functions to enable custom block captions and score-dependent shading of exon features.

```
24                                  end,
25      ...
26    },
27    ...
28  }
```

will result in a changed rendering (see Fig. 1.6). The `block_caption` function (line 4) overrides the default block naming scheme, allowing to set custom captions to each block depending on feature properties. Color definitions such as the `fill` setting (line 17) for a feature's fill color can also be individually styled using callbacks. In this case, the color intensity is shaded by the *exon* feature's score value (e.g. given in a GFF file).

1.2 The `gt sketch` tool

The *GenomeTools* `gt` executable provides a new tool which uses the *AnnotationSketch* library to create a drawing in PNG, PDF, PostScript or SVG format from GFF3 annotations. The annotations can be given by supplying one or more file names as command line arguments:

```
$ gt sketch output.png annotation.gff3
$
```

or by receiving GFF3 data via the standard input, here prepared by the `gt gff3` tool (here called with the `-addintrons` option to automatically add intron features between exons):

```
$ gt gff3 -addintrons annotation.gff3 | gt sketch output.png
$
```

The region to create a diagram for can be specified in detail by using the `-seqid`, `-start` and `-end` parameters. For example, if the *D. melanogaster* gene annotation is given in the `dmel_annotation.gff3` file, use

```
$ gt sketch -format pdf -seqid 2R -start 9326816 -end 9332879 output.pdf \
  dmel_annotation.gff3
$
```

to plot a graphical representation of the *cnn* and *cbs* gene region from the *FlyBase* default view in PDF format. The `-force` option can be used to force overwriting of an already existing output file. The `-pipe` option additionally allows passing the GFF3 input through the sketch tool via the standard output, allowing the intermediate visualisation of results in a longer pipeline of connected GFF3 tools. More command line options are available; their documentation can be viewed using the `-help` option.
If an input file is not plotted due to parsing errors, *GenomeTools* includes a strict GFF3 validator tool to check whether the input file is in valid GFF3 format. Simply run a command like the following:

```
$ gt gff3validator input_file.gff3
input is valid GFF3
$
```

This validator also allows one to check the SO types occurring in a GFF3 file against a given OBO ontology file. This checking can be enabled by specifying the file as an argument to the `-typecheck` option.
If the PDF, SVG and/or PostScript output format options are not available in the `gt` binary, the most likely cause is that PDF, SVG and/or PostScript support is disabled in your local *Cairo* headers and thus also not available in your local *Cairo* library. This issue is not directly related to *AnnotationSketch* and can be resolved by recompiling the *Cairo* library with the proper backend support enabled.

1.3 Dynamic track assignment

A special kind of function, called *track selector function*, can be used to customise the *AnnotationSketch* output by using arbitrary features of a block to assign blocks to tracks (and implicitly creating new tracks this way).

1.3.1 Default: Top level type decides track membership

By default, for each *Block* in a *Diagram*, its source filename and/or the type attribute of its top level element decides into which track the block is finally inserted during the layout phase. So by default, an annotation graph parsed from the GFF3 input file ‘example.gff3’ with *gene*, *mRNA* and *exon* type nodes will be rendered into two separate tracks (*exon*$\rightarrow$*mRNA* collapsing enabled, see Fig. 1.7):

- `example.gff3|gene`, and
- `example.gff3|mRNA`.

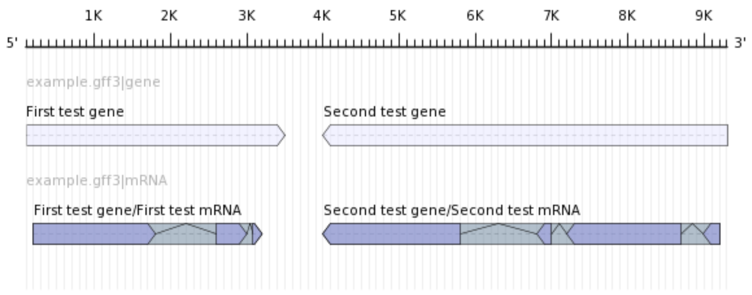


Figure 1.7: Default AnnotationSketch output for a simple GFF3 file with simple *exon→mRNA* collapsing.

We will call the second part (after the "|") of these track titles *track identifier strings* in the rest of this document.
While automatically determining tracks from the types actually present in the input annotations is convenient in many use cases, one could imagine cases in which more control about block handling may be desired. This leads to the question: How can one extract blocks with specific characteristics and assign them to a special track? The answer is simple: By overriding the default track identifier string, new tracks can be created and named on the fly as soon as a block satisfying user-defined rules is encountered.

1.3.2 Track selector functions

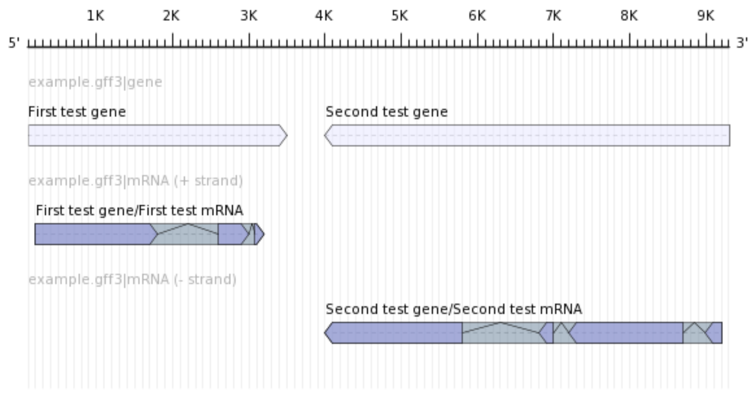
These rules take the form of *track selector functions*. Basically, a track selector function is a function which takes a block reference as an argument, and returns an appropriate track identifier string. For example, in Python the default track selector function would look like this:

```
def default_track_selector(block):
  return block.get_type()
```

This function simply returns a string representation of the type of a block's top level element, creating the tracks just like depicted in Fig. 1.7.
For a very simple example, let's assume that we want to create separate tracks for all mRNAs on the plus strand and for all mRNAs on the minus strand. The idea now is to change the strand identifier for blocks of the *mRNA* type to include the strand as additional information, thus creating different track identifiers for plus and minus strand features. In Python, this track selector function would construct a new string which contains both the type and the strand:

```
def strand_track_selector(block):
  if block.get_type() == "mRNA":
    return "%s (%s strand)" % (block.get_type(), block.get_strand())
  else:
    return block.get_type()
```

Using this track selector function would produce the desired result of separate tracks for the *mRNA* features for each strand (see Fig. 1.8).
A track selector function can be set for a *Diagram* object using the `diagram.set_track_selector_func()` method. In C, its argument is a pointer to a function of the signature



1K
2K
3K
4K
5K
6K
7K
8K
9K
5'
3'
example.gff3|gene
First test gene
Second test gene
example.gff3|mRNA (+ strand)
First test gene/First test mRNA
example.gff3|mRNA (- strand)
Second test gene/Second test mRNA

Figure 1.8: AnnotationSketch output with `strand_track_selector()` track selector function. This image now shows separate tracks for plus and minus strand features.

`void (*GtTrackSelectorFunc)(GtBlock*, GtStr*, void*)`

where arbitrary data can be passed via the third `void*` argument. The Python `set_track_selector_func()` method directly accepts a Python function as an argument, while the Ruby version takes a Proc object:

```
...
strand_track_selector = Proc.new { |block, data|
  "#{block.get_type} (#{block.get_strand} strand)"
}
...
diagram.set_track_selector_func(strand_track_selector)
...
```

Note that in Python and Ruby, it is also possible to reference data declared outside of the track selector function. For example, this can be used to filter blocks by pulling blocks whose description matches a pattern into a separate track:

```
...
interesting_genes = ["First test gene", "another gene"]

def filter_track_selector(block):
  if block.get_caption() in interesting_genes:
    return "interesting genes"
  else:
    return block.get_type()
...
diagram.set_track_selector_func(filter_track_selector)
...
```

This code results in the image shown in Fig. 1.9 :

1.4 Custom tracks

There are kinds of data which may be interesting to see together with annotation renderings, but that can not be expressed – or only in a complicated way – in GFF3 format. It may even be

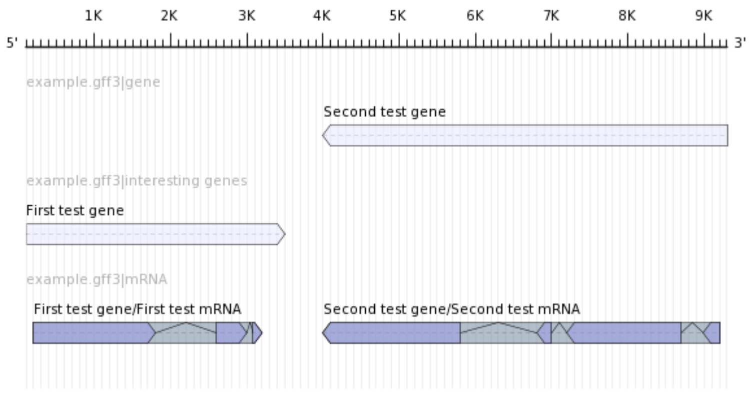

1K
2K
3K
4K
5K
6K
7K
8K
9K
5'
3'
example.gff3|gene
Second test gene
example.gff3|interesting genes
First test gene
example.gff3|mRNA
First test gene/First test mRNA
Second test gene/Second test mRNA

Figure 1.9: *AnnotationSketch* output with `filter_track_selector()` track selector function. This image now shows a separate track for features with a specific caption.

too difficult or counterintuitive to properly represent this data as typical *AnnotationSketch* box graphics. For example, this may be sequence data, numerical sequence analysis results, or other kinds of data which does not fit into the simple genomic feature scheme. For an example, see Fig. 1.10.
With *custom tracks*, *AnnotationSketch* provides a mechanism to use the internal drawing functionality to create user-defined output which can be tailored to fit this kind of data. A custom track looks just like a normal *AnnotationSketch* track, but is completely in control of the developer. While native *AnnotationSketch* primitives such as boxes can of course be used, the author of a custom track is not restricted to the layout algorithm and can draw anything anywhere (as long as it is provided by the *Graphics* class), taking arbitrary external data into account.

1.4.1 Anatomy of a custom track class

Simply put, custom tracks are classes which are derived from a *CustomTrack* base class and must implement a set of mandatory methods:

- `get_height()`: Returns the amount of vertical space (in pixels or points) the custom track will occupy in the final image. Must return a numeric value.
- `get_title()`: Returns a title for the custom track which is displayed at the top of the track. Note that, unlike a track identifier string e.g. produced by a track selector function, the string returned by this function is not prepended by a file name.
- `render(graphics, ypos, range, style, error)`: Performs the actual rendering operations. As parameters, this function receives
  - a *Graphics* object to draw on,
  - the vertical offset *ypos* of the drawing area assigned to the custom track,
  - the *Range* of the sequence positions for which annotations are currently displayed,

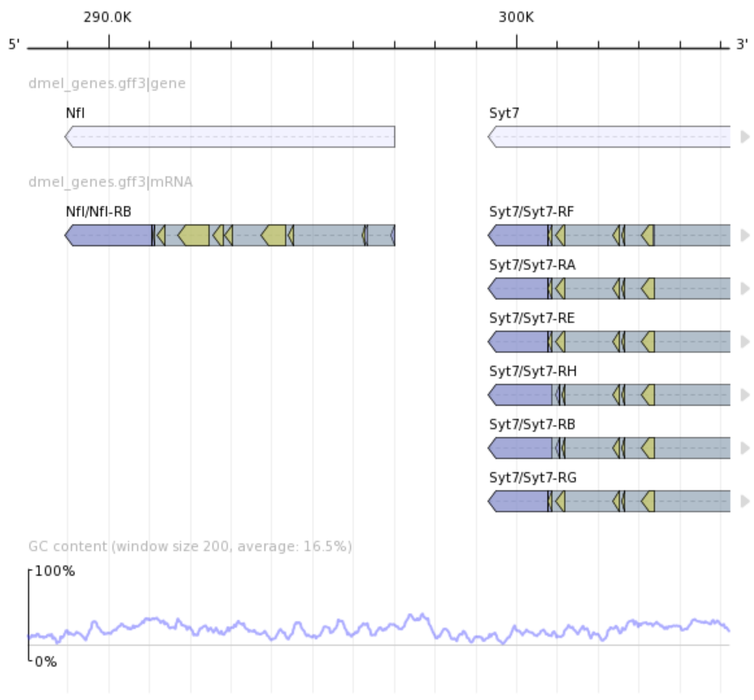


Figure 1.10: Example *AnnotationSketch* output with a custom track at the bottom, displaying the GC content over a window size of 200 bp.

- a *Style* object which can be used to obtain style information specific to this custom track, and
- an *Error* object which can be used to return an error message if the custom track needs to signal a problem.

The `render()` method must return 0 if drawing was successful, or a negative value if an error occurred.
Optionally, a `free()` method can be implemented if the subclass needs to clean up any private space allocated by itself. These methods are then called by the rendering code in AnnotationSketch when a Diagram containing a custom track is laid out and rendered. No other constraints apply on such a class besides that these methods are implemented (in the scripting language bindings, the parent classes' constructor must be called once).

1.4.2 Writing an example custom track

Let's suppose we are not satisfied with the display of single base features, such as transposable element insertion sites or SNPs. Instead of a single line denoting the feature location, we would like to have a small triangle pointing at the location. Suppose we also do not have this data in an annotation graph, so we cannot use the built-in rendering functions. It is straightforward to write

a small custom track class which does this for us. This tutorial uses Python code for simplicity, but the general approach is common to all supported languages.
First, we need to define a class inheriting from CustomTrack, call the parent constructor to register the functions and set instance variables for the triangle sidelength and a dictionary containing the feature positions and a description:

```
class CustomTrackInsertions(CustomTrack):
  def __init__(self, sidelength, data):
    super(CustomTrackInsertions, self).__init__()
    self.sidelength = sidelength
    self.data = data
```

We define the height to be 20 pixels:

```
  def get_height(self):
    return 20
```

As a track title, we set “Insertion site”:

```
  def get_title(self):
    return "Insertion site"
```

The rendering code then calculates the triangle coordinates and draws the respective lines:

```
  def render(self, graphics, ypos, rng, style, error):
    height = (self.sidelength*math.sqrt(3))/2
    margins = graphics.get_xmargins()
    red = Color(1, 0, 0, 0.7)
    for pos, desc in self.data.iteritems():
      drawpos = margins + (float(pos)-rng.start)/(rng.end-rng.start+1)         \
                  * (graphics.get_image_width()-2*margins)
      graphics.draw_line(drawpos-self.sidelength/2, ypos + height,              \
                         drawpos, ypos,                                         \
                         red, 1)
      graphics.draw_line(drawpos, ypos,                                         \
                         drawpos+self.sidelength/2, ypos + height,              \
                         red, 1)
      graphics.draw_line(drawpos-self.sidelength/2, ypos + height,              \
                         drawpos+self.sidelength/2, ypos + height,              \
                         red, 1)
      graphics.draw_text_centered(drawpos, ypos + height + 13, str(desc))
    return 0
```

For a Python custom track, that’s it! No more code is necessary for this very simple custom track. We can now instantiate this class and attach the instance to a *Diagram* object:

```
...
diagram = Diagram(feature_index, seqid, range, style)
...
ctt = CustomTrackInsertions(15, {2000:"foo", 4400:"bar", 8000:"baz"})
diagram.add_custom_track(ctt)
...
```

Running layout and drawing functions on this diagram then produces the desired image (see Fig. 1.11

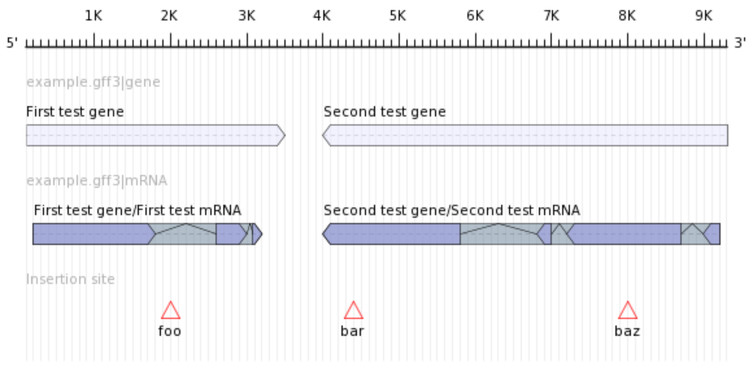


Figure 1.11: The example insertion site custom track (at the bottom), displaying three sample data points.

1.5 Examples

This section will show how to use the *AnnotationSketch* library in custom applications. As *AnnotationSketch* is distributed as a part of *GenomeTools*, its code is compiled into the `libgenometools.so` shared library. Please refer to the INSTALL file inside the *GenomeTools* distribution for installation instructions.
For a general idea about how to use the library, a simple implementation of the GFF3 validator is included in the source package (see `src/examples/gff3validator.c`) as an example showing how to create *GenomeTools*-based programs. In the same directory, there is also an appropriate `Makefile` to build and link this application against the installed shared library `libgenometools.so`.

1.5.1 Using AnnotationSketch to draw annotations from a file

The following code examples (in C and Lua) illustrate how to produce an image from a given GFF3 file using *AnnotationSketch*. The result is shown in Fig. 1.12. In essence, these code examples implement something like a simple version of the `gt sketch` tool from *GenomeTools* without most command-line options. The C-based examples mentioned below are compiled along with the *GenomeTools* library itself and available in the `bin/examples` directory.

C code

(See `src/examples/sketch_parsed.c` in the source distribution.)

```
#include "genometools.h"

static void handle_error(GtError *err)
{
  fprintf(stderr, "error: %s\n", gt_error_get(err));
  exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
```

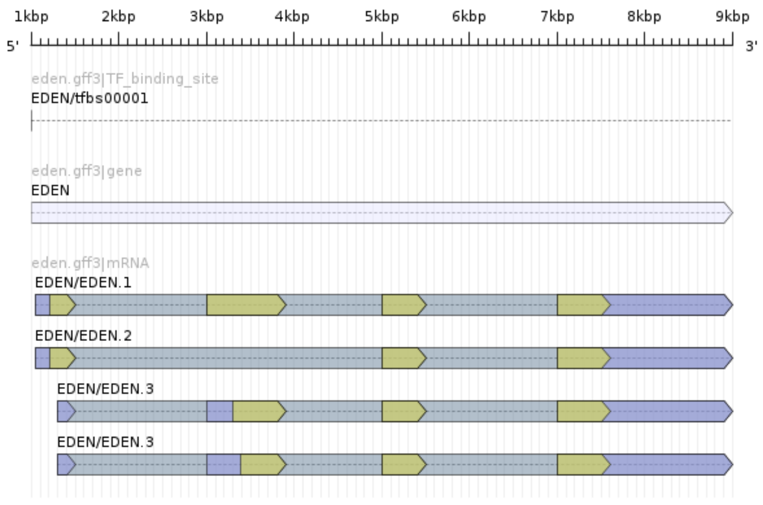


Figure 1.12: Example rendering of a GFF3 file with default style.

```
  const char *style_file, *png_file, *gff3_file;
  char *seqid;
  GtStyle *style;
  GtFeatureIndex *feature_index;
  GtRange range;
  GtDiagram *diagram;
  GtLayout *layout;
  GtCanvas *canvas;
  unsigned long height;
  GtError *err;

  if (argc != 4) {
    fprintf(stderr, "Usage: %s style_file PNG_file GFF3_file\n", argv[0]);
    return EXIT_FAILURE;
  }

  style_file = argv[1];
  png_file = argv[2];
  gff3_file = argv[3];

  /* initialize */
  gt_lib_init();

  /* create error object */
  err = gt_error_new();

  /* create style */
  if (!(style = gt_style_new(err)))
    handle_error(err);

  /* load style file */
  if (gt_style_load_file(style, style_file, err))
    handle_error(err);

  /* create feature index */
  feature_index = gt_feature_index_memory_new();
```

```c
  /* add GFF3 file to index */
  if (gt_feature_index_add_gff3file(feature_index, gff3_file, err))
    handle_error(err);

  /* create diagram for first sequence ID in feature index */
  if (!(seqid = gt_feature_index_get_first_seqid(feature_index, err))) {
    if (gt_error_is_set(err))
      handle_error(err);
  }
  if (gt_feature_index_get_range_for_seqid(feature_index, &range, seqid, err))
    handle_error(err);
  diagram = gt_diagram_new(feature_index, seqid, &range, style, err);
  gt_free(seqid);
  if (gt_error_is_set(err))
    handle_error(err);

  /* create layout with given width, determine resulting image height */
  layout = gt_layout_new(diagram, 600, style, err);
  if (!layout)
    handle_error(err);
  if (gt_layout_get_height(layout, &height, err))
    handle_error(err);

  /* create PNG canvas */
  canvas = gt_canvas_cairo_file_new(style, GT_GRAPHICS_PNG, 600, height,
                                    NULL, err);
  if (!canvas)
    handle_error(err);

  /* sketch layout on canvas */
  if (gt_layout_sketch(layout, canvas, err))
    handle_error(err);

  /* write canvas to file */
  if (gt_canvas_cairo_file_to_file((GtCanvasCairoFile*) canvas, png_file, err))
    handle_error(err);

  /* free */
  gt_canvas_delete(canvas);
  gt_layout_delete(layout);
  gt_diagram_delete(diagram);
  gt_feature_index_delete(feature_index);
  gt_style_delete(style);
  gt_error_delete(err);
  /* perform static data cleanup */
  gt_lib_clean();
  return EXIT_SUCCESS;
}
```

Lua code

(See gtscripts/sketch_parsed.lua in the source distribution. This example can be run by the command line gt gtscripts/sketch_parsed.lua <style_file> <PNG_file> <GFF3_file>)

```lua
function usage()
  io.stderr:write(string.format("Usage: %s Style_file PNG_file GFF3_file\n",
      arg[0]))
  io.stderr:write("Create PNG representation of GFF3 annotation file.\n")
  os.exit(1)
end
```

```
if #arg == 3 then
  style_file = arg[1]
  png_file   = arg[2]
  gff3_file  = arg[3]
else
  usage()
end

-- load style file
dofile(style_file)

-- create feature index
feature_index = gt.feature_index_memory_new()

-- add GFF3 file to index
feature_index:add_gff3file(gff3_file)

-- create diagram for first sequence ID in feature index
seqid = feature_index:get_first_seqid()
range = feature_index:get_range_for_seqid(seqid)
diagram = gt.diagram_new(feature_index, seqid, range)

-- create layout
layout = gt.layout_new(diagram, 600)
height = layout:get_height()

-- create canvas
canvas = gt.canvas_cairo_file_new_png(600, height, nil)

-- sketch layout on canvas
layout:sketch(canvas)

-- write canvas to file
canvas:to_file(png_file)
```

Ruby code

(See `gtruby/sketch_parsed.rb` in the source distribution.)

```
require 'gtruby'

if ARGV.size != 3 then
  STDERR.puts "Usage: #{$0} style_file PNG_file GFF3_file"
  STDERR.puts "Create PNG representation of GFF3 annotation file."
  exit(1)
end

(stylefile, pngfile, gff3file) = ARGV

# load style file
style = GT::Style.new()
style.load_file(stylefile)

# create feature index
feature_index = GT::FeatureIndexMemory.new()

# add GFF3 file to index
feature_index.add_gff3file(gff3file)
```

```
# create diagram for first sequence ID in feature index
seqid = feature_index.get_first_seqid()
range = feature_index.get_range_for_seqid(seqid)
diagram = GT::Diagram.from_index(feature_index, seqid, range, style)

# create layout for given width
layout = GT::Layout.new(diagram, 800, style)

# create canvas with given width and computed height
canvas = GT::CanvasCairoFile.new(style, 800, layout.get_height, nil)

# sketch layout on canvas
layout.sketch(canvas)

# write canvas to file
canvas.to_file(pngfile)
```

Python code

(See gtpython/sketch_parsed.py in the source distribution.)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from gt.annotationsketch import *
from gt.core.gtrange import Range
import sys

if __name__ == "__main__":
    if len(sys.argv) != 4:
        sys.stderr.write("Usage: " + (sys.argv)[0] +
                         " Style_file PNG_file GFF3_file\n")
        sys.stderr.write("Create PNG representation of GFF3 annotation file.")
        sys.exit(1)

    pngfile = (sys.argv)[2]

  # load style file

    style = Style()
    style.load_file((sys.argv)[1])

  # create feature index

    feature_index = FeatureIndexMemory()

  # add GFF3 file to index

    feature_index.add_gff3file((sys.argv)[3])

  # create diagram for first sequence ID in feature index

    seqid = feature_index.get_first_seqid()
    range = feature_index.get_range_for_seqid(seqid)
    diagram = Diagram.from_index(feature_index, seqid, range, style)

  # create layout

    layout = Layout(diagram, 600, style)
    height = layout.get_height()
```

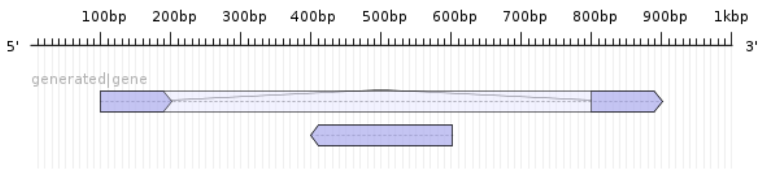


Figure 1.13: Example rendering of user-generated annotations with default style.

```
  # create canvas

    canvas = CanvasCairoFile(style, 600, height)

  # sketch layout on canvas

    layout.sketch(canvas)

  # write canvas to file

    canvas.to_file(pngfile)
```

1.5.2 Using AnnotationSketch to draw user-generated annotations

The following C code example illustrates how to produce an image from annotation graphs created by user code. The result is shown in Fig. 1.13.

C code

(See `src/examples/sketch_constructed.c` in the source distribution.)

```
#include "genometools.h"

static GtArray* create_example_features(void)
{
  GtArray *features;
  GtGenomeNode *gene, *exon, *intron; /* features */
  GtStr *seqid; /* holds the sequence id the features refer to */

  /* construct the example features */
  features = gt_array_new(sizeof (GtGenomeNode*));
  seqid = gt_str_new_cstr("chromosome_21");

  /* construct a gene on the forward strand with two exons */
  gene = gt_feature_node_new(seqid, "gene", 100, 900, GT_STRAND_FORWARD);
  exon = gt_feature_node_new(seqid, "exon", 100, 200, GT_STRAND_FORWARD);
  gt_feature_node_add_child((GtFeatureNode*) gene, (GtFeatureNode*) exon);
  intron = gt_feature_node_new(seqid, "intron", 201, 799, GT_STRAND_FORWARD);
  gt_feature_node_add_child((GtFeatureNode*) gene, (GtFeatureNode*) intron);
  exon = gt_feature_node_new(seqid, "exon", 800, 900, GT_STRAND_FORWARD);
  gt_feature_node_add_child((GtFeatureNode*) gene, (GtFeatureNode*) exon);

  /* store forward gene in feature array */
  gt_array_add(features, gene);
```

```c
  /* construct a single-exon gene on the reverse strand
     (within the intron of the forward strand gene) */
  gene = gt_feature_node_new(seqid, "gene", 400, 600, GT_STRAND_REVERSE);
  exon = gt_feature_node_new(seqid, "exon", 400, 600, GT_STRAND_REVERSE);
  gt_feature_node_add_child((GtFeatureNode*) gene, (GtFeatureNode*) exon);

  /* store reverse gene in feature array */
  gt_array_add(features, gene);

  /* free */
  gt_str_delete(seqid);

  return features;
}

static void handle_error(GtError *err)
{
  fprintf(stderr, "error writing canvas %s\n", gt_error_get(err));
  exit(EXIT_FAILURE);
}

static void draw_example_features(GtArray *features, const char *style_file,
                                  const char *output_file)
{
  GtRange range = { 1, 1000 }; /* the genomic range to draw */
  GtStyle *style;
  GtDiagram *diagram;
  GtLayout *layout;
  GtCanvas *canvas;
  unsigned long height;
  GtError *err = gt_error_new();

  /* create style */
  if (!(style = gt_style_new(err)))
    handle_error(err);

  /* load style file */
  if (gt_style_load_file(style, style_file, err))
    handle_error(err);

  /* create diagram */
  diagram = gt_diagram_new_from_array(features, &range, style);

  /* create layout with given width, determine resulting image height */
  layout = gt_layout_new(diagram, 600, style, err);
  if (!layout)
    handle_error(err);
  if (gt_layout_get_height(layout, &height, err))
    handle_error(err);

  /* create PNG canvas */
  canvas = gt_canvas_cairo_file_new(style, GT_GRAPHICS_PNG, 600, height,
                                    NULL, err);
  if (!canvas)
    handle_error(err);

  /* sketch layout on canvas */
  if (gt_layout_sketch(layout, canvas, err))
    handle_error(err);

  /* write canvas to file */
  if (gt_canvas_cairo_file_to_file((GtCanvasCairoFile*) canvas, output_file,
                                        err)) {
    handle_error(err);
  }

  /* free */
  gt_canvas_delete(canvas);
  gt_layout_delete(layout);
  gt_diagram_delete(diagram);
  gt_style_delete(style);
  gt_error_delete(err);
}

static void delete_example_features(GtArray *features)
{
  unsigned long i;
  for (i = 0; i < gt_array_size(features); i++)
    gt_genome_node_delete(*(GtGenomeNode**) gt_array_get(features, i));
  gt_array_delete(features);
}

int main(int argc, char *argv[])
{
  GtArray *features; /* stores the created example features */

  if (argc != 3) {
    fprintf(stderr, "Usage: %s style_file output_file\n", argv[0]);
    return EXIT_FAILURE;
  }

  gt_lib_init();

  features = create_example_features();

  draw_example_features(features, argv[1], argv[2]);

  delete_example_features(features);

  gt_lib_clean();
  return EXIT_SUCCESS;
}
```

Lua code

(See `gtscripts/sketch_constructed.lua` in the source distribution. This example can be run by the command line `gt gtscripts/sketch_constructed.lua <style_file> <PNG_file>`)

```
function usage()
  io.stderr:write(string.format("Usage: %s Style_file PNG_file\n", arg[0]))
  os.exit(1)
end

if #arg == 2 then
  style_file = arg[1]
  png_file   = arg[2]
else
  usage()
end

-- load style file
dofile(style_file)
```

```
-- construct the example features
seqid = "chromosome_21"
nodes = {}

-- construct a gene on the forward strand with two exons
gene   = gt.feature_node_new(seqid, "gene", 100, 900, "+")
exon   = gt.feature_node_new(seqid, "exon", 100, 200, "+")
gene:add_child(exon)
intron = gt.feature_node_new(seqid, "intron", 201, 799, "+")
gene:add_child(intron)
exon   = gt.feature_node_new(seqid, "exon", 800, 900, "+")
gene:add_child(exon)
nodes[1] = gene

-- construct a single-exon gene on the reverse strand
-- (within the intron of the forward strand gene)
reverse_gene = gt.feature_node_new(seqid, "gene", 400, 600, "-")
reverse_exon = gt.feature_node_new(seqid, "exon", 400, 600, "-")
reverse_gene:add_child(reverse_exon)
nodes[2] = reverse_gene

-- create diagram
diagram = gt.diagram_new_from_array(nodes, 1, 1000)
layout = gt.layout_new(diagram, 600)
height = layout:get_height()

-- create canvas
canvas = gt.canvas_cairo_file_new_png(600, height, nil)

-- sketch layout on canvas
layout:sketch(canvas)

-- write canvas to file
canvas:to_file(png_file)
```

Ruby code

(See gtruby/sketch_constructed.rb in the source distribution.)

```
require 'gtruby'

if ARGV.size != 2 then
  STDERR.puts "Usage: #{$0} style_file PNG_file"
  exit(1)
end

seqid = "chromosome_21"

# construct a gene on the forward strand with two exons
gene   = GT::FeatureNode.create(seqid, "gene", 100, 900, "+")
exon   = GT::FeatureNode.create(seqid, "exon", 100, 200, "+")
gene.add_child(exon)
intron = GT::FeatureNode.create(seqid, "intron", 201, 799, "+")
gene.add_child(intron)
exon   = GT::FeatureNode.create(seqid, "exon", 800, 900, "+")
gene.add_child(exon)

# construct a single-exon gene on the reverse strand
# (within the intron of the forward strand gene)
```

```
reverse_gene = GT::FeatureNode.create(seqid, "gene", 400, 600, "-")
reverse_exon = GT::FeatureNode.create(seqid, "exon", 400, 600, "-")
reverse_gene.add_child(reverse_exon)

pngfile = ARGV[1]

style = GT::Style.new()
style.load_file(ARGV[0])

rng = GT::Range.new(1, 1000)

diagram = GT::Diagram.from_array([gene, reverse_gene], rng, style)

layout = GT::Layout.new(diagram, 600, style)
canvas = GT::CanvasCairoFile.new(style, 600, layout.get_height, nil)
layout.sketch(canvas)

canvas.to_file(pngfile)
```

Python code

(See gtpython/sketch_constructed.py in the source distribution.)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from gt.core import *
from gt.extended import *
from gt.annotationsketch import *
from gt.annotationsketch.custom_track import CustomTrack
from gt.core.gtrange import Range
import sys

if __name__ == "__main__":
    if len(sys.argv) != 3:
        sys.stderr.write("Usage: " + (sys.argv)[0] +
                         " style_file PNG_file\n")
        sys.exit(1)

    seqid = "chromosome_21"
    nodes = []

  # construct a gene on the forward strand with two exons

    gene = FeatureNode.create_new(seqid, "gene", 100, 900, "+")
    exon = FeatureNode.create_new(seqid, "exon", 100, 200, "+")
    gene.add_child(exon)
    intron = FeatureNode.create_new(seqid, "intron", 201, 799, "+")
    gene.add_child(intron)
    exon = FeatureNode.create_new(seqid, "exon", 800, 900, "+")
    gene.add_child(exon)

  # construct a single-exon gene on the reverse strand
  # (within the intron of the forward strand gene)

    reverse_gene = FeatureNode.create_new(seqid, "gene", 400, 600, "-")
    reverse_exon = FeatureNode.create_new(seqid, "exon", 400, 600, "-")
    reverse_gene.add_child(reverse_exon)

    pngfile = (sys.argv)[2]
    style = Style()
    style.load_file((sys.argv)[1])

    diagram = Diagram.from_array([gene, reverse_gene], Range(1, 1000),
                                 style)

    layout = Layout(diagram, 600, style)
    height = layout.get_height()
    canvas = CanvasCairoFile(style, 600, height)
    layout.sketch(canvas)

    canvas.to_file(pngfile)
```

2 API Reference

Table of Classes

Table of Modules

2.1 Sole functions

`GtORFIterator* gt_orf_iterator_new(GtCodonIterator *ci, GtTranslator *translator)`

Return a new `GtORFIterator*` which detects ORFs.

`GtORFIteratorStatus gt_orf_iterator_next(GtORFIterator *orf_iterator, GtRange *orf_rng, unsigned int *orf_frame, GtError *err)`

Sets the values of `orf_rng.start`, `orf_rng.end` and `orf_frame` to the current reading position of `ci` if START/STOP AA is found. The frame in which the ORF is located is written to the position pointed to by `orf_frame`. This function returns one of three status codes: GT_ORF_ITERATOR_OK : an ORF was detected successfully(START/STOP AA pair), GT_ORF_ITERATOR_END : no ORF was detected because the end of the scan region has been reached, GT_ORF_ITERATOR_ERROR : no ORF was detected because an error occurred during sequence access. See `err` for details.

`void gt_orf_iterator_delete(GtORFIterator *orf_iterator)`

Delete `orf_iterator`.

`int gt_reverse_complement(char *dna_seq, unsigned long seqlen, GtError*)`

reverse `dna_seq` of length `seqlen` in place

`GtNodeStream* gt_feature_stream_new(GtNodeStream*, GtFeatureIndex*)`

create a FeatureStream which writes to GtFeatureIndex

2.2 Class GtAddIntronsStream

Implements the `GtNodeStream` interface. A `GtAddIntronsStream` inserts new feature nodes with type *intron* between existing feature nodes with type *exon*. This is a special case of the `GtInterFeatureStream`.

Methods

`GtNodeStream* gt_add_introns_stream_new(GtNodeStream *in_stream)`

Create a `GtAddIntronsStream*` which inserts feature nodes of type *intron* between feature nodes of type *exon* it retrieves from `in_stream` and returns them.

2.3 Class GtAlphabet

The following type is for storing alphabets.

Methods

`GtAlphabet* gt_alphabet_new_dna(void)`

Return a `GtAlphabet` object which represents a DNA alphabet.

`GtAlphabet* gt_alphabet_new_protein(void)`

Return a `GtAlphabet` object which represents a protein alphabet.

`GtAlphabet* gt_alphabet_new_empty(void)`

Return an empty `GtAlphabet` object.

`GtAlphabet* gt_alphabet_new_from_file(const char *filename, GtError *err)`

Return a `GtAlphabet` object, as read from an .al1 file specified by `filename` (i.e. no al1 suffix necessary).

`GtAlphabet* gt_alphabet_new_from_file_no_suffix(const char *filename, GtError *err)`

Return a `GtAlphabet` object, as read from a file specified by `filename`.

`GtAlphabet* gt_alphabet_new_from_string(const char *alphadef, unsigned long len, GtError *err)`

Return a `GtAlphabet` object, as read from a string of length `len` specified by `alphadef`.

`GtAlphabet* gt_alphabet_new_from_sequence(const GtStrArray *filenametab, GtError *err)`

Returns a new `GtAlphabet` object by scanning the sequence files in `filenametab` to determine whether they are DNA or protein sequences, and the appropriate alphabet will be used (see`gt_alphabet_guess()`). Returns NULL on error, see `err` for details.

`GtAlphabet* gt_alphabet_guess(const char *sequence, unsigned long seqlen)`

Try to guess which type the given `sequence` with `length` has (DNA or protein) and return an according `GtAlphabet*` object.

`GtAlphabet* gt_alphabet_clone(const GtAlphabet *alphabet)`

Return a clone of `alphabet`.

`GtAlphabet* gt_alphabet_ref(GtAlphabet *alphabet)`

Increase the reference count for `alphabet` and return it.

`void gt_alphabet_add_mapping(GtAlphabet *alphabet, const char *characters)`

Add the mapping of all given `characters` to the given `alphabet`. The first character is the result of subsequent `gt_alphabet_decode()` calls.

`void gt_alphabet_add_wildcard(GtAlphabet *alphabet, char wildcard)`

Add `wildcard` to the `alphabet`.

`const GtUchar* gt_alphabet_symbolmap(const GtAlphabet *alphabet)`

Returns the array of symbols from `alphabet` such that the index of the character equals its encoding.

`unsigned int gt_alphabet_num_of_chars(const GtAlphabet *alphabet)`

Returns number of characters in `alphabet` (excluding wildcards).

`unsigned int gt_alphabet_size(const GtAlphabet *alphabet)`

Returns number of characters in `alphabet` (including wildcards).

`const GtUchar* gt_alphabet_characters(const GtAlphabet *alphabet)`

Returns an array of the characters in `alphabet`.

`GtUchar gt_alphabet_wildcard_show(const GtAlphabet *alphabet)`

Returns the character used in `alphabet` to represent wildcards in output.

`unsigned int gt_alphabet_bits_per_symbol(const GtAlphabet *alphabet)`

Returns the required number of bits required to represent a symbol in `alphabet`.

`void gt_alphabet_output(const GtAlphabet *alphabet, FILE *fpout)`

Writes a representation of `alphabet` to the file pointer `fpout`.

`int gt_alphabet_to_file(const GtAlphabet *alphabet, const char *indexname, GtError *err)`

Writes a representation of `alphabet` to the .al1 output file as specified by `indexname` (i.e. without the .al1 suffix).

`void gt_alphabet_to_str(const GtAlphabet *alphabet, GtStr *dest)`

Writes a representation of `alphabet` to the `GtStr` as specified by `dest`.

`GtUchar gt_alphabet_pretty_symbol(const GtAlphabet *alphabet, unsigned int currentchar)`

Returns the printable character specified in `alphabet` for `currentchar`.

`void gt_alphabet_echo_pretty_symbol(const GtAlphabet *alphabet, FILE *fpout, GtUchar currentchar)`

Prints the printable character specified in `alphabet` for `currentchar` on `fpout`.

`bool gt_alphabet_is_protein(const GtAlphabet *alphabet)`

The following method checks if the given `alphabet` is the protein alphabet with the aminoacids A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y written in lower or upper case and returns `true`, if this is the case (`false` otherwise).

`bool gt_alphabet_is_dna(const GtAlphabet *alphabet)`

The following method checks if the given alphabet is the DNA alphabet with the bases A, C, G, T written in lower or upper case and returns `true`, if this is the case (`false` otherwise).

`bool gt_alphabet_valid_input(const GtAlphabet *alphabet, char c)`

Returns true if the character `c` is defined in `alphabet`.

`GtUchar gt_alphabet_encode(const GtAlphabet *alphabet, char c)`

Encode character `c` with given `alphabet`. Ensure that `c` is encodable with the given `alphabet`!

`char gt_alphabet_decode(const GtAlphabet *alphabet, GtUchar c)`

Decode character `c` with given `alphabet`.

`void gt_alphabet_encode_seq(const GtAlphabet *alphabet, GtUchar *out, const char *in, unsigned long length)`

Encode sequence `in` of given `length` with `alphabet` and store the result in `out`. `in` has to be encodable with the given `alphabet`!

`void gt_alphabet_decode_seq_to_fp(const GtAlphabet *alphabet, FILE *fpout, const GtUchar *src, unsigned long len)`

Suppose the string `src` of length `len` was transformed according to the `alphabet`. The following method shows each character in `src` as the printable character specified in the transformation. The output is written to the given file pointer `fpout`.

`void gt_alphabet_decode_seq_to_cstr(const GtAlphabet *alphabet, char *dest, const GtUchar *src, unsigned long len)`

Analog to `gt_alphabet_decode_seq_to_fp()` but writing the output to `dest`.

`GtStr* gt_alphabet_decode_seq_to_str(const GtAlphabet *alphabet, const GtUchar *src, unsigned long len)`

Analog to `gt_alphabet_decode_seq_to_fp()` writing the output to a new `GtStr`.

`void gt_alphabet_delete(GtAlphabet *alphabet)`

Decrease the reference count for `alphabet` or delete it, if this was the last reference.

2.4 Class GtAnnoDBSchema

The "GtAnnoDBSchema" interface for a database-backed abstract `GtFeatureIndex` factory

Methods

`GtFeatureIndex* gt_anno_db_schema_get_feature_index(GtAnnoDBSchema *schema, GtRDB *db, GtError *err)`

Returns a `GtFeatureIndex` object representing `GtRDB` object `db` interpreted as having schema `schema`. Returns NULL if an error occurred, `err` is set accordingly.

`void gt_anno_db_schema_delete(GtAnnoDBSchema *schema)`

Deletes `schema` and frees all associated memory.

2.5 Class GtArray

`GtArray` objects are generic arrays for elements of a certain size which grow on demand.

Methods

`GtArray* gt_array_new(size_t size_of_elem)`

Return a new `GtArray` object whose elements have the size `size_of_elem`.

`GtArray* gt_array_ref(GtArray *array)`

Increase the reference count for `array` and return it. If `array` is `NULL`, `NULL` is returned without any side effects.

`GtArray* gt_array_clone(const GtArray *array)`

Return a clone of `array`.

`void* gt_array_get(const GtArray *array, unsigned long index)`

Return pointer to element number `index` of `array`. `index` has to be smaller than `gt_array_size(array)`.

`void* gt_array_get_first(const GtArray *array)`

Return pointer to first element of `array`.

`void* gt_array_get_last(const GtArray *array)`

Return pointer to last element of `array`.

`void* gt_array_pop(GtArray *array)`

Return pointer to last element of `array` and remove it from `array`.

`void* gt_array_get_space(const GtArray *array)`

Return pointer to the internal space of `array` where the elements are stored.

`#define gt_array_add(array, elem)`

Add element `elem` to `array`. The size of `elem` must equal the given element size when the `array` was created and is determined automatically with the `sizeof` operator.

`void gt_array_add_elem(GtArray *array, void *elem, size_t size_of_elem)`

Add element `elem` with size `size_of_elem` to `array`. `size_of_elem` must equal the given element size when the `array` was created. Usually, this method is not used directly and the macro `gt_array_add()` is used instead.

`void gt_array_add_array(GtArray *dest, const GtArray *src)`

Add all elements of array `src` to the array `dest`. The element sizes of both arrays must be equal.

`void gt_array_rem(GtArray *array, unsigned long index)`

Remove element with number `index` from `array` in O(`gt_array_size(array)`) time. `index` has to be smaller than `gt_array_size(array)`.

`void gt_array_rem_span(GtArray *array, unsigned long frompos, unsigned long topos)`

Remove elements starting with number `frompos` up to (and including) `topos` from `array` in O(`gt_array_size(array)`) time. `frompos` has to be smaller or equal than `topos` and both have to be smaller than `gt_array_size(array)`.

`void gt_array_reverse(GtArray *array)`

Reverse the order of the elements in `array`.

`void gt_array_set_size(GtArray *array, unsigned long size)`

Set the size of `array` to `size`. `size` must be smaller or equal than `gt_array_size(array)`.

`void gt_array_reset(GtArray *array)`

Reset the `array`. That is, afterwards the array has size 0.

`size_t gt_array_elem_size(const GtArray *array)`

Return the size of the elements stored in `array`.

`unsigned long gt_array_size(const GtArray *array)`

Return the number of elements in `array`. If `array` equals `NULL`, 0 is returned.

`void gt_array_sort(GtArray *array, GtCompare compar)`

Sort `array` with the given compare function `compar`.

`void gt_array_sort_stable(GtArray *array, GtCompare compar)`

Sort `array` in a stable way with the given compare function `compar`.

`void gt_array_sort_with_data(GtArray *array, GtCompareWithData compar, void *data)`

Sort `array` with the given compare function `compar`. Passes a pointer with userdata `data` to `compar`.

`void gt_array_sort_stable_with_data(GtArray *array, GtCompareWithData compar, void *data)`

Sort `array` in a stable way with the given compare function `compar`. Passes a pointer with userdata `data` to `compar`.

`int gt_array_cmp(const GtArray *array_a, const GtArray *array_b)`

Compare the content of `array_a` with the content of `array_b`. `array_a` and `array_b` must have the same `gt_array_size()` and `gt_array_elem_size()`.

`void gt_array_delete(GtArray *array)`

Decrease the reference count for `array` or delete it, if this was the last reference.

2.6 Class GtArrayOutStream

`GtNodeStream* gt_array_out_stream_new(GtNodeStream *in_stream, GtArray *nodes, GtError *err)`

Implements the `GtNodeStream` interface. `GtArrayOutStream` takes `GtGenomeNodes` of tpe `GtFeatureNode` from `in_stream` and adds them to the array `nodes`. This stream can be used to obtain nodes for processing outside the usual stream flow

2.7 Class GtBEDInStream

Implements the `GtNodeStream` interface. A `GtBEDInStream` allows one to parse a BED file and return it as a stream of `GtGenomeNode` objects.

Methods

`GtNodeStream* gt_bed_in_stream_new(const char *filename)`

Return a `GtBEDInStream` object which subsequently reads the BED file with the given `filename`. If `filename` equals `NULL`, the BED data is read from `stdin`.

`void gt_bed_in_stream_set_feature_type(GtBEDInStream *bed_in_stream, const char *type)`

Create BED features parsed by `bed_in_stream` with given `type` (instead of the default "BED_feature").

`void gt_bed_in_stream_set_thick_feature_type(GtBEDInStream *bed_in_stream, const char *type)`

Create thick BED features parsed by `bed_in_stream` with given `type` (instead of the default "BED_thick_feature").

`void gt_bed_in_stream_set_block_type(GtBEDInStream *bed_in_stream, const char *type)`

Create BED blocks parsed by `bed_in_stream` with given `type` (instead of the default "BED_block").

2.8 Class GtBittab

Implements arbitrary-length bit arrays and various operations on them.

Methods

`GtBittab* gt_bittab_new(unsigned long num_of_bits)`

Return a new `GtBittab` of length `num_of_bits`, initialised to 0.

`void gt_bittab_set_bit(GtBittab *bittab, unsigned long i)`

Set bit `i` in `bittab` to 1.

`void gt_bittab_unset_bit(GtBittab *bittab, unsigned long i)`

Set bit `i` in `bittab` to 0.

`void gt_bittab_complement(GtBittab *bittab_a, const GtBittab *bittab_b)`

Set `bittab_a` to be the complement of `bittab_b`.

`void gt_bittab_equal(GtBittab *bittab_a, const GtBittab *bittab_b)`

Set `bittab_a` to be equal to `bittab_b`.

`void gt_bittab_and(GtBittab *bittab_a, const GtBittab *bittab_b, const GtBittab *bittab_c)`

Set `bittab_a` to be the bitwise AND of `bittab_b` and `bittab_c`.

`void gt_bittab_or(GtBittab *bittab_a, const GtBittab *bittab_b, const GtBittab *bittab_c)`

Set `bittab_a` to be the bitwise OR of `bittab_b` and `bittab_c`.

`void gt_bittab_nand(GtBittab *bittab_a, const GtBittab *bittab_b, const GtBittab *bittab_c)`

Set `bittab_a` to be `bittab_b` NAND `bittab_c`.

`void gt_bittab_and_equal(GtBittab *bittab_a, const GtBittab *bittab_b)`

Set `bittab_a` to be the bitwise AND of `bittab_a` and `bittab_b`.

`void gt_bittab_or_equal(GtBittab *bittab_a, const GtBittab *bittab_b)`

Set `bittab_a` to be the bitwise OR of `bittab_a` and `bittab_b`.

`void gt_bittab_shift_left_equal(GtBittab *bittab)`

Shift `bittab` by one position to the left.

`void gt_bittab_shift_right_equal(GtBittab *bittab)`

Shift `bittab` by one position to the right.

`void gt_bittab_unset(GtBittab *bittab)`

Set all bits in `bittab` to 0.

`void gt_bittab_show(const GtBittab *bittab, FILE *fp)`

Output a representation of `bittab` to `fp`.

`void gt_bittab_get_all_bitnums(const GtBittab *bittab, GtArray *array)`

Fill `array` with the indices of all set bits in `bittab`.

`bool gt_bittab_bit_is_set(const GtBittab *bittab, unsigned long i)`

Return `true` if bit `i` is set in `bittab`.

`bool gt_bittab_cmp(const GtBittab *bittab_a, const GtBittab *bittab_b)`

Return `true` if `bittab_a` and `bittab_b` are identical.

`unsigned long gt_bittab_get_first_bitnum(const GtBittab *bittab)`

Return the index of the first set bit in `bittab`.

`unsigned long gt_bittab_get_last_bitnum(const GtBittab *bittab)`

Return the index of the last set bit in `bittab`.

`unsigned long gt_bittab_get_next_bitnum(const GtBittab *bittab, unsigned long i)`

Return the index of the next set bit in `bittab` with an index greater than `i`.

`unsigned long gt_bittab_count_set_bits(const GtBittab *bittab)`

Return the number of set bits in `bittab`.

`unsigned long gt_bittab_size(GtBittab *bittab)`

Return the total number of bits of `bittab`.

`void gt_bittab_delete(GtBittab *bittab)`

Delete `bittab`.

2.9 Class GtBlock

The `GtBlock` class represents a portion of screen space which relates to a specific "top-level" feature (and maybe its collapsed child features). It is the smallest layoutable unit in AnnotationSketch and has a caption (which may be displayed above the block rendering).

Methods

`GtBlock* gt_block_new(void)`

Creates a new `GtBlock` object.

`GtBlock* gt_block_ref(GtBlock*)`

Increases the reference count.

`GtBlock* gt_block_new_from_node(GtFeatureNode *node)`

Create a new GtBlock object, setting block parameters (such as strand, range) from a given `node` template.

`GtRange gt_block_get_range(const GtBlock*)`

Returns the base range of the `GtBlock`'s top level element.

`GtRange* gt_block_get_range_ptr(const GtBlock *block)`

Returns a pointer to the base range of the `GtBlock`'s top level element.

`bool gt_block_has_only_one_fullsize_element(const GtBlock*)`

Checks whether a `GtBlock` is occupied completely by a single element.

`void gt_block_merge(GtBlock*, GtBlock*)`

Merges the contents of two `GtBlocks` into the first one.

`GtBlock* gt_block_clone(GtBlock*)`

Returns an independent copy of a `GtBlock`.

`void gt_block_set_caption_visibility(GtBlock*, bool)`

Set whether a block caption should be displayed or not.

`bool gt_block_caption_is_visible(const GtBlock*)`

Returns whether a block caption should be displayed or not.

`void gt_block_set_caption(GtBlock*, GtStr *caption)`

Sets the `GtBlock`'s caption to `caption`.

`GtStr* gt_block_get_caption(const GtBlock*)`

Returns the `GtBlock`'s caption.

`void gt_block_set_strand(GtBlock*, GtStrand strand)`

Sets the `GtBlock`'s strand to `strand`.

`GtStrand gt_block_get_strand(const GtBlock*)`

Returns the `GtBlock`'s strand.

`GtFeatureNode* gt_block_get_top_level_feature(const GtBlock*)`

Returns the `GtBlock`'s top level feature as a `GtFeatureNode` object.

`unsigned long gt_block_get_size(const GtBlock*)`

Returns the number of elements in the `GtBlock`.

`const char* gt_block_get_type(const GtBlock*)`

Returns the feature type of the `GtBlock`.

`void gt_block_delete(GtBlock*)`

Deletes a `GtBlock`.

2.10 Class GtCDSStream

Implements the `GtNodeStream` interface. A `GtCDSStream` determines the coding sequence (CDS) for sequences determined by feature nodes of type *exon* and adds them as feature nodes of type *CDS*.

Methods

`GtNodeStream* gt_cds_stream_new(GtNodeStream *in_stream, GtRegionMapping *region_mapping, unsigned int minorflen, const char *source, bool start_codon, bool final_stop_codon, bool generic_star_codons)`

Create a `GtCDSStream*` which determines the coding sequence (CDS) for sequences determined by feature nodes of type *exon* it retrieves from `in_stream`, adds them as feature nodes of type *CDS* and returns all nodes. `region_mapping` is used to map the sequence IDs of the feature nodes to the regions of the actual sequences. `minorflen` is the minimum length an ORF must have in order to be added. The CDS features are created with the given `source`. If `start_codon` equals `true` an ORF must begin with a start codon, otherwise it can start at any position. If `final_stop_codon` equals `true` the final ORF must end with a stop codon. If `generic_start_codons` equals `true`, the start codons of the standard translation scheme are used as start codons (otherwise the amino acid 'M' is regarded as a start codon).

2.11 Class GtCSAStream

Implements the `GtNodeStream` interface. A `GtCSAStream` takes spliced alignments and transforms them into consensus spliced alignments.

Methods

`GtNodeStream* gt_csa_stream_new(GtNodeStream *in_stream, unsigned long join_length)`

Create a `GtCSAStream*` which takes spliced alignments from its `in_stream` (which are at most `join_length` many bases apart), transforms them into consensus spliced alignments, and returns them.

2.12 Class GtCanvas

The `GtCanvas` class is an abstraction of a stateful drawing surface. Constructors must be implemented in subclasses as different arguments are required for drawing to specific graphics back-ends.

Methods

`unsigned long gt_canvas_get_height(GtCanvas *canvas)`

Returns the height of the given `canvas`.

`void gt_canvas_delete(GtCanvas *canvas)`

Delete the given `canvas`.

2.13 Class GtCanvasCairoContext

Implements the `GtCanvas` interface using a Cairo context (`cairo_t`) as input. This Canvas uses the `GtGraphicsCairo` class.
Drawing to a `cairo_t` allows the use of the *AnnotationSketch* engine in any Cairo-based graphical application.

Methods

`GtCanvas* gt_canvas_cairo_context_new(GtStyle *style, cairo_t *context, double offsetpos, unsigned long width, unsigned long height, GtImageInfo *image_info, GtError *err)`

Create a new `GtCanvas` object tied to the cairo_t `context`, `width` and `height` using the given `style`. The optional `image_info` is filled when the created Canvas object is used to render a `GtDiagram` object. `offsetpos` determines where to start drawing on the surface.

2.14 Class GtCanvasCairoFile

Implements the `GtCanvas` interface. This Canvas uses the `GtGraphicsCairo` class.

Methods

`GtCanvas* gt_canvas_cairo_file_new(GtStyle *style, GtGraphicsOutType output_type, unsigned long width, unsigned long height, GtImageInfo *image_info, GtError *err)`

Create a new `GtCanvasCairoFile` object with given `output_type` and `width` using the configuration given in `style`. The optional `image_info` is filled when the created object is used to render a `GtDiagram` object. Possible `GtGraphicsOutType` values are `GRAPHICS_PNG`, `GRAPHICS_PS`, `GRAPHICS_PDF` and `GRAPHICS_SVG`. Dependent on the local Cairo installation, not all of them may be available.

`int gt_canvas_cairo_file_to_file(GtCanvasCairoFile *canvas, const char *filename, GtError *err)`

Write rendered `canvas` to the file with name `filename`. If this method returns a value other than 0, check `err` for an error message.

`int gt_canvas_cairo_file_to_stream(GtCanvasCairoFile *canvas, GtStr *stream)`

Append rendered `canvas` image data to given `stream`.

2.15 Class GtCodonIterator

the "codon iterator" interface

Methods

`unsigned long gt_codon_iterator_current_position(GtCodonIterator *ci)`

Return the current reading offset of `ci`, starting from the position in the sequence given at iterator instantiation time.

`unsigned long gt_codon_iterator_length(GtCodonIterator *ci)`

Return the length of the substring to scan, given at instantiation time.

`void gt_codon_iterator_rewind(GtCodonIterator *ci)`

Rewind the iterator to point again to the position in the sequence given at iterator instantiation time.

`GtCodonIteratorStatus gt_codon_iterator_next(GtCodonIterator *ci, char *n1, char *n2, char *n3, unsigned int *frame, GtError *err)`

Sets the values of `n1`, `n2` and `n3` to the codon beginning at the current reading position of `ci` and then advances the reading position by one. The current reading frame shift (0, 1 or 2) is for the current codon is written to the position pointed to by `frame`. This function returns one of three status codes: GT_CODON_ITERATOR_OK : a codon was read successfully, GT_CODON_ITERATOR_END : no codon was read because the end of the scan region has been reached, GT_CODON_ITERATOR_ERROR : no codon was read because an error occurred during sequence access. See `err` for details.

`void gt_codon_iterator_delete(GtCodonIterator *ci)`

Delete `ci`.

2.16 Class GtColor

The `GtColor` class holds a RGB color definition.

Methods

`GtColor* gt_color_new(double red, double green, double blue, double alpha)`

Create a new `GtColor` object with the color given by the `red`, `green`, and `blue` arguments. The value for each color channel must be between 0 and 1.

`void gt_color_set(GtColor *color, double red, double green, double blue, double alpha)`

Change the color of the `color` object to the color given by the `red`, `green`, and `blue` arguments. The value for each color channel must be between 0 and 1.

`bool gt_color_equals(const GtColor *c1, const GtColor *c2)`

Returns `true` if the colors `c1` and `c2` are equal.

`void gt_color_delete(GtColor *color)`

Delete the `color` object.

2.17 Class GtCommentNode

Implements the `GtGenomeNode` interface. Comment nodes correspond to comment lines in GFF3 files (i.e., lines which start with a single “#”).

Methods

`GtGenomeNode* gt_comment_node_new(const char *comment)`

Return a new `GtCommentNode` object representing a `comment`. Please note that the single leading “#” which denotes comment lines in GFF3 files should not be part of `comment`.

`const char* gt_comment_node_get_comment(const GtCommentNode *comment_node)`

Return the comment stored in `comment_node`.

2.18 Class GtCstrTable

Implements a table of C strings.

Methods

`GtCstrTable* gt_cstr_table_new(void)`
: Return a new `GtCstrTable` object.

`void gt_cstr_table_add(GtCstrTable *table, const char *cstr)`
: Add `cstr` to `table`. `table` must not already contain `cstr`!

`const char* gt_cstr_table_get(const GtCstrTable *table, const char *cstr)`
: If a C string equal to `cstr` is contained in `table`, it is returned. Otherwise `NULL` is returned.

`GtStrArray* gt_cstr_table_get_all(const GtCstrTable *table)`
: Return a `GtStrArray*` which contains all `cstr`s added to `table` in alphabetical order. The caller is responsible to free it!

`void gt_cstr_table_remove(GtCstrTable *table, const char *cstr)`
: Remove `cstr` from `table`.

`void gt_cstr_table_reset(GtCstrTable *table)`
: Reset `table` (that is, remove all contained C strings).

`void gt_cstr_table_delete(GtCstrTable *table)`
: Delete C string `table`.

2.19 Class GtCustomTrack

The `GtCustomTrack` interface allows the `GtCanvas` to call user-defined drawing functions on a `GtGraphics` object. Please refer to the specific implementations' documentation for more information on a particular custom track.

Methods

`GtCustomTrack* gt_custom_track_ref(GtCustomTrack *ctrack)`
: Increase the reference count for `ctrack`.

`void gt_custom_track_delete(GtCustomTrack *ctrack)`
: Delete the given `ctrack`.

2.20 Class GtCustomTrackGcContent

Implements the `GtCustomTrack` interface. This custom track draws a plot of the GC content of a given sequence in the displayed range. As a window size for GC content calculation, `windowsize` is used.

Methods

`GtCustomTrack* gt_custom_track_gc_content_new(const char *seq, unsigned long seqlen, unsigned long windowsize, unsigned long height, double avg, bool show_scale)`

Creates a new `GtCustomTrackGcContent` for sequence `seq` with length `seqlen` of height `height` with windowsize `windowsize`. A horizontal line is drawn for the percentage value `avg`, with `avg` between 0 and 1. If `show_scale` is set to true, then a vertical scale rule is drawn at the left end of the curve.

2.21 Class GtCustomTrackScriptWrapper

Implements the `GtCustomTrack` interface. This custom track is only used to store pointers to external callbacks, e.g. written in a scripting language. This class does not store any state, relying on the developer of the external custom track class to do so.

Methods

`GtCustomTrack* gt_custom_track_script_wrapper_new(GtCtScriptRenderFunc render_func, GtCtScriptGetHeightFunc get_height_func, GtCtScriptGetTitleFunc get_title_func, GtCtScriptFreeFunc free_func)`

Creates a new `GtCustomTrackScriptWrapper` object.

2.22 Class GtDiagram

The `GtDiagram` class acts as a representation of a sequence annotation diagram independent of any output format. Besides annotation features as annotation graphs, it can contain one or more custom tracks. A individual graphical representation of the `GtDiagram` contents is created by creating a `GtLayout` object using the `GtDiagram` and then calling `gt_layout_sketch()` with an appropriate `GtCanvas` object.

Methods

`GtDiagram* gt_diagram_new(GtFeatureIndex *feature_index, const char *seqid, const GtRange *range, GtStyle *style, GtError*)`

Create a new `GtDiagram` object representing the feature nodes in `feature_index` in region `seqid` overlapping with `range`. The `GtStyle` object `style` will be used to determine collapsing options during the layout process.

`GtDiagram* gt_diagram_new_from_array(GtArray *features, const GtRange *range, GtStyle *style)`

Create a new `GtDiagram` object representing the feature nodes in `features`. The features must overlap with `range`. The `GtStyle` object `style` will be used to determine collapsing options during the layout process.

`GtRange gt_diagram_get_range(const GtDiagram *diagram)`

Returns the sequence position range represented by the `diagram`.

`void gt_diagram_set_track_selector_func(GtDiagram*, GtTrackSelectorFunc, void*)`

Assigns a GtTrackSelectorFunc to use to assign blocks to tracks. If none is set, or set to NULL, then track types are used as track keys (default behavior).

`void gt_diagram_reset_track_selector_func(GtDiagram *diagram)`

Resets the track selection behavior of this `GtDiagram` back to the default.

`void gt_diagram_add_custom_track(GtDiagram*, GtCustomTrack*)`

Registers a new custom track in the diagram.

`void gt_diagram_delete(GtDiagram*)`

Delete the `diagram` and all its components.

2.23 Class GtDlist

A double-linked list which is sorted according to a `GtCompare` compare function (`qsort(3)`-like, only if one was supplied to the constructor).

Methods

`GtDlist* gt_dlist_new(GtCompare compar)`

Return a new `GtDlist` object sorted according to `compar` function. If `compar` equals `NULL`, no sorting is enforced.

`GtDlistelem* gt_dlist_first(const GtDlist *dlist)`

Return the first `GtDlistelem` object in `dlist`.

`GtDlistelem* gt_dlist_last(const GtDlist *dlist)`

Return the last `GtDlistelem` object in `dlist`.

`GtDlistelem* gt_dlist_find(const GtDlist *dlist, void *data)`

Return the first `GtDlistelem` object in `dlist` which contains data identical to `data`. Takes O(n) time.

`unsigned long gt_dlist_size(const GtDlist *dlist)`

Return the number of `GtDlistelem` objects in `dlist`.

`void gt_dlist_add(GtDlist *dlist, void *data)`

Add a new `GtDlistelem` object containing `data` to `dlist`. Usually O(n), but O(1) if data is added in sorted order.

`void gt_dlist_remove(GtDlist *dlist, GtDlistelem *dlistelem)`

Remove `dlistelem` from `dlist` and free it.

`int gt_dlist_example(GtError *err)`

Example for usage of the `GtDlist` class.

`void gt_dlist_delete(GtDlist *dlist)`

Delete `dlist`.

2.24 Class GtDlistelem

`GtDlistelem* gt_dlistelem_next(const GtDlistelem *dlistelem)`

Return the successor of `dlistelem`, or `NULL` if the element is the last one in the `GtDlist`.

`GtDlistelem* gt_dlistelem_previous(const GtDlistelem *dlistelem)`

Return the predecessor of `dlistelem`, or `NULL` if the element is the first one in the `GtDlist`.

`void* gt_dlistelem_get_data(const GtDlistelem *dlistelem)`

Return the data pointer attached to `dlistelem`.

2.25 Class GtEOFNode

Implements the `GtGenomeNode` interface. EOF nodes mark the barrier between separate input files in an GFF3 stream.

Methods

`GtGenomeNode* gt_eof_node_new(void)`

Create a new `GtEOFNode*` representing an EOF marker.

2.26 Class GtEncseq

The `GtEncseq` class represents a concatenated collection of sequences from one or more input files in a bit-compressed encoding. It is stored in a number of `mmap()`able files, depending on which features it is meant to support. The main compressed sequence information is stored in an *encoded sequence* table, with the file suffix '.esq'. This table is the minimum requirement for the `GtEncseq` structure and must always be present. In addition, if support for multiple sequences is desired, a *sequence separator position* table with the '.ssp' suffix is required. If support for sequence descriptions is required, two additional tables are needed: a *description* table with the suffix '.des' and a *description separator* table with the file suffix '.sds'. Creation and requirement of these tables can be switched on and off using API functions as outlined below. The `GtEncseq` represents the stored sequences as one concatenated string. It allows access to the sequences by providing start positions and lengths for each sequence, making it possible to extract encoded substrings into a given buffer, as well as accessing single characters both in a random and a sequential fashion.

Methods

`const char* gt_encseq_indexname(const GtEncseq *encseq)`

Returns the indexname (as given at loading time) of `encseq` or "generated" if the GtEncseq was build in memory only.

`unsigned long gt_encseq_total_length(const GtEncseq *encseq)`

Returns the total number of characters in all sequences of `encseq`, including separators and wildcards.

`unsigned long gt_encseq_num_of_sequences(const GtEncseq *encseq)`

Returns the total number of sequences contained in `encseq`.

`GtUchar gt_encseq_get_encoded_char(const GtEncseq *encseq, unsigned long pos, GtReadmode readmode)`

Returns the encoded representation of the character at position `pos` of `encseq` read in the direction as indicated by `readmode`.

`char gt_encseq_get_decoded_char(const GtEncseq *encseq, unsigned long pos, GtReadmode readmode)`

Returns the decoded representation of the character at position `pos` of `encseq` read in the direction as indicated by `readmode`.

`bool gt_encseq_position_is_separator(const GtEncseq *encseq, unsigned long pos, GtReadmode readmode)`

Returns true iff `pos` is a separator position of `encseq` read in the direction as indicated by `readmode`.

`GtEncseq* gt_encseq_ref(GtEncseq *encseq)`

Increases the reference count of `encseq`.

`GtEncseqReader* gt_encseq_create_reader_with_readmode(const GtEncseq *encseq, GtReadmode readmode, unsigned long startpos)`

Returns a new `GtEncseqReader` for `encseq`, starting from position `startpos`. Also supports reading the sequence from the reverse and delivering (reverse) complement characters on DNA alphabets using the `readmode` option. Please make sure that the `GT_READMODE_COMPL` and `GT_READMODE_REVCOMPL` readmodes are only used on DNA alphabets.

`void gt_encseq_extract_encoded(const GtEncseq *encseq, GtUchar *buffer, unsigned long frompos, unsigned long topos)`

Returns the encoded representation of the substring from position `frompos` to position `topos` of `encseq`. The result is written to the location pointed to by `buffer`, which must be large enough to hold the result.

`void gt_encseq_extract_decoded(const GtEncseq *encseq, char *buffer, unsigned long frompos, unsigned long topos)`

Returns the decoded version of the substring from position `frompos` to position `topos` of `encseq`. The result is written to the location pointed to by `buffer`, which must be large enough to hold the result.

`unsigned long gt_encseq_seqlength(const GtEncseq *encseq, unsigned long seqnum)`

Returns the length of the `seqnum`-th sequence in the `encseq`. Requires multiple sequence support enabled in `encseq`.

`unsigned long gt_encseq_min_seq_length(const GtEncseq *encseq)`

Returns the length of the shortest sequence in the `encseq`.

`unsigned long gt_encseq_max_seq_length(const GtEncseq *encseq)`

Returns the length of the longest sequence in the `encseq`.

`bool gt_encseq_has_multiseq_support(const GtEncseq *encseq)`

Returns `true` if `encseq` has multiple sequence support.

`bool gt_encseq_has_description_support(const GtEncseq *encseq)`

Returns `true` if `encseq` has description support.

`bool gt_encseq_has_md5_support(const GtEncseq *encseq)`

Returns `true` if `encseq` has MD5 support.

`unsigned long gt_encseq_seqstartpos(const GtEncseq *encseq, unsigned long seqnum)`

Returns the start position of the `seqnum`-th sequence in the `encseq`. Requires multiple sequence support enabled in `encseq`.

`unsigned long gt_encseq_seqnum(const GtEncseq *encseq, unsigned long position)`

Returns the sequence number from the given `position` for a given GtEncseq `encseq`.

`const char* gt_encseq_description(const GtEncseq *encseq, unsigned long *desclen, unsigned long seqnum)`

Returns a pointer to the description of the `seqnum`-th sequence in the `encseq`. The length of the returned string is written to the location pointed at by `desclen`. The returned description pointer is not `\0`-terminated! Requires description support enabled in `encseq`.

`const GtStrArray* gt_encseq_filenames(const GtEncseq *encseq)`

Returns a `GtStrArray` of the names of the original sequence files contained in `encseq`.

`unsigned long gt_encseq_num_of_files(const GtEncseq *encseq)`

Returns the number of files contained in `encseq`.

`uint64_t gt_encseq_effective_filelength(const GtEncseq *encseq, unsigned long filenum)`

Returns the effective length (sum of sequence lengths and separators between them) of the `filenum`-th file contained in `encseq`.

`unsigned long gt_encseq_filestartpos(const GtEncseq *encseq, unsigned long filenum)`

Returns the start position of the sequences of the `filenum`-th file in the `encseq`. Requires multiple file support enabled in `encseq`.

`unsigned long gt_encseq_filenum(const GtEncseq *encseq, unsigned long position)`

Returns the file number from the given `position` for a given GtEncseq `encseq`.

`GtAlphabet* gt_encseq_alphabet(const GtEncseq *encseq)`

Returns the `GtAlphabet` associated with `encseq`.

`int gt_encseq_mirror(GtEncseq *encseq, GtError *err)`

Extends `encseq` by virtual reverse complement sequences. Returns 0 if mirroring has been successfully enabled, otherwise -1. `err` is set accordingly.

`void gt_encseq_unmirror(GtEncseq *encseq)`

Removes virtual reverse complement sequences added by `gt_encseq_mirror()`.

`bool gt_encseq_is_mirrored(const GtEncseq *encseq)`

Returns `true` if `encseq` contains virtual reverse complement sequences as added by `gt_encseq_mirror()`.

`unsigned long gt_encseq_version(const GtEncseq *encseq)`

Returns the version number of the file representation of `encseq` if it exists, or 0 if it was not mapped from a file.

`bool gt_encseq_is_64_bit(const GtEncseq *encseq)`

Returns TRUE if `encseq` was created on a 64-bit system.

`void gt_encseq_delete(GtEncseq *encseq)`

Deletes `encseq` and frees all associated space.

2.27 Class GtEncseqBuilder

The `GtEncseqBuilder` class creates `GtEncseq` objects by constructing uncompressed, encoded string copies in memory.

Methods

`GtEncseqBuilder* gt_encseq_builder_new(GtAlphabet *alpha)`

Creates a new `GtEncseqBuilder` using the alphabet `alpha` as a basis for on-the-fly encoding of sequences in memory.

`void gt_encseq_builder_enable_description_support(GtEncseqBuilder *eb)`

Enables support for retrieving descriptions from the encoded sequence to be built by `eb`. Requires additional memory to hold the descriptions and a position index. Activated by default.

`void gt_encseq_builder_disable_description_support(GtEncseqBuilder *eb)`

Disables support for retrieving descriptions from the encoded sequence to be built by `eb`. Disabling this support will result in an error when trying to call the method `gt_encseq_description()` on the `GtEncseq` object created by `eb`.

`void gt_encseq_builder_enable_multiseq_support(GtEncseqBuilder *eb)`

Enables support for random access to multiple sequences in the encoded sequence to be built by `eb`. Requires additional memory for an index of starting positions. Activated by default.

`void gt_encseq_builder_disable_multiseq_support(GtEncseqBuilder *eb)`

Disables support for random access to multiple sequences in the encoded sequence to be built by `eb`. Disabling this support will result in an error when trying to call the method `gt_encseq_seqlength()` or `gt_encseq_seqstartpos()` on the `GtEncseq` object created by `eb`.

`void gt_encseq_builder_create_esq_tab(GtEncseqBuilder *eb)`

Enables creation of the .esq table containing the encoded sequence itself. Naturally, enabled by default.

`void gt_encseq_builder_do_not_create_esq_tab(GtEncseqBuilder *eb)`

Disables creation of the .esq table.

`void gt_encseq_builder_create_des_tab(GtEncseqBuilder *eb)`

Enables creation of the .des table containing sequence descriptions.

`void gt_encseq_builder_do_not_create_des_tab(GtEncseqBuilder *eb)`

Disables creation of the .des table.

`void gt_encseq_builder_create_ssp_tab(GtEncseqBuilder *eb)`

Enables creation of the .ssp table containing indexes for multiple sequences.

`void gt_encseq_builder_do_not_create_ssp_tab(GtEncseqBuilder *eb)`

Disables creation of the .ssp table.

`void gt_encseq_builder_create_sds_tab(GtEncseqBuilder *eb)`

Enables creation of the .sds table containing indexes for sequence descriptions.

`void gt_encseq_builder_do_not_create_sds_tab(GtEncseqBuilder *eb)`

Disables creation of the .sds table.

`void gt_encseq_builder_add_cstr(GtEncseqBuilder *eb, const char *str, unsigned long strlen, const char *desc)`

Adds a sequence given as a C string `str` of length `strlen` to the encoded sequence to be built by `eb`. Additionally, a description can be given (`desc`). If description support is enabled, this must not be `NULL`. A copy will be made during the addition process and the sequence will be encoded using the alphabet set at the construction time of `eb`. Thus it must only contain symbols compatible with the alphabet.

`void gt_encseq_builder_add_str(GtEncseqBuilder *eb, GtStr *str, const char *desc)`

Adds a sequence given as a GtStr `str` to the encoded sequence to be built by `eb`. Additionally, a description can be given. If description support is enabled, `desc` must not be `NULL`. A copy will be made during the addition process and the sequence will be encoded using the alphabet set at the construction time of `eb`. Thus it must only contain symbols compatible with the alphabet.

`void gt_encseq_builder_add_encoded(GtEncseqBuilder *eb, const GtUchar *str, unsigned long strlen, const char *desc)`

Adds a sequence given as a pre-encoded string `str` of length `strlen` to the encoded sequence to be built by `eb`. `str` must be encoded using the alphabet set at the construction time of `eb`. Does not take ownership of `str`. Additionally, a description `desc` can be given. If description support is enabled, this must not be `NULL`.

`void gt_encseq_builder_add_encoded_own(GtEncseqBuilder *eb, const GtUchar *str, unsigned long strlen, const char *desc)`

Adds a sequence given as a pre-encoded string `str` of length `strlen` to the encoded sequence to be built by `eb`. `str` must be encoded using the alphabet set at the construction time of `eb`. Always creates a copy of `str`, so it can be used with memory that is to be freed immediately after adding. Additionally, a description `desc` can be given. If description support is enabled, this must not be `NULL`.

`void gt_encseq_builder_set_logger(GtEncseqBuilder*, GtLogger *l)`

Sets the logger to use by `ee` during encoding to `l`. Default is `NULL` (no logging).

`GtEncseq* gt_encseq_builder_build(GtEncseqBuilder *eb, GtError *err)`

Creates a new `GtEncseq` from the sequences added to `eb`. Returns a `GtEncseq` instance on success, or `NULL` on error. If an error occurred, `err` is set accordingly. The state of `eb` is reset to empty after successful creation of a new `GtEncseq` (like having called `gt_encseq_builder_reset()`).

`void gt_encseq_builder_reset(GtEncseqBuilder *eb)`

Clears all added sequences and descriptions, resetting `eb` to a state similar to the state immediately after its initial creation.

`void gt_encseq_builder_delete(GtEncseqBuilder *eb)`

Deletes `eb`.

2.28 Class GtEncseqEncoder

The `GtEncseqEncoder` class creates objects encapsulating a parameter set for conversion from sequence files into encoded sequence files on secondary storage.

Methods

`GtEncseqEncoder* gt_encseq_encoder_new(void)`

Creates a new `GtEncseqEncoder`.

`void gt_encseq_encoder_set_timer(GtEncseqEncoder *ee, GtTimer *t)`

Sets `t` to be the timer for `ee`. Default is `NULL` (no progress reporting).

`GtTimer* gt_encseq_encoder_get_timer(const GtEncseqEncoder *ee)`

Returns the timer set for `ee`.

`int gt_encseq_encoder_use_representation(GtEncseqEncoder *ee, const char *sat, GtError *err)`

Sets the representation of `ee` to `sat` which must be one of 'direct', 'bytecompress', 'bit', 'uchar', 'ushort' or 'uint32'. Returns 0 on success, and a negative value on error (`err` is set accordingly).

`GtStr* gt_encseq_encoder_representation(const GtEncseqEncoder *ee)`

Returns the representation requested for `ee`.

`int gt_encseq_encoder_use_symbolmap_file(GtEncseqEncoder *ee, const char *smap, GtError *err)`

Sets the symbol map file to use in `ee` to `smap` which must a valid alphabet description file. Returns 0 on success, and a negative value on error (`err` is set accordingly). Default is `NULL` (no alphabet transformation).

`const char* gt_encseq_encoder_symbolmap_file(const GtEncseqEncoder *ee)`

Returns the symbol map file requested for `ee`.

`void gt_encseq_encoder_set_logger(GtEncseqEncoder *ee, GtLogger *l)`

Sets the logger to use by `ee` during encoding to `l`. Default is `NULL` (no logging).

`void gt_encseq_encoder_enable_description_support(GtEncseqEncoder *ee)`

Enables support for retrieving descriptions from the encoded sequence encoded by `ee`. That is, the .des and .sds tables are created. This is a prerequisite for being able to activate description support in `gt_encseq_loader_require_description_support()`. Activated by default.

`void gt_encseq_encoder_disable_description_support(GtEncseqEncoder *ee)`

Disables support for retrieving descriptions from the encoded sequence encoded by `ee`. That is, the .des and .sds tables are not created. Encoded sequences created without this support will not be able to be loaded via a `GtEncseqLoader` with `gt_encseq_loader_require_description_support()` enabled.

`void gt_encseq_encoder_enable_multiseq_support(GtEncseqEncoder *ee)`

Enables support for random access to multiple sequences in the encoded sequence encoded by `ee`. That is, the .ssp table is created. This is a prerequisite for being able to activate description support in `gt_encseq_loader_require_multiseq_support()`. Activated by default.

`void gt_encseq_encoder_disable_multiseq_support(GtEncseqEncoder *ee)`

Disables support for random access to multiple sequences in the encoded sequence encoded by `ee`. That is, the .ssp table is not created. Encoded sequences created without this support will not be able to be loaded via a `GtEncseqLoader` with `gt_encseq_loader_require_multiseq_support()` enabled.

`void gt_encseq_encoder_enable_lossless_support(GtEncseqEncoder *ee)`

Enables support for lossless reproduction of the original sequence, regardless of alphabet transformations that may apply. Deactivated by default.

`void gt_encseq_encoder_disable_lossless_support(GtEncseqEncoder *ee)`

Enables support for lossless reproduction of the original sequence, regardless of alphabet transformations that may apply. Encoded sequences created without this support will not be able to be loaded via a `GtEncseqLoader` with `gt_encseq_loader_require_lossless_support()` enabled.

`void gt_encseq_encoder_enable_md5_support(GtEncseqEncoder *ee)`

Enables support for quick MD5 indexing of the sequences in `ee`. Activated by default.

`void gt_encseq_encoder_disable_md5_support(GtEncseqEncoder *ee)`

Enables support for quick MD5 indexing of the sequences in `ee`. Encoded sequences created without this support will not be able to be loaded via a `GtEncseqLoader` with `gt_encseq_loader_require_md5_support()` enabled.

`void gt_encseq_encoder_create_des_tab(GtEncseqEncoder *ee)`

Enables creation of the .des table containing sequence descriptions. Enabled by default.

`void gt_encseq_encoder_do_not_create_des_tab(GtEncseqEncoder *ee)`

Disables creation of the .des table.

`bool gt_encseq_encoder_des_tab_requested(const GtEncseqEncoder *ee)`

Returns `true` if the creation of the .des table has been requested, `false` otherwise.

`void gt_encseq_encoder_create_ssp_tab(GtEncseqEncoder *ee)`

Enables creation of the .ssp table containing indexes for multiple sequences. Enabled by default.

`void gt_encseq_encoder_do_not_create_ssp_tab(GtEncseqEncoder *ee)`

Disables creation of the .ssp table.

`bool gt_encseq_encoder_ssp_tab_requested(const GtEncseqEncoder *ee)`

Returns `true` if the creation of the .ssp table has been requested, `false` otherwise.

`void gt_encseq_encoder_create_sds_tab(GtEncseqEncoder *ee)`

Enables creation of the .sds table containing indexes for sequence descriptions. Enabled by default.

`void gt_encseq_encoder_do_not_create_sds_tab(GtEncseqEncoder *ee)`

Disables creation of the .sds table.

`bool gt_encseq_encoder_sds_tab_requested(const GtEncseqEncoder *ee)`

Returns `true` if the creation of the .sds table has been requested, `false` otherwise.

`void gt_encseq_encoder_create_md5_tab(GtEncseqEncoder *ee)`

Enables creation of the .md5 table containing MD5 sums. Enabled by default.

`void gt_encseq_encoder_do_not_create_md5_tab(GtEncseqEncoder *ee)`

Disables creation of the .md5 table.

`bool gt_encseq_encoder_md5_tab_requested(const GtEncseqEncoder *ee)`

Returns `true` if the creation of the .md5 table has been requested, `false` otherwise.

`void gt_encseq_encoder_set_input_dna(GtEncseqEncoder *ee)`

Sets the sequence input type for `ee` to DNA.

`bool gt_encseq_encoder_is_input_dna(GtEncseqEncoder *ee)`

Returns `true` if the input sequence has been defined as being DNA.

`void gt_encseq_encoder_set_input_protein(GtEncseqEncoder *ee)`

Sets the sequence input type for `ee` to protein/amino acids.

`bool gt_encseq_encoder_is_input_protein(GtEncseqEncoder *ee)`

Returns `true` if the input sequence has been defined as being protein.

`int gt_encseq_encoder_encode(GtEncseqEncoder *ee, GtStrArray *seqfiles, const char *indexname, GtError *err)`

Encodes the sequence files given in `seqfiles` using the settings in `ee` and `indexname` as the prefix for the index tables. Returns 0 on success, or a negative value on error (`err` is set accordingly).

`void gt_encseq_encoder_delete(GtEncseqEncoder *ee)`

Deletes `ee`.

2.29 Class GtEncseqLoader

The `GtEncseqLoader` class creates `GtEncseq` objects by mapping index files from secondary storage into memory.

Methods

`GtEncseqLoader* gt_encseq_loader_new(void)`
: Creates a new `GtEncseqLoader`.

`void gt_encseq_loader_enable_autosupport(GtEncseqLoader *el)`
: Enables auto-discovery of supported features when loading an encoded sequence. That is, if a file with `indexname.suffix` exists which is named like a table file, it is loaded automatically. Use`gt_encseq_has_multiseq_support()` etc. to query for these capabilities.

`void gt_encseq_loader_disable_autosupport(GtEncseqLoader *el)`
: Disables auto-discovery of supported features.

`void gt_encseq_loader_require_description_support(GtEncseqLoader *el)`
: Enables support for retrieving descriptions from the encoded sequence to be loaded by `el`. That is, the .des and .sds tables must be present. For example, these tables are created by having enabled the `gt_encseq_encoder_enable_description_support()` option when encoding. Activated by default.

`void gt_encseq_loader_drop_description_support(GtEncseqLoader *el)`
: Disables support for retrieving descriptions from the encoded sequence to be loaded by `el`. That is, the .des and .sds tables need not be present. However, disabling this support will result in an error when trying to call the method `gt_encseq_description()` on the `GtEncseq` object created by `el`.

`void gt_encseq_loader_require_multiseq_support(GtEncseqLoader *el)`
: Enables support for random access to multiple sequences in the encoded sequence to be loaded by `el`. That is, the .ssp table must be present. For example, this table is created by having enabled the `gt_encseq_encoder_enable_multiseq_support()` option when encoding. Activated by default.

`void gt_encseq_loader_drop_multiseq_support(GtEncseqLoader *el)`
: Disables support for random access to multiple sequences in the encoded sequence to be loaded by `el`. That is, the .ssp table needs not be present. However, disabling this support will result in an error when trying to call the method `gt_encseq_seqlength()` and `gt_encseq_seqstartpos()` on the `GtEncseq` object created by `el`.

`void gt_encseq_loader_require_lossless_support(GtEncseqLoader *el)`

Enables support for lossless reproduction of the original sequence in the encoded sequence to be loaded by `el`. That is, the .ois table must be present. For example, this table is created by having enabled the `gt_encseq_encoder_enable_lossless_support()` option when encoding. Deactivated by default.

`void gt_encseq_loader_drop_lossless_support(GtEncseqLoader *el)`

Disables support for lossless reproduction of the original sequence in the encoded sequence to be loaded by `el`. That is, the .ois table needs not be present. However, disabling this support may result in a reduced alphabet representation when accessing decoded characters.

`void gt_encseq_loader_require_md5_support(GtEncseqLoader *el)`

Enables support for quick retrieval of the MD5 sums for the sequences in the encoded sequence to be loaded by `el`. That is, the .md5 table must be present. For example, this table is created by having enabled the `gt_encseq_encoder_enable_md5_support()` option when encoding. Activated by default.

`void gt_encseq_loader_drop_md5_support(GtEncseqLoader *el)`

Disables support for quick retrieval of the MD5 sums for the sequences in the encoded sequence to be loaded by `el`. That is, the .md5 table needs not be present.

`void gt_encseq_loader_require_des_tab(GtEncseqLoader *el)`

Requires presence of the .des table containing sequence descriptions. Enabled by default.

`void gt_encseq_loader_do_not_require_des_tab(GtEncseqLoader *el)`

Disables requirement of the .des table for loading a `GtEncseq` using `el`.

`bool gt_encseq_loader_des_tab_required(const GtEncseqLoader *el)`

Returns `true` if a .des table must be present for loading to succeed.

`void gt_encseq_loader_require_ssp_tab(GtEncseqLoader *el)`

Requires presence of the .ssp table containing indexes for multiple sequences. Enabled by default.

`void gt_encseq_loader_do_not_require_ssp_tab(GtEncseqLoader *el)`

Disables requirement of the .ssp table for loading a `GtEncseq` using `el`.

`bool gt_encseq_loader_ssp_tab_required(const GtEncseqLoader *el)`

Returns `true` if a .ssp table must be present for loading to succeed.

`void gt_encseq_loader_require_sds_tab(GtEncseqLoader *el)`

Requires presence of the .sds table containing indexes for sequence descriptions. Enabled by default.

`void gt_encseq_loader_do_not_require_sds_tab(GtEncseqLoader *el)`

Disables requirement of the .sds table for loading a `GtEncseq` using `el`.

`bool gt_encseq_loader_sds_tab_required(const GtEncseqLoader *el)`

Returns `true` if a .sds table must be present for loading to succeed.

`void gt_encseq_loader_set_logger(GtEncseqLoader *el, GtLogger *l)`

Sets the logger to use by `ee` during encoding to `l`. Default is `NULL` (no logging).

`void gt_encseq_loader_mirror(GtEncseqLoader *el)`

Enables loading of a sequence using `el` with mirroring enabled from the start. Identical to invoking `gt_encseq_mirror()` directly after loading.

`void gt_encseq_loader_do_not_mirror(GtEncseqLoader *el)`

Disables loading of a sequence using `el` with mirroring enabled right from the start.

`GtEncseq* gt_encseq_loader_load(GtEncseqLoader *el, const char *indexname, GtError *err)`

Attempts to map the index files as specified by `indexname` using the options set in `el` using this interface. Returns a `GtEncseq` instance on success, or `NULL` on error. If an error occurred, `err` is set accordingly.

`void gt_encseq_loader_delete(GtEncseqLoader *el)`

Deletes `el`.

2.30 Class GtEncseqReader

The `GtEncseqReader` class represents the current state of a sequential scan of a `GtEncseq` region as an iterator.

Methods

`void gt_encseq_reader_reinit_with_readmode(GtEncseqReader *esr, const GtEncseq *encseq, GtReadmode readmode, unsigned long startpos)`

Reinitializes the given `esr` with the values as described in `gt_encseq_create_reader_with_readmode()`.

`GtUchar gt_encseq_reader_next_encoded_char(GtEncseqReader *esr)`

Returns the next encoded character from current position of `esr`, advancing the iterator by one position.

`char gt_encseq_reader_next_decoded_char(GtEncseqReader *esr)`

Returns the next decoded character from current position of `esr`, advancing the iterator by one position.

`void gt_encseq_reader_delete(GtEncseqReader *esr)`

Deletes `esr`, freeing all associated space.

2.31 Class GtError

This class is used for the handling of **user errors** in *GenomeTools*. Thereby, the actual `GtError` object is used to store the *error message* while it is signaled by the return value of the called function, if an error occured.
By convention in *GenomeTools*, the `GtError` object is always passed into a function as the last parameter and -1 (or `NULL` for constructors) is used as return value to indicate that an error occurred. Success is usually indicated by 0 as return value or via a non-`NULL` object pointer for constructors.
It is possible to use `NULL` as an `GtError` object, if one is not interested in the actual error message.
Functions which do not get an `GtError` object cannot fail due to a user error and it is not necessary to check their return code for an error condition.

Methods

`GtError* gt_error_new(void)`
: Return a new `GtError` object

`#define gt_error_check(err)`
: Insert an assertion to check that the error `err` is not set or is `NULL`. This macro should be used at the beginning of every routine which has an `GtError*` argument to make sure the error propagation has been coded correctly.

`void gt_error_set(GtError *err, const char *format, ...)`
: Set the error message stored in `err` according to `format` (as in `printf(3)`).

`void gt_error_vset(GtError *err, const char *format, va_list ap)`
: Set the error message stored in `err` according to `format` (as in `vprintf(3)`).

`void gt_error_set_nonvariadic(GtError *err, const char *msg)`
: Set the error message stored in `err` to `msg`.

`bool gt_error_is_set(const GtError *err)`
: Return `true` if the error `err` is set, `false` otherwise.

`void gt_error_unset(GtError *err)`
: Unset the error `err`.

`const char* gt_error_get(const GtError *err)`
: Return the error string stored in `err` (the error must be set).

`void gt_error_delete(GtError *err)`
: Delete the error object `err`.

2.32 Class GtExtractFeatureStream

Implements the GtNodeStream interface. A GtExtractFeatureStream extracts the corresponding sequences of features.

Methods

GtNodeStream* gt_extract_feature_stream_new(GtNodeStream *in_stream, GtRegionMapping *region_Mapping, const char *type, bool join, bool translate, bool seqid, bool target, unsigned long width, GtFile *outfp)

Create a GtExtractFeatureStream* which extracts the corresponding sequences of feature nodes (of the given type) it retrieves from in_stream and writes them in FASTA format (with the given width) to outfp. If join is true, features of the given type are joined together before the sequence is extracted. If translate is true, the sequences are translated into amino acid sequences before they are written to outfp. If seqid is true the sequence IDs of the extracted features are added to the FASTA header. If target is true the target IDs of the extracted features are added to the FASTA header. Takes ownership of region_mapping!

2.33 Class GtFeatureIndex

This interface represents a searchable container for GtFeatureNode objects, typically root nodes of larger structures. How storage and searching takes place is left to the discretion of the implementing class.
Output from a gt_feature_index_get_features_*() method should always be sorted by feature start position.

Methods

int gt_feature_index_add_region_node(GtFeatureIndex *feature_index, GtRegionNode *region_node, GtError *err)

Add region_node to feature_index.

int gt_feature_index_add_feature_node(GtFeatureIndex *feature_index, GtFeatureNode *feature_node, GtError *err)

Add feature_node to feature_index, associating it with a sequence region denoted by its identifier string.

int gt_feature_index_remove_node(GtFeatureIndex *feature_index, GtFeatureNode *node, GtError *err)

Removes node genome_node from feature_index.

`int gt_feature_index_add_gff3file(GtFeatureIndex *feature_index, const char *gff3file, GtError *err)`

Add all features contained in `gff3file` to `feature_index`, if `gff3file` is valid. Otherwise, `feature_index` is not changed and `err` is set.

`GtArray* gt_feature_index_get_features_for_seqid(GtFeatureIndex*, const char *seqid, GtError *err)`

Returns an array of `GtFeatureNodes` associated with a given sequence region identifier `seqid`.

`int gt_feature_index_get_features_for_range(GtFeatureIndex *feature_index, GtArray *results, const char *seqid, const GtRange *range, GtError*)`

Look up genome features in `feature_index` for sequence region `seqid` in `range` and store them in `results`.

`char* gt_feature_index_get_first_seqid(const GtFeatureIndex *feature_index, GtError *err)`

Returns the first sequence region identifier added to `feature_index`.

`GtStrArray* gt_feature_index_get_seqids(const GtFeatureIndex *feature_index, GtError *err)`

Returns a `GtStrArray` of all sequence region identifiers contained in `feature_index` (in alphabetical order).

`int gt_feature_index_get_range_for_seqid(GtFeatureIndex *feature_index, GtRange *range, const char *seqid, GtError *err)`

Writes the range of all features contained in the `feature_index` for region identifier `seqid` to the `GtRange` pointer `range`.

`int gt_feature_index_has_seqid(const GtFeatureIndex *feature_index, bool *has_seqid, const char *seqid, GtError *err)`

Returns `has_seqid` to true if the sequence region identified by `seqid` has been registered in the `feature_index`.

`int gt_feature_index_save(GtFeatureIndex *feature_index, GtError *err)`

TODO: document me

`void gt_feature_index_delete(GtFeatureIndex*)`

Deletes the `feature_index` and all its referenced features.

2.34 Class GtFeatureIndexMemory

The `GtFeatureIndexMemory` class implements a `GtFeatureIndex` in memory. Features are organized by region node. Each region node collects its feature nodes in an interval tree structure, which allows for efficient range queries.

Methods

`GtFeatureIndex* gt_feature_index_memory_new(void)`

Creates a new `GtFeatureIndexMemory` object.

`GtFeatureNode* gt_feature_index_memory_get_node_by_ptr(GtFeatureIndexMemory*, GtFeatureNode *ptr, GtError *err)`

Returns `ptr` if it is a valid node indexed in `GtFeatureIndexMemory`. Otherwise NULL is returned and `err` is set accordingly.

2.35 Class GtFeatureNode

Implements the `GtGenomeNode` interface. A single feature node corresponds to a GFF3 feature line (i.e., a line which does not start with `#`). Part-of relationships (which are realized in GFF3 with the `Parent` and `ID` attributes) are realized in the C API with the `gt_feature_node_add_child()` method.
Besides the “mere” feature nodes two “special” feature nodes exist: multi-features and pseudo-features.
Multi-features represent features which span multiple lines (it is indicated in GFF3 files by the fact, that each line has the same ID attribute).
To check if a feature is a multi-feature use the method `gt_feature_node_is_multi()`. Multi-features are connected via a “representative”. That is, two features are part of the same multi-feature if they have the same representative. The feature node representative can be be retrieved via the `gt_feature_node_get_multi_representative()` method.
Pseudo-features became a technical necessity to be able to pass related top-level features as a single entity through the streaming machinery. There are two cases in which a pseudo-feature has to be introduced.
First, if a multi-feature has no parent. In this case all features which comprise the multi-feature become the children of a pseudo-feature.
Second, if two or more top-level features have the same children (and are thereby connected). In this case all these top-level features become the children of a pseudo-feature.
It should be clear from the explanation above that pseudo-features make only sense as top-level features (a fact which is enforced in the code).
Pseudo-features are typically ignored during a traversal to give the illusion that they do not exist.

Methods

`GtGenomeNode* gt_feature_node_new(GtStr *seqid, const char *type, unsigned long start, unsigned long end, GtStrand strand)`

Return an new `GtFeatureNode` object on sequence with ID `seqid` and type `type` which lies from `start` to `end` on strand `strand`. The `GtFeatureNode*` stores a new reference to `seqid`, so make sure you do not modify the original `seqid` afterwards! `start` and `end` always refer to the forward strand, therefore `start` has to be smaller or equal than `end`.

`GtGenomeNode* gt_feature_node_new_pseudo(GtStr *seqid, unsigned long start, unsigned long end, GtStrand strand)`

Return a new pseudo-`GtFeatureNode` object on sequence with ID `seqid` which lies from `start` to `end` on strand `strand`. Pseudo-features do not have a type. The <GtFeatureNode > stores a new reference to `seqid`, so make sure you do not modify the original `seqid` afterwards. `start` and `end` always refer to the forward strand, therefore `start` has to be smaller or equal than `end`.

`GtGenomeNode* gt_feature_node_new_pseudo_template(GtFeatureNode *feature_node)`

Return a new pseudo-`GtFeatureNode` object which uses `feature_node` as template. That is, the sequence ID, range, strand, and source are taken from `feature_node`.

`GtGenomeNode* gt_feature_node_new_standard_gene(void)`

Return the “standard gene” (mainly for testing purposes).

`void gt_feature_node_add_child(GtFeatureNode *parent, GtFeatureNode *child)`

Add `child` feature node to `parent` feature node. `parent` takes ownership of `child`.

`const char* gt_feature_node_get_source(const GtFeatureNode *feature_node)`

Return the source of `feature_node`. If no source has been set, ”.” is returned. Corresponds to column 2 of GFF3 feature lines.

`void gt_feature_node_set_source(GtFeatureNode *feature_node, GtStr *source)`

Set the `source` of `feature_node`. Stores a new reference to `source`. Corresponds to column 2 of GFF3 feature lines.

`bool gt_feature_node_has_source(const GtFeatureNode *feature_node)`

Return `true` if `feature_node` has a defined source (i.e., on different from ”.”). `false` otherwise.

`const char* gt_feature_node_get_type(const GtFeatureNode *feature_node)`

Return the type of `feature_node`. Corresponds to column 3 of GFF3 feature lines.

`void gt_feature_node_set_type(GtFeatureNode *feature_node, const char *type)`

Set the type of `feature_node` to `type`.

`bool gt_feature_node_has_type(GtFeatureNode *feature_node, const char *type)`

Return `true` if `feature_node` has given `type`, `false` otherwise.

`unsigned long gt_feature_node_number_of_children(const GtFeatureNode *feature_node)`

Return the number of children for given `feature_node`.

`unsigned long gt_feature_node_number_of_children_of_type(const GtFeatureNode *parent, const GtFeatureNode *node)`

Return the number of children of type `node` for given GtFeatureNode `parent`.

`bool gt_feature_node_score_is_defined(const GtFeatureNode *feature_node)`

Return `true` if the score of `feature_node` is defined, `false` otherwise.

`float gt_feature_node_get_score(const GtFeatureNode *feature_node)`

Return the score of `feature_node`. The score has to be defined. Corresponds to column 6 of GFF3 feature lines.

`void gt_feature_node_set_score(GtFeatureNode *feature_node, float score)`

Set the score of `feature_node` to `score`.

`void gt_feature_node_unset_score(GtFeatureNode *feature_node)`

Unset the score of `feature_node`.

`GtStrand gt_feature_node_get_strand(const GtFeatureNode *feature_node)`

Return the strand of `feature_node`. Corresponds to column 7 of GFF3 feature lines.

`void gt_feature_node_set_strand(GtFeatureNode *feature_node, GtStrand strand)`

Set the strand of `feature_node` to `strand`.

`GtPhase gt_feature_node_get_phase(const GtFeatureNode *feature_node)`

Return the phase of `feature_node`. Corresponds to column 8 of GFF3 feature lines.

`void gt_feature_node_set_phase(GtFeatureNode *feature_node, GtPhase phase)`

Set the phase of `feature_node` to `phase`.

`const char* gt_feature_node_get_attribute(const GtFeatureNode *feature_node, const char *name)`

Return the attribute of `feature_node` with the given `name`. If no such attribute has been added, `NULL` is returned. The attributes are stored in column 9 of GFF3 feature lines.

`GtStrArray* gt_feature_node_get_attribute_list(const GtFeatureNode *feature_node)`

Return a string array containing the used attribute names of `feature_node`. The caller is responsible to free the returned `GtStrArray*`.

`void gt_feature_node_add_attribute(GtFeatureNode *feature_node, const char *tag, const char *value)`

Add attribute `tag=value` to `feature_node`. `tag` and `value` must at least have length 1. `feature_node` must not contain an attribute with the given `tag` already. You should not add Parent and ID attributes, use `gt_feature_node_add_child()` to denote part-of relationships.

`void gt_feature_node_set_attribute(GtFeatureNode* feature_node, const char *tag, const char *value)`

Set attribute `tag` to new `value` in `feature_node`, if it exists already. Otherwise the attribute `tag=value` is added to `feature_node`. `tag` and `value` must at least have length 1. You should not set Parent and ID attributes, use `gt_feature_node_add_child()` to denote part-of relationships.

`void gt_feature_node_remove_attribute(GtFeatureNode* feature_node, const char *tag)`

Remove attribute `tag` from `feature_node`. `feature_node` must contain an attribute with the given `tag` already! You should not remove Parent and ID attributes.

`bool gt_feature_node_is_multi(const GtFeatureNode *feature_node)`

Return `true` if `feature_node` is a multi-feature, `false` otherwise.

`bool gt_feature_node_is_pseudo(const GtFeatureNode *feature_node)`

Return `true` if `feature_node` is a pseudo-feature, `false` otherwise.

`void gt_feature_node_make_multi_representative(GtFeatureNode *feature_node)`

Make `feature_node` the representative of a multi-feature. Thereby `feature_node` becomes a multi-feature.

`void gt_feature_node_set_multi_representative(GtFeatureNode *feature_node, GtFeatureNode *representative)`

Set the multi-feature representative of `feature_node` to `representative`. Thereby `feature_node` becomes a multi-feature.

`void gt_feature_node_unset_multi(GtFeatureNode *feature_node)`

Unset the multi-feature status of `feature_node` and remove its multi-feature representative.

`GtFeatureNode* gt_feature_node_get_multi_representative(GtFeatureNode *feature_node)`

Return the representative of the multi-feature `feature_node`.

`bool gt_feature_node_is_similar(const GtFeatureNode *feature_node_a, const GtFeatureNode *feature_node_b)`

Returns `true`, if the given `feature_node_a` has the same seqid, feature type, range, strand, and phase as `feature_node_b`. Returns `false` otherwise.

`void gt_feature_node_mark(GtFeatureNode*)`

Marks the given `feature_node`.

`void gt_feature_node_unmark(GtFeatureNode*)`

If the given `feature_node` is marked it will be unmarked.

`bool gt_feature_node_contains_marked(GtFeatureNode *feature_node)`

Returns `true` if the given `feature_node` graph contains a marked node.

`bool gt_feature_node_is_marked(const GtFeatureNode *feature_node)`

Returns `true` if the (top-level) `feature_node` is marked.

2.36 Class GtFeatureNodeIterator

`GtFeatureNodeIterator* gt_feature_node_iterator_new(const GtFeatureNode *feature_node)`

Return a new `GtFeatureNodeIterator*` which performs a depth-first traversal of `feature_node` (including `feature_node` itself). It ignores pseudo-features.

`GtFeatureNodeIterator* gt_feature_node_iterator_new_direct(const GtFeatureNode *feature_node)`

Return a new `GtFeatureNodeIterator*` which iterates over all direct children of `feature_node` (without `feature_node` itself).

`GtFeatureNode* gt_feature_node_iterator_next(GtFeatureNodeIterator *feature_node_iterator)`

Return the next `GtFeatureNode*` in `feature_node_iterator` or `NULL` if none exists.

`void gt_feature_node_iterator_delete(GtFeatureNodeIterator *feature_node_iterator)`

Delete `feature_node_iterator`.

2.37 Class GtFile

This class defines (generic) files in *GenomeTools*. A generic file is is a file which either uncompressed or compressed (with gzip or bzip2). A `NULL`-pointer as generic file implies `stdout`.

Methods

`GtFile* gt_file_new(const char *path, const char *mode, GtError *err)`

Return a new `GtFile` object for the given `path` and open the underlying file handle with given `mode`. Returns `NULL` and sets `err` accordingly, if the file `path` could not be opened. The compression mode is determined by the ending of `path` (gzip compression if it ends with '.gz', bzip2 compression if it ends with '.bz2', and uncompressed otherwise).

`void gt_file_xprintf(GtFile *file, const char *format, ...)`

`printf(3)` for generic `file`.

`void gt_file_xfputs(const char *cstr, GtFile *file)`

Write \0-terminated C string `cstr` to `file`. Similar to `fputs(3)`, but terminates on error.

`int gt_file_xfgetc(GtFile *file)`

Return next character from `file` or `EOF`, if end-of-file is reached.

`int gt_file_xread(GtFile *file, void *buf, size_t nbytes)`

Read up to `nbytes` from generic `file` and store result in `buf`, returns bytes read.

`void gt_file_xwrite(GtFile *file, void *buf, size_t nbytes)`

Write `nbytes` from `buf` to given generic `file`.

`void gt_file_xrewind(GtFile *file)`

Rewind the generic `file`.

`void gt_file_delete(GtFile *file)`

Close the underlying file handle and destroy the `file` object.

2.38 Class GtGFF3InStream

Implements the `GtNodeStream` interface. A `GtGFF3InStream` parses GFF3 files and returns them as a stream of `GtGenomeNode` objects.

Methods

`GtNodeStream* gt_gff3_in_stream_new_unsorted(int num_of_files, const char **filenames)`

Return a `GtGFF3InStream` object which subsequently reads the `num_of_files` many GFF3 files denoted in `filenames`. The GFF3 files do not have to be sorted. If `num_of_files` is 0 or a file name is "-", it is read from `stdin`. The memory footprint is O(file size) in the worst-case.

`GtNodeStream* gt_gff3_in_stream_new_sorted(const char *filename)`

Create a `GtGFF3InStream*` which reads the sorted GFF3 file denoted by `filename`. If filename is `NULL`, it is read from `stdin`. The memory footprint is O(1) on average.

`void gt_gff3_in_stream_check_id_attributes(GtGFF3InStream *gff3_in_stream)`

Make sure all ID attributes which are parsed by `gff3_in_stream` are correct. Increases the memory footprint to O(file size).

`void gt_gff3_in_stream_enable_tidy_mode(GtGFF3InStream *gff3_in_stream)`

Enable tidy mode for `gff3_in_stream`. That is, the GFF3 parser tries to tidy up features which would normally lead to an error.

`void gt_gff3_in_stream_show_progress_bar(GtGFF3InStream *gff3_in_stream)`

Show progress bar on `stdout` to convey the progress of parsing the GFF3 files underlying `gff3_in_stream`.

2.39 Class GtGFF3OutStream

Implements the `GtNodeStream` interface. A `GtGFF3OutStream` produces GFF3 output. It automatically inserts termination lines at the appropriate places.

Methods

`GtNodeStream* gt_gff3_out_stream_new(GtNodeStream *in_stream, GtFile *outfp)`

Create a `GtGFF3OutStream*` which uses `in_stream` as input. It shows the nodes passed through it as GFF3 on `outfp`.

`void gt_gff3_out_stream_set_fasta_width(GtGFF3OutStream *gff3_out_stream, unsigned long fasta_width)`

Set the width with which the FASTA sequences of `GtSequenceNodes` passed through `gff3_out_stream` are shown to `fasta_width`. Per default, each FASTA entry is shown on a single line.

`void gt_gff3_out_stream_retain_id_attributes(GtGFF3OutStream *gff3_out_stream)`

If this method is called upon `gff3_out_stream`, use the original ID attributes provided in the input (instead of creating new ones, which is the default). Memory consumption for `gff3_out_stream` is raised from O(1) to O(`input_size`), because bookkeeping of used IDs becomes necessary to avoid ID collisions.

2.40 Class GtGFF3Parser

A `GtGFF3Parser` can be used to parse GFF3 files and convert them into `GtGenomeNode` objects. If the GFF3 files do not contain the encouraged sequence-region meta directives, the GFF3 parser introduces the corresponding region nodes automatically. This is a low-level class and it is usually not used directly. Normally, a `GtGFF3InStream` is used to parse GFF3 files.

Methods

`GtGFF3Parser* gt_gff3_parser_new(GtTypeChecker *type_checker)`

Return a new `GtGFF3Parser` object with optional `type_checker`. If a `type_checker` was given, the `GtGFF3Parser` stores a new reference to it internally and uses the `type_checker` to check types during parsing.

`void gt_gff3_parser_check_id_attributes(GtGFF3Parser *gff3_parser)`

Enable ID attribute checking in `gff3_parser`. Thereby, the memory consumption of the `gff3_parser` becomes proportional to the input file size(s).

`void gt_gff3_parser_check_region_boundaries(GtGFF3Parser *gff3_parser)`

Enable sequence region boundary checking in `gff3_parser`. That is, encountering features outside the sequence region boundaries will result in an error.

`void gt_gff3_parser_do_not_check_region_boundaries(GtGFF3Parser *gff3_parser)`

Disable sequence region boundary checking in `gff3_parser`. That is, features outside the sequence region boundaries will be permitted.

`void gt_gff3_parser_set_offset(GtGFF3Parser *gff3_parser, long offset)`

Transform all features parsed by `gff3_parser` by the given `offset`.

`void gt_gff3_parser_set_type_checker(GtGFF3Parser *gff3_parser, GtTypeChecker *type_checker)`

Set `type_checker` used by `gff3_parser`.

`void gt_gff3_parser_enable_tidy_mode(GtGFF3Parser *gff3_parser)`

Enable the tidy mode in `gff3_parser`. In tidy mode the `gff3_parser` parser tries to tidy up features which would normally lead to a parse error.

`int gt_gff3_parser_parse_genome_nodes(GtGFF3Parser *gff3_parser, int *status_code, GtQueue *genome_nodes, GtCstrTable *used_types, GtStr *filenamestr, unsigned long long *line_number, GtFile *fpin, GtError *err)`

Use `gff3_parser` to parse genome nodes from file pointer `fpin`. `status_code` is set to 0 if at least one genome node was created (and stored in `genome_nodes`) and to `EOF` if no further genome nodes could be parsed from `fpin`. Every encountered (genome feature) type is recorded in the C string table `used_types`. The parser uses the given `filenamestr` to store the file name of `fpin` in the created genome nodes or to give the correct filename in error messages, if necessary. `line_number` is increased accordingly during parsing and has to be set to 0 before parsing a new `fpin`. If an error occurs during parsing this method returns -1 and sets `err` accordingly.

`void gt_gff3_parser_reset(GtGFF3Parser *gff3_parser)`

Reset the `gff3_parser` (necessary if the input file is switched).

`void gt_gff3_parser_delete(GtGFF3Parser *gff3_parser)`

Delete the `gff3_parser`.

2.41 Class GtGFF3Visitor

Implements the `GtNodeVisitor` interface with a visitor that produces GFF3 output. This is a low-level class and it is usually not used directly. Normally, a `GtGFF3OutStream` is used to produce GFF3 output.

Methods

`GtNodeVisitor* gt_gff3_visitor_new(GtFile *outfp)`

Create a new `GtNodeVisitor*` which writes the output it produces to the given output file pointer `outfp`. If `outfp` is `NULL`, the output is written to `stdout`.

`void gt_gff3_visitor_set_fasta_width(GtGFF3Visitor *gff3_visitor, unsigned long fasta_width)`

Set the width with which the FASTA sequences of `GtSequenceNodes` visited by `gff3_visitor` are shown to `fasta_width`. Per default, each FASTA entry is shown on a single line.

`void gt_gff3_visitor_retain_id_attributes(GtGFF3Visitor *gff3_visitor)`

Retain the original ID attributes (instead of creating new ones), if possible. Memory consumption for `gff3_visitor` is raised from O(1) to O(`input_size`), because bookkeeping of used IDs becomes necessary to avoid ID collisions.

2.42 Class GtGTFInStream

Implements the `GtNodeStream` interface. A `GtGTFInStream` parses a GTF2.2 file and returns it as a stream of `GtGenomeNode` objects.

Methods

`GtNodeStream* gt_gtf_in_stream_new(const char *filename)`

Create a `GtGTFInStream*` which subsequently reads the GTF file with the given `filename`. If `filename` equals `NULL`, the GTF data is read from `stdin`.

2.43 Class GtGTFOutStream

Implements the `GtNodeStream` interface. A `GtGTFOutStream` produces GTF2.2 output.

Methods

`GtNodeStream* gt_gtf_out_stream_new(GtNodeStream *in_stream, GtFile *outfp)`

Create a `GtNodeStream*` which uses `in_stream` as input. It shows the nodes passed through it as GTF2.2 on `outfp`.

2.44 Class GtGenomeNode

The `GtGenomeNode` interface. The different implementation of the `GtGenomeNode` interface represent different parts of genome annotations (as they are usually found in GFF3 files).

Methods

`GtGenomeNode* gt_genome_node_ref(GtGenomeNode *genome_node)`

Increase the reference count for `genome_node` and return it. `genome_node` cannot be `NULL`.

`GtStr* gt_genome_node_get_seqid(GtGenomeNode *genome_node)`

Return the sequence ID of `genome_node`. Corresponds to column 1 of GFF3 feature lines.

`GtRange gt_genome_node_get_range(GtGenomeNode *genome_node)`

Return the genomic range of of `genome_node`. Corresponds to columns 4 and 5 of GFF3 feature lines.

`unsigned long gt_genome_node_get_start(GtGenomeNode *genome_node)`

Return the start of `genome_node`. Corresponds to column 4 of GFF3 feature lines.

`unsigned long gt_genome_node_get_end(GtGenomeNode *genome_node)`

Return the end of `genome_node`. Corresponds to column 5 of GFF3 feature lines.

`unsigned long gt_genome_node_get_length(GtGenomeNode *genome_node)`

Return the length of `genome_node`. Computed from column 4 and 5 of GFF3 feature lines.

`const char* gt_genome_node_get_filename(const GtGenomeNode* genome_node)`

Return the filename the `genome_node` was read from. If the node did not originate from a file, an appropriate string is returned.

`unsigned int gt_genome_node_get_line_number(const GtGenomeNode*)`

Return the line of the source file the `genome_node` was encountered on (if the node was read from a file, otherwise 0 is returned).

`void gt_genome_node_set_range(GtGenomeNode *genome_node, const GtRange *range)`

Set the genomic range of `genome_node` to given `range`.

`void gt_genome_node_add_user_data(GtGenomeNode *genome_node, const char *key, void *data, GtFree free_func)`

Attach a pointer to `data` to the `genome_node` using a given string as `key`. `free_func` is the optional destructor for `data`.

`void* gt_genome_node_get_user_data(const GtGenomeNode *genome_node, const char *key)`

Return the pointer attached to the `genome_node` for a given `key`.

`void gt_genome_node_release_user_data(GtGenomeNode *genome_node, const char *key)`

Call the destructor function associated with the user data attached to `genome_node` under the `key` on the attached data.

`int gt_genome_node_cmp(GtGenomeNode *genome_node_a, GtGenomeNode *genome_node_b)`

Compare `genome_node_a` with `genome_node_b` and return the result (similar to `strcmp(3)`). This method is the criterion used to sort genome nodes.

`void gt_genome_nodes_sort(GtArray *nodes)`

Sort node array `nodes`

`void gt_genome_nodes_sort_stable(GtArray *nodes)`

Sort node array `nodes` in a stable way

`int gt_genome_node_accept(GtGenomeNode *genome_node, GtNodeVisitor *node_visitor, GtError *err)`

Let `genome_node` accept the `node_visitor`. In the case of an error, -1 is returned and `err` is set accordingly.

`void gt_genome_node_delete(GtGenomeNode *genome_node)`

Decrease the reference count for `genome_node` or delete it, if this was the last reference.

2.45 Class GtGraphics

The `GtGraphics` interface acts as a low-level abstraction of a drawing surface. It is used as a common drawing object in `GtCanvas` and `GtCustomTrack` implementations and supports a variety of drawing operations for both text and basic primitive shapes.

Methods

`void gt_graphics_draw_text(GtGraphics*, double x, double y, const char*)`

Draws text in black to the right of (x,y). The coordinate `y` is used as a baseline.

`void gt_graphics_draw_text_clip(GtGraphics*, double x, double y, const char*)`

Draws text in black to the right of (x,y). The coordinate `y` is used as a baseline. If the text exceeds the margins, it is clipped.

`#define gt_graphics_draw_text_left(g,x,y,t)`

Synonym to `gt_graphics_draw_text()`

`void gt_graphics_draw_text_centered(GtGraphics*, double x, double y, const char*)`

Draws text in black centered at (x,y). The coordinate `y` is used as a baseline.

`void gt_graphics_draw_text_right(GtGraphics*, double x, double y, const char*)`

Draws text in black to the left of (x,y). The coordinate `y` is used as a baseline.

`void gt_graphics_draw_colored_text(GtGraphics*, double x, double y, GtColor, const char*)`

Draws text in a given `GtColor` to the right of (x,y). The coordinate `y` is used as a baseline.

`double gt_graphics_get_text_height(GtGraphics*)`

Returns the height of a capital letter in pixels/points.

`int gt_graphics_set_background_color(GtGraphics*, GtColor)`

Sets the background color of the `GtGraphics` to a specific color. Note that this may only be supported for bitmap output formats.

`double gt_graphics_get_text_width(GtGraphics*, const char *text)`

Returns the width of the given string in pixels/points.

`void gt_graphics_set_font(GtGraphics *g, const char *family, FontSlant slant, FontWeight weight, double size)`

Sets basic font family, slant and weight options. Font families are implementation-specific, e.g. in Cairo there is no operation to list available family names on the system, but the standard CSS2 generic family names, (”serif”, ”sans-serif”, ”cursive”, ”fantasy”, ”monospace”), are likely to work as expected.

`double gt_graphics_get_image_width(GtGraphics*)`

Returns the width of the image in pixels/points.

`double gt_graphics_get_image_height(GtGraphics*)`

Returns the height of the image in pixels/points.

`void gt_graphics_set_margins(GtGraphics*, double margin_x, double margin_y)`

Set margins (space to the image boundaries that are clear of elements) in the graphics. `margin_x` denotes the Margin to the left and right, in pixels. `margin_y` denotes the Margin to the top and bottom, in pixels.

`double gt_graphics_get_xmargins(GtGraphics*)`

Returns the horizontal margins in pixels/points.

`double gt_graphics_get_ymargins(GtGraphics*)`

Returns the vertical margins in pixels/points.

`void gt_graphics_draw_horizontal_line(GtGraphics *g, double x, double y, GtColor color, double width, double stroke_width)`

Draws a horizontal line of length `width` beginning at the given coordinates to the right in the color `color` with stroke width `stroke_width`.

`void gt_graphics_draw_vertical_line(GtGraphics *g, double x, double y, GtColor color, double length, double stroke_width)`

Draws a vertical line of length `length` beginning at the given coordinates downwards in the color `color` with stroke width `stroke_width`.

`void gt_graphics_draw_line(GtGraphics *g, double x, double y, double xto, double yto, GtColor color, double stroke_width)`

Draws a line beginning at (`x`,`y`) to (`xto`,`yto`) in the color `color` with stroke width `stroke_width`.

`void gt_graphics_draw_box(GtGraphics*, double x, double y, double width, double height, GtColor fill_color, ArrowStatus arrow_status, double arrow_width, double stroke_width, GtColor stroke_color, bool dashed)`

Draws a arrow-like box glyph at (`x`,`y`) where these are the top left coordinates. The box extends `width` pixels (incl. arrowhead) into the x direction and `height` pixels into the y direction. It will be filled with `fill_color` and stroked with width `stroke_width` and color `stroke_color`. The width of the arrowhead is given by the `arrow_width` parameter. The `arrow_status` parameter determines whether an arrowhead will be drawn at the left or right end, both ends, or none. If `dashed` is set to true, then the outline will be dashed instead of solid.

`void gt_graphics_draw_dashes(GtGraphics*, double x, double y, double width, double height, ArrowStatus arrow_status, double arrow_width, double stroke_width, GtColor stroke_color)`

Draws a transparent box with a dashed line at the center at (`x`,`y`) (where these are the top left coordinates). The box extends `width` pixels (incl. arrowhead) into the x direction and `height` pixels into the y direction. It will be stroked with width `stroke_width` and color `stroke_color`. The width of the arrowhead is given by the `arrow_width` parameter. The `arrow_status` parameter determines whether an arrowhead will be drawn at the left or right end, both ends, or none.

`void gt_graphics_draw_caret(GtGraphics*, double x, double y, double width, double height, ArrowStatus arrow_status, double arrow_width, double stroke_width, GtColor stroke_color)`

Draws a caret ("hat") style glyph at (x,y) (where these are the top left coordinates). The box extends `width` pixels (incl. arrowhead) into the x direction and `height` pixels into the y direction. It will be stroked with width `stroke_width` and color `stroke_color`. The width of the arrowhead is given by the `arrow_width` parameter. The `arrow_status` parameter determines whether an arrowhead will be drawn at the left or right end, both ends, or none.

`void gt_graphics_draw_rectangle(GtGraphics*, double x, double y, bool filled, GtColor fill_color, bool stroked, GtColor stroke_color, double stroke_width, double width, double height)`

Draws a rectangle at (x,y) where these are the top left coordinates. The rectangle extends `width` pixels (incl. arrowhead) into the x direction and `height` pixels into the y direction. It will be filled with `fill_color` if `filled` is set to true and stroked with width `stroke_width` and color `stroke_color` if `stroked` is set to true.

`void gt_graphics_draw_arrowhead(GtGraphics*, double x, double y, GtColor, ArrowStatus arrow_status)`

Draws an arrowhead at (x,y) where these are the top left coordinates. The direction is determined by the `arrow_status` parameter.

`void gt_graphics_draw_curve_data(GtGraphics *g, double x, double y, GtColor color, double data[], unsigned long ndata, GtRange valrange, unsigned long height)`

Draws a curve over the full visible image width (without margins) at (x,y) where these are the top left coordinates. As input, the array of double values `data` with `ndata` data points is used. The `valrange` gives the minimum and maximum value of the displayed data. If a value outside the data range is encountered, the drawing will be stopped at this data point.

`int gt_graphics_save_to_file(const GtGraphics*, const char *filename, GtError*)`

Write out the `GtGraphics` object to the given file with `filename`.

`void gt_graphics_save_to_stream(const GtGraphics*, GtStr *stream)`

Write out the `GtGraphics` object to the given `stream`.

`void gt_graphics_delete(GtGraphics*)`

Deletes the the `GtGraphics` object.

2.46 Class GtHashmap

A hashmap allowing to index any kind of pointer (as a value). As keys, strings or any other pointer can be used.

Methods

`GtHashmap* gt_hashmap_new(GtHashType keyhashtype, GtFree keyfree, GtFree valuefree)`

Creates a new `GtHashmap` object of type `keyhashtype`. If `keyfree` and/or `valuefree` are given, they will be used to free the hashmap members when the `GtHashmap` is deleted. `keyhashtype` defines how to hash the keys given when using the `GtHashmap`. `GT_HASH_DIRECT` uses the key pointer as a basis for the hash function. Equal pointers will refer to the same value. If `GT_HASH_STRING` is used, the keys will be evaluated as strings and keys will be considered equal if the strings are identical, regardless of their address in memory

`GtHashmap* gt_hashmap_ref(GtHashmap *hm)`

Increase the reference count of `hm`.

`void* gt_hashmap_get(GtHashmap *hashmap, const void *key)`

Return the value stored in `hashmap` for `key` or `NULL` if no such key exists.

`void gt_hashmap_add(GtHashmap *hashmap, void *key, void *value)`

Set the value stored in `hashmap` for `key` to `value`, overwriting the prior value for that key if present.

`void gt_hashmap_remove(GtHashmap *hashmap, const void *key)`

Remove the member with key `key` from `hashmap`.

`int gt_hashmap_foreach_ordered(GtHashmap *hashmap, GtHashmapVisitFunc func, void *data, GtCompare cmp, GtError *err)`

Iterate over `hashmap` in order given by compare function `cmp`. For each member, `func` is called (see interface).

`int gt_hashmap_foreach(GtHashmap *hashmap, GtHashmapVisitFunc func, void *data, GtError *err)`

Iterate over `hashmap` in arbitrary order. For each member, `func` is called (see interface).

`int gt_hashmap_foreach_in_key_order(GtHashmap *hashmap, GtHashmapVisitFunc func, void *data, GtError *err)`

Iterate over `hashmap` in either alphabetical order (if `GtHashType` was specified as `GT_HASH_STRING`) or numerical order (if `GtHashType` was specified as `GT_HASH_DIRECT`).

`void gt_hashmap_reset(GtHashmap *hashmap)`

Reset `hashmap` by unsetting values for all keys, calling the free function if necessary.

`void gt_hashmap_delete(GtHashmap *hashmap)`

Delete `hashmap`, calling the free function if necessary.

2.47 Class GtIDToMD5Stream

Implements the GtNodeStream interface. A GtIDToMD5Stream converts "regular" sequence IDs to MD5 fingerprints.

Methods

GtNodeStream* gt_id_to_md5_stream_new(GtNodeStream *in_stream, GtRegionMapping *region_mapping, bool substitute_target_ids)

Create a GtIDToMD5Stream object which converts "regular" sequence IDs from nodes it retrieves from its in_stream to MD5 fingerprints (with the help of the given region_mapping). If substitute_target_ids is true, the IDs of Target attributes are also converted to MD5 fingerprints. Takes ownership of region_mapping!

2.48 Class GtImageInfo

The GtImageInfo class is a container for 2D coordinate to GtFeatureNode mappings which could, for example, be used to associate sections of a rendered image with GUI widgets or HTML imagemap areas. This information is given in the form of GtRecMap objects. They are created during the image rendering process and stored inside a GtImageInfo object for later retrieval. Additionally, the rendered width of an image can be obtained via a GtImageInfo method.

Methods

GtImageInfo* gt_image_info_new(void)

Creates a new GtImageInfo object.

unsigned int gt_image_info_get_height(GtImageInfo *image_info)

Returns the height of the rendered image (in pixels or points).

unsigned long gt_image_info_num_of_rec_maps(GtImageInfo *image_info)

Returns the total number of mappings in image_info.

const GtRecMap* gt_image_info_get_rec_map(GtImageInfo *image_info, unsigned long i)

Returns the i-th GtRecMap mapping in image_info.

void gt_image_info_delete(GtImageInfo *image_info)

Deletes image_info and all the GtRecMap objects created by it.

2.49 Class GtInterFeatureStream

Implements the GtNodeStream interface. A GtInterFeatureStream inserts new feature nodes between existing feature nodes of a certain type.

Methods

`GtNodeStream* gt_inter_feature_stream_new(GtNodeStream *in_stream, const char *outside_type, const char *inter_type)`

Create a `GtInterFeatureStream*` which inserts feature nodes of type `inter_type` between the feature nodes of type `outside_type` it retrieves from `in_stream` and returns them.

2.50 Class GtIntervalTree

This is an interval tree data structure, implemented according to Cormen et al., Introduction to Algorithms, 2nd edition, MIT Press, Cambridge, MA, USA, 2001

Methods

`GtIntervalTree* gt_interval_tree_new(GtFree)`

Creates a new `GtIntervalTree`. If a `GtFree` function is given as an argument, it is applied on the data pointers in all inserted nodes when the `GtIntervalTree` is deleted.

`unsigned long gt_interval_tree_size(GtIntervalTree*)`

Returns the number of elements in the `GtIntervalTree`.

`GtIntervalTreeNode* gt_interval_tree_find_first_overlapping(GtIntervalTree*, unsigned long start, unsigned long end)`

Returns the first node in the `GtIntervalTree` which overlaps the given range (from `start` to `end`).

`void gt_interval_tree_insert(GtIntervalTree *tree, GtIntervalTreeNode *node)`

Inserts node `node` into `tree`.

`void gt_interval_tree_find_all_overlapping(GtIntervalTree*, unsigned long start, unsigned long end, GtArray*)`

Collects data pointers of all `GtIntervalTreeNodes` in the tree which overlap with the query range (from `start` to `end`) in a `GtArray`.

`void gt_interval_tree_iterate_overlapping(GtIntervalTree *it, GtIntervalTreeIteratorFunc func, unsigned long start, unsigned long end, void *data)`

Call `func` for all `GtIntervalTreeNodes` in the tree which overlap with the query range (from `start` to `end`). Use `data` to pass in arbitrary user data.

`int gt_interval_tree_traverse(GtIntervalTree*, GtIntervalTreeIteratorFunc func, void *data)`

Traverses the `GtIntervalTree` in a depth-first fashion, applying `func` to each node encountered. The `data` pointer can be used to reference arbitrary data needed in the `GtIntervalTreeIteratorFunc`.

`void gt_interval_tree_remove(GtIntervalTree*, GtIntervalTreeNode *node)`

> Removes the entry referenced by `node` from the `GtIntervalTree`. The data attached to `node` is freed according to the free function defined in the tree. Note that the memory pointed to by `node` can be re-used internally, referencing other data in the tree. Make sure to handle this pointer as expired after calling `gt_interval_tree_remove()`!

`void gt_interval_tree_delete(GtIntervalTree*)`

> Deletes a `GtIntervalTree`. If a `GtFree` function was set in the tree constructor, data pointers specified in the nodes are freed using the given `GtFree` function.

2.51 Class GtIntervalTreeNode

`GtIntervalTreeNode* gt_interval_tree_node_new(void *data, unsigned long low, unsigned long high)`

> Creates a new `GtIntervalTreeNode`. Transfers ownership of `data` to interval tree if inserted into a `GtIntervalTree` in which a `GtIntervalTreeDataFreeFunc` is set.

`void* gt_interval_tree_node_get_data(GtIntervalTreeNode* node)`

> Returns a pointer to the data associated with node `node`.

2.52 Class GtLayout

The `GtLayout` class represents contents (tracks) of a `GtDiagram` broken up into lines such that a given horizontal space allotment given in pixels or points is used up most efficiently. This is done using the `GtLineBreaker` and `GtTextWidthCalculator` classes. As defaults, Cairo-based instances of these classes are used but can be specified separately.
A `GtLayout` can be queried for the height of the laid out representation and finally be rendered to a `GtCanvas`.

Methods

`GtLayout* gt_layout_new(GtDiagram *diagram, unsigned int width, GtStyle*, GtError*)`

> Creates a new `GtLayout` object for the contents of `diagram`. The layout is done for a target image width of `width` and using the rules in `GtStyle` object `style`.

`GtLayout* gt_layout_new_with_twc(GtDiagram*, unsigned int width, GtStyle*, GtTextWidthCalculator*, GtError*)`

> Like `gt_layout_new()`, but allows use of a different `GtTextWidthCalculator` implementation.

`void gt_layout_set_track_ordering_func(GtLayout *layout, GtTrackOrderingFunc func, void *data)`

Sets the `GtTrackOrderingFunc` comparator function `func` which defines an order on the tracks contained in `layout`. This determines the order in which the tracks are drawn vertically. Additional data necessary in the comparator function can be given in `data`, the caller is responsible to free it.

`int gt_layout_get_height(const GtLayout *layout, unsigned long *result, GtError *err)`

Calculates the height of `layout` in pixels. The height value is written to the location pointed to by `result`. If an error occurs during the calculation, this function returns -1 and `err` is set accordingly. Returns 0 on success.

`int gt_layout_sketch(GtLayout *layout, GtCanvas *target_canvas, GtError*)`

Renders `layout` on the `target_canvas`.

`void gt_layout_delete(GtLayout*)`

Destroys a layout.

2.53 Class GtLogger

`GtLogger* gt_logger_new(bool enabled, const char *prefix, FILE *target)`

Creates a new `GtLogger`, with logging `enabled` or not, and prefixing all log entries with `prefix` (e.g. ”debug”). The log output is terminated by a newline. All log output will be written to `target`.

`void gt_logger_enable(GtLogger *logger)`

Enable logging on `logger`.

`void gt_logger_disable(GtLogger *logger)`

Disable logging on `logger`.

`bool gt_logger_enabled(GtLogger *logger)`

Return `true` if logging is enabled on `logger`, false otherwise.

`FILE* gt_logger_target(GtLogger *logger)`

Return logging target of `logger`.

`void gt_logger_set_target(GtLogger *logger, FILE *fp)`

Set logging target of `logger` to `fp`.

`void gt_logger_log_force(GtLogger *logger, const char *format, ...)`

Log to target regardless of logging status.

`void gt_logger_log(GtLogger *logger, const char *format, ...)`

Log to target depending on logging status.

`void gt_logger_log_va_force(GtLogger *logger, const char *format, va_list)`

Log to target regardless of logging status, using a va_list argument.

`void gt_logger_log_va(GtLogger *logger, const char *format, va_list)`

Log to target depending on logging status, using a va_list argument.

`void gt_logger_delete(GtLogger *logger)`

Delete `logger`.

2.54 Class GtMD5ToIDStream

Implements the `GtNodeStream` interface. A `GtMD5ToIDStream` converts MD5 fingerprints used as sequence IDs to "regular" ones.

Methods

`GtNodeStream* gt_md5_to_id_stream_new(GtNodeStream *in_stream, GtRegionMapping *region_mapping)`

Create a `GtMD5toIDStream*` which converts MD5 sequence IDs from nodes it retrieves from its `in_stream` to "regular" ones (with the help of the given `region_mapping`). Takes ownership of `region_mapping`!

2.55 Class GtMatchBlast

`GtMatch* gt_match_blast_new(char *seqid1, char *seqid2, unsigned long start_seq1, unsigned long start_seq2, unsigned long end_seq1, unsigned long end_seq2, double evalue, float bitscore, unsigned long ali_l, double similarity, GtMatchDirection dir)`

Creates a new `GtMatch` object meant to store results in the BLAST format. That is, it stores double values `evalue` for match E-values, `bitscore`s and the alignment length `ali_l` in addition to the generic match contents `seqid1`, `seqid2`, `start_seq1`, `start_seq2`, `end_seq1`, and `end_seq2`.

`void gt_match_blast_set_evalue(GtMatchBlast *mb, long double evalue)`

Sets `evalue` to be the E-value in `mb`.

`void gt_match_blast_set_bitscore(GtMatchBlast *mb, float bits)`

Sets `bits` to be the bit-score in `mb`.

`void gt_match_blast_set_align_length(GtMatchBlast *mb, unsigned long length)`

Sets `length` to be the alignment length in `mb`.

`void gt_match_blast_set_similarity(GtMatchBlast *mb, double similarity)`

Sets `similarity` to be the match similarity in `mb`.

`long double gt_match_blast_get_evalue(GtMatchBlast *mb)`

Returns the E-value stored in `mb`.

`float gt_match_blast_get_bitscore(GtMatchBlast *mb)`

Returns the bit-score value stored in `mb`.

`unsigned long gt_match_blast_get_align_length(GtMatchBlast *mb)`

Returns the alignment length stored in `mb`.

`double gt_match_blast_get_similarity(GtMatchBlast *mb)`

Returns the alignment similarity stored in `mb`.

2.56 Class GtMatchIterator

`GtMatchIteratorStatus gt_match_iterator_next(GtMatchIterator *mp, GtMatch **match, GtError *err)`

Advances `mp` by one, returning the next match. Writes a pointer to the next match to the position pointed to by `match`. Returns GT_MATCHER_STATUS_OK when the match could be delivered and there are more matches to come, GT_MATCHER_STATUS_END when no more matches are available, and GT_MATCHER_STATUS_ERROR if an error occurred. `err` is set accordingly.

`void gt_match_iterator_delete(GtMatchIterator *mp)`

Deletes `mp`, freeing all associated space.

2.57 Class GtMatchLAST

`GtMatch* gt_match_last_new(const char *seqid1, const char *seqid2, unsigned long score, unsigned long seqno1, unsigned long seqno2, unsigned long start_seq1, unsigned long start_seq2, unsigned long end_seq1, unsigned long end_seq2, GtMatchDirection dir)`

Creates a new `GtMatch` object meant to store results from the LAST software.

`unsigned long gt_match_last_get_seqno1(const GtMatchLAST *ml)`

Returns the sequence number of the match `ms` in the first `GtEncseq`.

`unsigned long gt_match_last_get_seqno2(const GtMatchLAST *ml)`

Returns the sequence number of the match `ms` in the second `GtEncseq`.

`unsigned long gt_match_last_get_score(const GtMatchLAST *ml)`

Returns the LAST score of the match `ms`.

2.58 Class GtMatchOpen

`GtMatch* gt_match_open_new(char *seqid1, char *seqid2, unsigned long start_seq1, unsigned long start_seq2, unsigned long end_seq1, unsigned long end_seq2, long weight, GtMatchDirection dir)`

Creates a new `GtMatch` object meant to store results in the OpenMatch format. That is, it stores long values `weight` in addition to the generic match contents `seqid1`, `seqid2`, `start_seq1`, `start_seq2`, `end_seq1`, and `end_seq2`.

`void gt_match_open_set_weight(GtMatchOpen *mo, long weight)`

Sets `weight` to be the weight value in `mo`.

`long gt_match_open_get_weight(GtMatchOpen *mo)`

Returns the weight value stored in `mo`.

2.59 Class GtMatchSW

`GtMatch* gt_match_sw_new(const char *seqid1, const char *seqid2, unsigned long seqno1, unsigned long seqno2, unsigned long length, unsigned long edist, unsigned long start_seq1, unsigned long start_seq2, unsigned long end_seq1, unsigned long end_seq2, GtMatchDirection dir)`

Creates a new `GtMatch` object meant to store results from Smith-Waterman matching (using the `swalign` module). That is, it stores the alignment length `length`, the edit distance `edist` and the sequence numbers in the `GtEncseqs` in addition to the generic match contents `seqid1`, `seqid2`, `start_seq1`, `start_seq2`, `end_seq1` and `end_seq2`.

`unsigned long gt_match_sw_get_seqno1(const GtMatchSW *ms)`

Returns the sequence number of the match `ms` in the first `GtEncseq`.

`unsigned long gt_match_sw_get_seqno2(const GtMatchSW *ms)`

Returns the sequence number of the match `ms` in the second `GtEncseq`.

`unsigned long gt_match_sw_get_alignment_length(const GtMatchSW *ms)`

Returns the alignment length of the match `ms`.

`unsigned long gt_match_sw_get_edist(const GtMatchSW *ms)`

Returns the edit distance of the match `ms`.

2.60 Class GtMergeFeatureStream

Implements the `GtNodeStream` interface. A `GtMergeFeatureStream` merges adjacent features of the same type.

Methods

`GtNodeStream* gt_merge_feature_stream_new(GtNodeStream *in_stream)`

Create a `GtMergeFeatureStream*` which merges adjacent features of the same type it retrieves from `in_stream` and returns them (and all other unmodified features).

2.61 Class GtMergeStream

Implements the `GtNodeStream` interface. A `GtMergeStream` allows one to merge a set of sorted streams in a sorted fashion.

Methods

`GtNodeStream* gt_merge_stream_new(const GtArray *node_streams)`

Create a `GtMergeStream*` which merges the given (sorted) `node_streams` in a sorted fashion.

2.62 Class GtMetaNode

Implements the `GtGenomeNode` interface. Meta nodes correspond to meta lines in GFF3 files (i.e., lines which start with "`##`") which are not sequence-region lines.

Methods

`GtGenomeNode* gt_meta_node_new(const char *meta_directive, const char *meta_data)`

Return a new `GtMetaNode` object representing a `meta_directive` with the corresponding `meta_data`. Please note that the leading "`##`" which denotes meta lines in GFF3 files should not be part of the `meta_directive`.

`const char* gt_meta_node_get_directive(const GtMetaNode *meta_node)`

Return the meta directive stored in `meta_node`.

`const char* gt_meta_node_get_data(const GtMetaNode *meta_node)`

Return the meta data stored in `meta_node`.

2.63 Class GtNodeStream

The `GtNodeStream` interface. `GtNodeStream` objects process `GtGenomeNode` objects in a pull-based architecture and can be chained together.

Methods

`GtNodeStream* gt_node_stream_ref(GtNodeStream *node_stream)`

Increase the reference count for `node_stream` and return it.

`int gt_node_stream_next(GtNodeStream *node_stream, GtGenomeNode **genome_node, GtError *err)`

Try to get the the next `GtGenomeNode` from `node_stream` and store it in `genome_node` (transfers ownership to `genome_node`). If no error occurs, 0 is returned and `genome_node` contains either the next `GtGenomeNode` or `NULL`, if the `node_stream` is exhausted. If an error occurs, -1 is returned and `err` is set accordingly (the status of `genome_node` is undefined, but no ownership transfer occured).

`int gt_node_stream_pull(GtNodeStream *node_stream, GtError *err)`

Calls `gt_node_stream_next()` on `node_stream` repeatedly until the `node_stream` is exhausted (0 is returned) or an error occurs (-1 is returned and `err` is set). All retrieved `GtGenomeNodes` are deleted automatically with calls to `gt_genome_node_delete()`. This method is basically a convenience method which simplifies calls to `gt_node_stream_next()` in a loop where the retrieved `GtGenomeNode` objects are not processed any further.

`bool gt_node_stream_is_sorted(GtNodeStream *node_stream)`

Return `true` if `node_stream` is a sorted stream, `false` otherwise.

`void gt_node_stream_delete(GtNodeStream *node_stream)`

Decrease the reference count for `node_stream` or delete it, if this was the last reference.

`GtNodeStream* gt_node_stream_create(const GtNodeStreamClass *node_stream_class, bool ensure_sorting)`

Create a new object of the given `node_stream_class`. If `ensure_sorting` is `true`, it is enforced that all genome node objects pulled from this class are sorted. That is, for consecutive nodes `a` and `b` obtained from the given `node_stream_class` the return code of `gt_genome_node_compare(a,b)` has to be smaller or equal than 0. If this condition is not met, an assertion fails.

`void* gt_node_stream_cast(const GtNodeStreamClass *node_stream_class, GtNodeStream *node_stream)`

Cast `node_stream` to the given `node_stream_class`. That is, if `node_stream` is not from the given `node_stream_class`, an assertion will fail.

2.64 Class GtNodeStreamClass

`const GtNodeStreamClass* gt_node_stream_class_new(size_t size, GtNodeStreamFreeFunc free, GtNodeStreamNextFunc next)`

Create a new node stream class (that is, a class which implements the node stream interface). `size` denotes the size of objects of the new node stream class. The optional `free` method is called once, if an object of the new class is deleted. The mandatory `next` method has to implement the `gt_node_stream_next()` semantic for the new class.

2.65 Class GtNodeVisitor

The `GtNodeVisitor` interface, a visitor for `GtGenomeNode` objects.

Methods

`int gt_node_visitor_visit_comment_node(GtNodeVisitor *node_visitor, GtCommentNode *comment_node, GtError *err)`

Visit `comment_node` with `node_visitor`.

`int gt_node_visitor_visit_feature_node(GtNodeVisitor *node_visitor, GtFeatureNode *feature_node, GtError *err)`

Visit `feature_node` with `node_visitor`.

`int gt_node_visitor_visit_meta_node(GtNodeVisitor *node_visitor, GtMetaNode *meta_node, GtError *err)`

Visit `meta_node` with `node_visitor`.

`int gt_node_visitor_visit_region_node(GtNodeVisitor *node_visitor, GtRegionNode *region_node, GtError *err)`

Visit `region_node` with `node_visitor`.

`int gt_node_visitor_visit_sequence_node(GtNodeVisitor *node_visitor, GtSequenceNode *sequence_node, GtError *err)`

Visit `sequence_node` with `node_visitor`.

`void gt_node_visitor_delete(GtNodeVisitor *node_visitor)`

Delete `node_visitor`.

2.66 Class GtOption

`GtOption` objects represent command line options (which are used in a `GtOptionParser`). Option descriptions are automatically formatted to `GT_OPTION_PARSER_TERMINAL_WIDTH`, but it is possible to embed newlines into the descriptions to manually affect the formatting.

Methods

`GtOption* gt_option_new_bool(const char *option_string, const char *description, bool *value, bool default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_double(const char *option_string, const char *description, double *value, double default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_double_min(const char *option_string, const char *description, double *value, double default_value, double minimum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value`.

`GtOption* gt_option_new_double_min_max(const char *option_string, const char *description, double *value, double default_value, double minimum_value, double maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value` and at most the `maximum_value`.

`GtOption* gt_option_new_probability(const char *option_string, const char *description, double *value, double default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at larger or equal than 0.0 and smaller or equal than 1.0.

`GtOption* gt_option_new_int(const char *option_string, const char *description, int *value, int default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_int_min(const char *option_string, const char *description, int *value, int default_value, int minimum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value`.

`GtOption* gt_option_new_int_max(const char *option_string, const char *description, int *value, int default_value, int maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at most have the `maximum_value`.

`GtOption* gt_option_new_int_min_max(const char *option_string, const char *description, int *value, int default_value, int minimum_value, int maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value` and at most the `maximum_value`.

`GtOption* gt_option_new_uint(const char *option_string, const char *description, unsigned int *value, unsigned int default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_uint_min(const char *option_string, const char *description, unsigned int *value, unsigned int default_value, unsigned int minimum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value`.

`GtOption* gt_option_new_uint_max(const char *option_string, const char *description, unsigned int *value, unsigned int default_value, unsigned int maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at most have the `maximum_value`.

`GtOption* gt_option_new_uint_min_max(const char *option_string, const char *description, unsigned int *value, unsigned int default_value, unsigned int minimum_value, unsigned int maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value` and at most the `maximum_value`.

`GtOption* gt_option_new_long(const char *option_string, const char *description, long *value, long default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_ulong(const char *option_string, const char *description, unsigned long *value, unsigned long default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

GtOption* gt_option_new_ulong_min(const char *option_string, const char *description, unsigned long *value, unsigned long default_value, unsigned long minimum_value)

Return a new GtOption with the given option_string, description, and default_value. The result of the option parsing is stored in value. The argument to this option must at least have the minimum_value.

GtOption* gt_option_new_ulong_min_max(const char *option_string, const char *description, unsigned long *value, unsigned long default_value, unsigned long minimum_value, unsigned long maximum_value)

Return a new GtOption with the given option_string, description, and default_value. The result of the option parsing is stored in value. The argument to this option must at least have the minimum_value and at most the maximum_value.

GtOption* gt_option_new_range(const char *option_string, const char *description, GtRange *value, GtRange *default_value)

Return a new GtOption with the given option_string, description, and default_value. The result of the option parsing is stored in value. If default_value equals NULL, GT_UNDEF_LONG will be used as the default start and end point of value.

GtOption* gt_option_new_range_min_max(const char *option_string, const char *description, GtRange *value, GtRange *default_value, unsigned long minimum_value, unsigned long maximum_value)

Return a new GtOption with the given option_string, description, and default_value. The result of the option parsing is stored in value. The first argument to this option (which will be used as the start) must at least have the minimum_value and the second argument (which will be used as the end) at most the maximum_value.

GtOption* gt_option_new_string(const char *option_string, const char *description, GtStr *value, const char *default_value)

Return a new GtOption with the given option_string, description, and default_value. The result of the option parsing is stored in value.

GtOption* gt_option_new_string_array(const char *option_string, const char *description, GtStrArray *value)

Return a new GtOption with the given option_string, description, and default_value. The result of the option parsing are stored in value.

GtOption* gt_option_new_choice(const char *option_string, const char *description, GtStr *value, const char *default_value, const char **domain)

Return a GtOption with the given option_string, description, and default_value which allows only arguments given in the NULL-terminated domain (default_value must be an entry of domain or NULL).

`GtOption* gt_option_new_filename(const char *option_string, const char *description, GtStr *filename)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing are stored in `value`.

`GtOption* gt_option_new_filename_array(const char *option_string, const char *description, GtStrArray *filename_array)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The results of the option parsing are stored in `value`.

`GtOption* gt_option_new_debug(bool *value)`

Return a new debug `GtOption` object: `-debug`, "enable debugging output", default is `false`. The result of the option parsing is stored in `value`

`GtOption* gt_option_new_verbose(bool *value)`

Return a new verbose `GtOption` object: `-v`, "be verbose", default is `false`. The result of the option parsing is stored in `value`

`GtOption* gt_option_new_width(unsigned long *value)`

Return a new width `GtOption` object: `-width`, "set output width for FASTA sequence printing (0 disables formatting)", default is 0. The result of the option parsing is stored in `value`

`GtOption* gt_option_ref(GtOption *option)`

Increase the reference count for `option` and return it.

`const char* gt_option_get_name(const GtOption * option)`

Return the name of `option`

`void gt_option_is_mandatory(GtOption *option)`

Make `option` mandatory.

`void gt_option_is_mandatory_either(GtOption *option_a, const GtOption *option_b)`

Make it mandatory, that either `option_a` or `option_b` is used.

`void gt_option_is_mandatory_either_3(GtOption *option_a, const GtOption *option_b, const GtOption *option_c)`

Make it mandatory, that one of the options `option_a`, `option_b`, or `option_c` is used.

`void gt_option_is_extended_option(GtOption *option)`

Set that `option` is only shown in the output of `-help+`.

`void gt_option_is_development_option(GtOption *option)`

Set that `option` is only shown in the output of `-helpdev`.

`void gt_option_imply(GtOption *option_a, const GtOption *option_b)`

Make `option_a` imply `option_b`.

void gt_option_imply_either_2(GtOption *option_a, const GtOption *option_b, const GtOption *option_c)

Make option_a imply either option_b or option_c

void gt_option_exclude(GtOption *option_a, GtOption *option_b)

Set that the options option_a and option_b exclude each other.

void gt_option_hide_default(GtOption *option)

Hide the default value of option in -help output.

void gt_option_argument_is_optional(GtOption *option)

Set that the argument to option is optional

bool gt_option_is_set(const GtOption *option)

Return true if option was set, false otherwise.

void gt_option_delete(GtOption*)

Delete option.

int gt_option_parse_spacespec(unsigned long *maximumspace, const char *optname, const GtStr *memlimit, GtError *err)

Parse the argument to option -memlimit. Could be made into a special parser, but I do not know how. SK. 2011-09-19

2.67 Class GtOptionParser

GtOptionParser objects can be used to parse command line options.

Methods

#define GT_OPTION_PARSER_TERMINAL_WIDTH

The default terminal width used in the output of the GtOptionParser.

GtOptionParser* gt_option_parser_new(const char *synopsis, const char *one_liner)

Return a new GtOptionParser object. The synopsis should summarize the command line arguments and mandatory arguments in a single line. The one_liner should describe the program for which the GtOptionParser is used in a single line and must have an upper case letter at the start and a '.' at the end.

void gt_option_parser_add_option(GtOptionParser *option_parser, GtOption *option)

Add option to option_parser. Takes ownership of option.

`GtOption* gt_option_parser_get_option(GtOptionParser *option_parser, const char *option_string)`

Return the `GtOption` object if an option named `option_string` is present in `option_parser`, and `NULL` if no such option exists.

`void gt_option_parser_refer_to_manual(GtOptionParser *option_parser)`

Refer to manual at the end of `-help` output of `opion_parser`.

`void gt_option_parser_set_comment_func(GtOptionParser *option_parser, GtShowCommentFunc comment_func, void *data)`

Set `comment_func` in `option_parser` (`data` is passed along).

`void gt_option_parser_set_version_func(GtOptionParser *option_parser, GtShowVersionFunc version_func)`

Set the version function used by `option_parser` to `version_func`. This version function takes precedence to the one supplied to `gt_option_parser_parse()`.

`void gt_option_parser_set_mail_address(GtOptionParser*, const char *mail_address)`

Set the `mail_address` used in the final "Report bugs to" line of the `-help` output. It should be of the form <`bill@microsoft.com`> (email address enclosed in one pair of angle brackets).

`void gt_option_parser_register_hook(GtOptionParser *option_parser, GtOptionParserHookFunc hook_function, void *data)`

Register a `hook_function` with `option_parser`. All registered hook functions are called at the end of `gt_option_parser_parse()`. This allows one to have a module which registers a bunch of options in the option parser and automatically performs necessary postprocessing after the option parsing has been done via the hook function.

`void gt_option_parser_set_min_args(GtOptionParser *option_parser, unsigned int minimum)`

The the `minimum` number of additional command line arguments `option_parser` must parse in order to succeed.

`void gt_option_parser_set_max_args(GtOptionParser *option_parser, unsigned int maximum)`

The the `maximum` number of additional command line arguments `option_parser` must parse in order to succeed.

`void gt_option_parser_set_min_max_args(GtOptionParser *option_parser, unsigned int minimum, unsigned int maximum)`

The the `minimum` and `maximum` number of additional command line arguments `option_parser` must parse in order to succeed.

`GtOPrval gt_option_parser_parse(GtOptionParser *option_parser, int *parsed_args, int argc, const char **argv, GtShowVersionFunc version_func, GtError *err)`

Use `option_parser` to parse options given in argument vector `argv` (with `argc` many arguments). The number of parsed arguments is stored in `parsed_args`. `version_func` is used for the output of option `-version`. In case of error, `GT_OPTION_PARSER_ERROR` is returned and `err` is set accordingly.

`void gt_option_parser_delete(GtOptionParser *option_parser)`

Delete `option_parser`.

2.68 Class GtPhase

This enum type defines the possible phases. The following phases are defined: `GT_PHASE_ZERO`, `GT_PHASE_ONE`, `GT_PHASE_TWO`, and `GT_PHASE_UNDEFINED`.

Methods

`#define GT_PHASE_CHARS`

Use this string to map phase enum types to their corresponding character.

`GtPhase gt_phase_get(char phase_char)`

Map `phase_char` to the corresponding phase enum type. An assertion will fail if `phase_char` is not a valid one.

2.69 Class GtQueue

`GtQueue` objects are generic queues which can be used to process objects of any type in an First-In-First-Out (FIFO) fashion.

Methods

`GtQueue* gt_queue_new(void)`

Return a new `GtQueue` object.

`void gt_queue_add(GtQueue *queue, void *elem)`

Add `elem` to queue (*enqueue* in computer science terminology).

`void* gt_queue_get(GtQueue *queue)`

Remove the first element from non-empty `queue` and return it (*dequeue* in computer science terminology).

`void* gt_queue_head(GtQueue *queue)`

Return the first element in non-empty `queue` without removing it.

`void gt_queue_remove(GtQueue *queue, void *elem)`

Remove `elem` from `queue` (`elem` has to be in `queue`). Thereby `queue` is traversed in reverse order, leading to O(`gt_queue_size(queue)`) worst-case running time.

`unsigned long gt_queue_size(const GtQueue *queue)`

Return the number of elements in `queue`.

`void gt_queue_delete(GtQueue *queue)`

Delete `queue`. Elements contained in `queue` are not freed!

2.70 Class GtRDBVisitor

The `GtRDBVisitor` interface, a visitor for `GtRDB` objects.

Methods

`int gt_rdb_visitor_visit_sqlite(GtRDBVisitor *rdbv, GtRDBSqlite *rdbs, GtError *err)`

Visit a SQLite database `rdbs` with `rdbv`. Returns 0 on success, a negative value otherwise, and `err` is set accordingly.

`int gt_rdb_visitor_visit_mysql(GtRDBVisitor *rdbv, GtRDBMySQL *rdbm, GtError *err)`

Visit a MySQL database `rdbm` with `rdbv`. Returns 0 on success, a negative value otherwise, and `err` is set accordingly.

`void gt_rdb_visitor_delete(GtRDBVisitor *rdbv)`

Delete `rdbv`.

2.71 Class GtRange

The `GtRange` class is used to represent genomic ranges in *GenomeTools*. Thereby, the `start` must **always** be smaller or equal than the `end`.

Methods

`int gt_range_compare(const GtRange *range_a, const GtRange *range_b)`

Compare `range_a` and `range_b`. Returns 0 if `range_a` equals `range_b`, -1 if `range_a` starts before `range_b` or (for equal starts) `range_a` ends before `range_b`, and 1 else.

`int gt_range_compare_with_delta(const GtRange *range_a, const GtRange *range_b, unsigned long delta)`

Compare `range_a` and `range_b` with given `delta`. Returns 0 if `range_a` equals `range_b` modulo `delta` (i.e., the start and end points of `range_a` and `range_b` are at most `delta` bases apart), -1 if `range_a` starts before `range_b` or (for equal starts) `range_a` ends before `range_b`, and 1 else.

`bool gt_range_overlap(const GtRange *range_a, const GtRange *range_b)`

Returns `true` if `range_a` and `range_b` overlap, `false` otherwise.

`bool gt_range_overlap_delta(const GtRange *range_a, const GtRange *range_b, unsigned long delta)`

Returns `true` if `range_a` and `range_b` overlap **at least** `delta` many positions, `false` otherwise.

`bool gt_range_contains(const GtRange *range_a, const GtRange *range_b)`

Returns `true` if `range_b` is contained in `range_a`, `false` otherwise.

`bool gt_range_within(const GtRange *range, unsigned long point)`

Returns `true` if `point` lies within `range`, `false` otherwise.

`GtRange gt_range_join(const GtRange *range_a, const GtRange *range_b)`

Join `range_a` and `range_b` and return the result.

`GtRange gt_range_offset(const GtRange *range, long offset)`

Transform start and end of `range` by `offset` and return the result.

`unsigned long gt_range_length(const GtRange *range)`

Returns the length of the given `range`.

2.72 Class GtReadmode

This enum type defines the possible reamodes, namely `GT_READMODE_FORWARD`, `GT_READMODE_REVERSE`, `GT_READMODE_COMPL`, and `GT_READMODE_REVCOMPL`.

Methods

`const char* gt_readmode_show(GtReadmode readmode)`

Returns the descriptive string for `readmode`.

`int gt_readmode_parse(const char *string, GtError *err)`

Returns the `GtReadmode` for the description `string`, which must be one of ”fwd”,”rev”,”cpl” or ”rcl”. If `string` does not equal any of them, -1 is returned and `err` is set accordingly.

2.73 Class GtRecMap

A `GtRecMap` object contains a mapping from a 2D coordinate pair which identifies a rectangle in a rendered image to the `GtFeatureNode` it represents. The rectangle is defined by the coordinates of its upper left (“northwest”) and lower right (“southeast”) points.
`GtRecMap` objects are created by an `GtImageInfo` object which is filled during the generation of an image by *AnnotationSketch*.

Methods

`double gt_rec_map_get_northwest_x(const GtRecMap*)`

Retrieve *x* value of the the upper left point of the rectangle.

`double gt_rec_map_get_northwest_y(const GtRecMap*)`

Retrieve *y* value of the the upper left point of the rectangle.

`double gt_rec_map_get_southeast_x(const GtRecMap*)`

Retrieve *x* value of the the lower right point of the rectangle.

`double gt_rec_map_get_southeast_y(const GtRecMap*)`

Retrieve *y* value of the the lower right point of the rectangle.

`const GtFeatureNode* gt_rec_map_get_genome_feature(const GtRecMap*)`

Retrieve `GtFeatureNode` associated with this rectangle.

`bool gt_rec_map_has_omitted_children(const GtRecMap*)`

Returns `true` if the rectangle represents a block root whose elements have not been drawn due to size restrictions.

2.74 Class GtRegionMapping

A `GtRegionMapping` objects maps sequence-regions to the corresponding entries of sequence files.

Methods

`GtRegionMapping* gt_region_mapping_new_mapping(GtStr *mapping_filename, GtError *err)`

Return a new `GtRegionMapping` object for the mapping file with the given `mapping_filename`. In the case of an error, `NULL` is returned and `err` is set accordingly.

`GtRegionMapping* gt_region_mapping_new_seqfiles(GtStrArray *sequence_filenames, bool matchdesc, bool usedesc)`

Return a new `GtRegionMapping` object for the sequence files given in `sequence_filenames`. If `matchdesc` is `true`, the sequence descriptions from the input files are matched for the desired sequence IDs (in GFF3).
If `usedesc` is `true`, the sequence descriptions are used to map the sequence IDs (in GFF3) to actual sequence entries. If a description contains a sequence range (e.g., III:1000001..2000000), the first part is used as sequence ID ('III') and the first range position as offset ('1000001').
`matchdesc` and `usedesc` cannot be `true` at the same time.

`GtRegionMapping* gt_region_mapping_new_rawseq(const char *rawseq, unsigned long length, unsigned long offset)`

Return a new `GtRegionMapping` object which maps to the given sequence `rawseq` with the corresponding `length` and `offset`.

`GtRegionMapping* gt_region_mapping_ref(GtRegionMapping *region_mapping)`

Increase the reference count for `region_mapping` and return it.

`int gt_region_mapping_get_raw_sequence(GtRegionMapping *region_mapping, const char **rawseq, unsigned long *length, unsigned long *offset, GtStr *seqid, const GtRange *range, GtError *err)`

Use `region_mapping` to map the given sequence ID `seqid` and its corresponding `range` to an actual sequence. The sequence is returned in `rawseq`, its length and offset in `length` and `offset`. In the case of an error, -1 is returned and `err` is set accordingly.

`int gt_region_mapping_get_description(GtRegionMapping *region_mapping, GtStr *desc, GtStr *seqid, GtError *err)`

Use `region_mapping` to get the description of the MD5 sequence ID `seqid`. The description is appended to `desc`. In the case of an error, -1 is returned and `err` is set accordingly.

`const char* gt_region_mapping_get_md5_fingerprint(GtRegionMapping *region_mapping, GtStr *seqid, const GtRange *range, unsigned long *offset, GtError *err)`

Use `region_mapping` to return the MD5 fingerprint of the sequence with the sequence ID `seqid` and its corresponding `range`. The offset of the sequence is stored in `offset`. In the case of an error, `NULL` is returned and `err` is set accordingly.

`void gt_region_mapping_delete(GtRegionMapping *region_mapping)`

Delete `region_mapping`.

2.75 Class GtRegionNode

Implements the `GtGenomeNode` interface. Region nodes correspond to the `##sequence-region` lines in GFF3 files.

Methods

`GtGenomeNode* gt_region_node_new(GtStr *seqid, unsigned long start, unsigned long end)`

Create a new `GtRegionNode*` representing sequence with ID `seqid` from base position `start` to base position `end` (1-based). `start` has to be smaller or equal than `end`. The `GtRegionNode*` stores a new reference to `seqid`, so make sure you do not modify the original `seqid` afterwards!

2.76 Class GtSelectStream

Implements the `GtNodeStream` interface. A `GtSelectStream` selects certain nodes it retrieves from its node source and passes them along.

Methods

`GtNodeStream* gt_select_stream_new(GtNodeStream *in_stream, GtStr *seqid, GtStr *source, const GtRange *contain_range, const GtRange *overlap_range, GtStrand strand, GtStrand targetstrand, bool has_CDS, unsigned long max_gene_length, unsigned long max_gene_num, double min_gene_score, double max_gene_score, double min_average_splice_site_prob, unsigned long feature_num, GtStrArray *select_files, GtStr *select_logic, GtError *err)`

Create a `GtSelectStream` object which selects genome nodes it retrieves from its `in_stream` and passes them along if they meet the criteria defined by the other arguments. All comment nodes are selected. If `seqid` is defined, a genome node must have it to be selected. If `source` is defined, a genome node must have it to be selected. If `contain_range` is defined, a genome node must be contained in it to be selected. If `overlap_range` is defined, a genome node must overlap it to be selected. If `strand` is defined, a (top-level) genome node must have it to be selected. If `targetstrand` is defined, a feature with a target attribute must have exactly one of it and its strand must equal `targetstrand`. If `had_cds` is `true`, all top-level features are selected which have a child with type *CDS*. If `max_gene_length` is defined, only genes up to the this length are selected. If `max_gene_num` is defined, only so many genes are selected. If `min_gene_score` is defined, only genes with at least this score are selected. If `max_gene_score` is defined, only genes with at most this score are selected. If `min_average_splice_site_prob` is defined, feature nodes which have splice sites must have at least this average splice site score to be selected. If `feature_num` is defined, just the `feature_num`th feature node occurring in the `in_stream` is selected. If `select_files` is defined and has at least one entry, the entries are evaluated as Lua scripts containing functions taking `GtGenomeNodes` that are evaluated to boolean values to determine selection. `select_logic` can be ”OR” or ”AND”, defining how the results from the select scripts are combined. Returns a pointer to a new `GtSelectStream` or NULL on error (`err` is set accordingly).

`void gt_select_stream_set_drophandler(GtSelectStream *sstr, GtSelectNodeFunc fp, void *data)`

Sets `fp` as a handler function to be called for every `GtGenomeNode` not selected by `sstr`. The void pointer `data` can be used for arbitrary user data.

2.77 Class GtSequenceNode

Implements the `GtGenomeNode` interface. Sequence nodes correspond to singular embedded FASTA sequences in GFF3 files.

Methods

`GtGenomeNode* gt_sequence_node_new(const char *description, GtStr *sequence)`

Create a new `GtSequenceNode*` representing a FASTA entry with the given `description` and `sequence`. Takes ownership of `sequence`.

`const char* gt_sequence_node_get_description(const GtSequenceNode *sequence_node)`

Return the description of `sequence_node`.

`const char* gt_sequence_node_get_sequence(const GtSequenceNode *sequence_node)`

Return the sequence of `sequence_node`.

`unsigned long gt_sequence_node_get_sequence_length(const GtSequenceNode *sequence_node)`

Return the sequence length of `sequence_node`.

2.78 Class GtSortStream

Implements the `GtNodeStream` interface. A `GtSortStream` sorts the `GtGenomeNode` objects it retrieves from its node source.

Methods

`GtNodeStream* gt_sort_stream_new(GtNodeStream *in_stream)`

Create a `GtSortStream*` which sorts the genome nodes it retrieves from `in_stream` and returns them unmodified, but in sorted order.

2.79 Class GtSplitter

The `GtSplitter` class defines objects which can split given strings into tokens delimited by a given character, allowing for convenient access to each token.

Methods

`GtSplitter* gt_splitter_new(void)`

Create a new `GtSplitter` object.

`void gt_splitter_split(GtSplitter *splitter, char *string, unsigned long length, char delimiter)`

Use `splitter` to split `string` of given `length` into tokens delimited by `delimiter`. Note that `string` is modified in the splitting process!

`char** gt_splitter_get_tokens(GtSplitter *splitter)`

Return all tokens split by `splitter` in an array.

`char* gt_splitter_get_token(GtSplitter *splitter, unsigned long token_num)`

Return token with number `token_num` from `splitter`.

`void gt_splitter_reset(GtSplitter *splitter)`

Reset the `splitter`.

`unsigned long gt_splitter_size(GtSplitter *splitter)`

Return the number of tokens in `splitter`.

`void gt_splitter_delete(GtSplitter *splitter)`

Delete the `splitter`.

2.80 Class GtStatStream

Implements the `GtNodeStream` interface. A `GtStatStream` gathers statistics about the `GtGenomeNode` objects it retrieves from its node source and passes them along unmodified.

Methods

`GtNodeStream* gt_stat_stream_new(GtNodeStream *in_stream, bool gene_length_distribution, bool gene_score_distribution, bool exon_length_distribution, bool exon_number_distribution, bool intron_length_distribution, bool cds_length_distribution, bool used_sources)`

Create a `GtStatStream` object which gathers statistics about the `GtGenomeNode` objects it retrieves from its `in_stream` and returns them unmodified. Besides the basic statistics, statistics about the following distributions can be gathered, if the corresponding argument equals `true`: `gene_length_distribution`, `gene_score_distribution`, `exon_length_distribution`, `exon_number_distribution`, `intron_length_distribution`, `cds_length_distribution`.
If `used_sources` equals `true`, it is recorded which source tags have been encountered.

`void gt_stat_stream_show_stats(GtStatStream *stat_stream, GtFile *outfp)`

Write the statistics gathered by `stat_stream` to `outfp`.

2.81 Class GtStr

Objects of the `GtStr` class are strings which grow on demand.

Methods

`GtStr* gt_str_new(void)`

Return an empty `GtStr` object.

`GtStr* gt_str_new_cstr(const char *cstr)`

Return a new `GtStr` object whose content is set to `cstr`.

`GtStr* gt_str_clone(const GtStr *str)`

Return a clone of `str`.

`GtStr* gt_str_ref(GtStr *str)`

Increase the reference count for `str` and return it. If `str` is `NULL`, `NULL` is returned without any side effects.

`char* gt_str_get(const GtStr *str)`

Return the content of `str`. Never returns NULL, and the content is always \0-terminated

`void gt_str_set(GtStr *str, const char *cstr)`

Set the content of `str` to `cstr`.

`void gt_str_append_str(GtStr *dest, const GtStr *src)`

Append the string `src` to `dest`.

`void gt_str_append_cstr(GtStr *str, const char *cstr)`

Append the \0-terminated `cstr` to `str`.

`void gt_str_append_cstr_nt(GtStr *str, const char *cstr, unsigned long length)`

Append the (not necessarily \0-terminated) `cstr` with given `length` to `str`.

`void gt_str_append_char(GtStr *str, char c)`

Append character `c` to `str`.

`void gt_str_append_double(GtStr *str, double d, int precision)`

Append double `d` to `str` with given `precision`.

`void gt_str_append_ulong(GtStr *str, unsigned long ulong)`

Append `ulong` to `str`.

`void gt_str_append_int(GtStr *str, int intval)`

Append `intval` to `str`.

`void gt_str_append_uint(GtStr *str, unsigned int uint)`

Append `uint` to `str`.

`void gt_str_set_length(GtStr *str, unsigned long length)`

Set length of `str` to `length`. `length` must be smaller or equal than `gt_str_length(str)`.

`void gt_str_reset(GtStr *str)`

Reset `str` to length 0.

`int gt_str_cmp(const GtStr *str1, const GtStr *str2)`

Compare `str1` and `str2` and return the result (similar to `strcmp(3)`).

`unsigned long gt_str_length(const GtStr *str)`

Return the length of `str`. If `str` is `NULL`, 0 is returned.

`void gt_str_delete(GtStr *str)`

Decrease the reference count for `str` or delete it, if this was the last reference.

2.82 Class GtStrArray

`GtStrArray*` objects are arrays of string which grow on demand.

Methods

`GtStrArray* gt_str_array_new(void)`

Return a new `GtStrArray` object.

`GtStrArray* gt_str_array_ref(GtStrArray*)`

Increases the reference to a GtStrArray.

`void gt_str_array_add_cstr(GtStrArray *str_array, const char *cstr)`

Add `cstr` to `str_array`. Thereby, an internal copy of `cstr` is created.

`void gt_str_array_add_cstr_nt(GtStrArray *str_array, const char *cstr, unsigned long length)`

Add the non `\0`-terminated `cstr` with given `length` to `str_array`. Thereby, an internal copy of `cstr` is created.

`void gt_str_array_add(GtStrArray *str_array, const GtStr *str)`

Add `str` to `str_array`. Thereby, an internal copy of `str` is created.

`const char* gt_str_array_get(const GtStrArray *str_array, unsigned long strnum)`

Return pointer to internal string with number `strnum` of `str_array`. `strnum` must be smaller than `gt_str_array_size(str_array)`.

`void gt_str_array_set_cstr(GtStrArray *str_array, unsigned long strnum, const char *cstr)`

Set the string with number `strnum` in `str_array` to `cstr`.

`void gt_str_array_set(GtStrArray *str_array, unsigned long strnum, const GtStr *str)`

Set the string with number `strnum` in `str_array` to `str`.

`void gt_str_array_set_size(GtStrArray *str_array, unsigned long size)`

Set the size of `str_array` to `size`. `size` must be smaller or equal than `gt_str_array_size(str_array)`.

`void gt_str_array_reset(GtStrArray *str_array)`

Set the size of `str_array` to 0.

`unsigned long gt_str_array_size(const GtStrArray *str_array)`

Return the number of strings stored in `str_array`.

`void gt_str_array_delete(GtStrArray *str_array)`

Delete `str_array`.

2.83 Class GtStrand

This enum type defines the possible strands. The following strands are defined: `GT_STRAND_FORWARD`, `GT_STRAND_REVERSE`, `GT_STRAND_BOTH`, and `GT_STRAND_UNKNOWN`.

Methods

`#define GT_STRAND_CHARS`

Use this string to map strand enum types to their corresponding character.

`GtStrand gt_strand_get(char strand_char)`

Map `strand_char` to the corresponding strand enum type. Returns `GT_NUM_OF_STRAND_TYPES` if `strand_char` is not a valid one.

2.84 Class GtStyle

Objects of the `GtStyle` class hold *AnnotationSketch* style information like colors, margins, collapsing options, and others. The class provides methods to set values of various types. Each value is organized into a *section* and is identified by a *key*. That is, a *section*, *key* pair must uniquely identify a value.

Methods

`GtStyle* gt_style_new(GtError*)`

Creates a new `GtStyle` object.

`GtStyle* gt_style_ref(GtStyle*)`

Increments the reference count of the given `GtStyle`.

`void gt_style_unsafe_mode(GtStyle*)`

Enables unsafe mode (“io” and “os” libraries loaded).

`void gt_style_safe_mode(GtStyle*)`

Enables safe mode ("io" and "os" libraries not accessible).

`bool gt_style_is_unsafe(GtStyle *sty)`

Returns true if `sty` is in unsafe mode.

`GtStyle* gt_style_clone(const GtStyle*, GtError*)`

Creates a independent ("deep") copy of the given `GtStyle` object.

`int gt_style_load_file(GtStyle*, const char *filename, GtError*)`

Loads and executes Lua style file with given `filename`. This file must define a global table called *style*.

`int gt_style_load_str(GtStyle*, GtStr *instr, GtError*)`

Loads and executes Lua style code from the given `GtStr` `instr`. This code must define a global table called *style*.

`int gt_style_to_str(const GtStyle*, GtStr *outstr, GtError*)`

Generates Lua code which represents the given `GtStyle` object and writes it into the `GtStr` object `outstr`.

`void gt_style_reload(GtStyle*)`

Reloads the Lua style file.

`void gt_style_set_color(GtStyle*, const char *section, const char *key, const GtColor *color)`

Sets a color value in the `GtStyle` for section `section` and `key` to a certain `color`.

`GtStyleQueryStatus gt_style_get_color(const GtStyle *style, const char *section, const char *key, GtColor *result, GtFeatureNode *fn, GtError *err)`

Retrieves a color value from `style` for key `key` in section `section`. The color is written to the location pointed to by `result`. Optionally, a feature node pointer `fn` can be specified for handling in node-specific callbacks. Because color definitions can be functions, `gt_style_get_color()` can fail at runtime. In this case, this function returns GT_STYLE_QUERY_ERROR and `err` is set accordingly. If the color was not specified in `style`, a grey default color is written to `result` and GT_STYLE_QUERY_NOT_SET is returned so the caller can provide a custom default. In case of successful retrieval of an existing color, GT_STYLE_QUERY_OK is returned.

`void gt_style_set_str(GtStyle*, const char *section, const char *key, GtStr *value)`

Set string with key `key` in `section` to `value`.

`void gt_style_set_num(GtStyle*, const char *section, const char *key, double number)`

Set numeric value of key `key` in `section` to `number`.

`void gt_style_set_bool(GtStyle*, const char *section, const char *key, bool val)`

Set boolean value of key `key` in `section` to `val`.

`void gt_style_unset(GtStyle*, const char *section, const char *key)`

Unset value of key `key` in `section`.

`void gt_style_delete(GtStyle *style)`

Deletes this `style`.

2.85 Class GtTagValueMap

A very simple tag/value map absolutely optimized for space (i.e., memory consumption) on the cost of time. Basically, each read/write access costs O(n) time, whereas n denotes the accumulated length of all tags and values contained in the map. Tags and values cannot have length 0. The implementation as a char* shines through (also to save one additional memory allocation), therefore the usage is a little bit different compared to other *GenomeTools* classes. See the implementation of `gt_tag_value_map_example()` for an ussage example.

Methods

`GtTagValueMap gt_tag_value_map_new(const char *tag, const char *value)`

Return a new `GtTagValueMap` object which stores the given `tag`/`value` pair.

`void gt_tag_value_map_add(GtTagValueMap *tag_value_map, const char *tag, const char *value)`

Add `tag`/`value` pair to `tag_value_map`. `tag_value_map` must not contain the given `tag` already!

`void gt_tag_value_map_set(GtTagValueMap *tag_value_map, const char *tag, const char *value)`

Set the given `tag` in `tag_value_map` to `value`.

`const char* gt_tag_value_map_get(const GtTagValueMap tag_value_map, const char *tag)`

Return value corresponding to `tag` from `tag_value_map`. If `tag_value_map` does not contain such a value, `NULL` is returned.

`void gt_tag_value_map_remove(GtTagValueMap *tag_value_map, const char *tag)`

Removes the given `tag` from `tag_value_map`. `tag_value_map` must contain the given `tag` already!

`void gt_tag_value_map_foreach(const GtTagValueMap tag_value_map, GtTagValueMapIteratorFunc iterator_func, void *data)`

Apply `iterator_func` to each tag/value pair contained in `tag_value_map` and pass `data` along.

`int gt_tag_value_map_example(GtError *err)`

Implements an example useage of a tag/value map.

`void gt_tag_value_map_delete(GtTagValueMap tag_value_map)`

Delete `tag_value_map`.

2.86 Class GtTextWidthCalculator

The GtTextWidthCalculator interface answers queries w.r.t. text width in a specific drawing backend. This interface is needed to do proper line breaking in a `GtLayout` even if there is no `GtCanvas` or `GtGraphics` created yet.

Methods

`GtTextWidthCalculator* gt_text_width_calculator_ref(GtTextWidthCalculator*)`

Increases the reference count of the `GtTextWidthCalculator`.

`double gt_text_width_calculator_get_text_width(GtTextWidthCalculator*, const char *text, GtError *err)`

Requests the width of `text` from the `GtTextWidthCalculator`. If the returned value is negative, an error occurred. Otherwise, a positive double value is returned.

`void gt_text_width_calculator_delete(GtTextWidthCalculator*)`

Deletes a `GtTextWidthCalculator` instance.

2.87 Class GtTextWidthCalculatorCairo

Implements the GtTextWidthCalculator interface with Cairo as the drawing backend. If text width is to be calculated with regard to a specific transformation etc. which is in effect in a `cairo_t` and which should be used later via a `GtCanvasCairoContext`, create a `GtTextWidthCalculatorCairo` object and pass it to the `GtLayout` via `gt_layout_new_with_twc()`.

Methods

`GtTextWidthCalculator* gt_text_width_calculator_cairo_new(cairo_t*, GtStyle*, GtError*)`

Creates a new `GtTextWidthCalculatorCairo` object for the given context using the text size options given in the `GtStyle`. If the `GtStyle` is NULL, the current font settings in the `cairo_t` will be used for all text width calculations.

2.88 Class GtTimer

The `GtTimer` class encapsulates a timer which can be used for run-time measurements.

Methods

`GtTimer* gt_timer_new(void)`

Return a new `GtTimer` object.

`GtTimer* gt_timer_new_with_progress_description(const char* description)`

Return a new `GtTimer` object with the first `description`.

`void gt_timer_start(GtTimer *t)`

Start the time measurement on `t`.

`void gt_timer_stop(GtTimer *t)`

Stop the time measurement on `t`.

`void gt_timer_show(GtTimer *t, FILE *fp)`

Output the current state of `t` in the format ”pointer `fp` (see `gt_timer_show_formatted`). The timer is then stopped.

`void gt_timer_show_formatted(GtTimer *t, const char *fmt, FILE *fp)`

Output the current state of `t` in a user-defined format given by `fmt`. `fmt` must be a format string for four elapsed seconds, elapsed microseconds, used usertime in seconds, system time in seconds. The output is written to `fp`.

`void gt_timer_show_progress(GtTimer *t, const char *desc, FILE *fp)`

Output the current state of `t` on `fp` since the last call of `gt_timer_show_progress()` or the last start of `t`, along with the current description. The timer is not stopped, but updated with `desc` to be the next description.

`void gt_timer_show_progress_final(GtTimer *t, FILE *fp)`

Output the overall time measured with `t` from start to now on `fp`.

`void gt_timer_show_cpu_time_by_progress(GtTimer *t)`

Show also user and sys time in output of gt_timer_show_progress[_final]

`void gt_timer_omit_last_stage(GtTimer *t)`

Hide output of last stage time in gt_timer_show_progress_final

`void gt_timer_delete(GtTimer *t)`

Delete `t`.

2.89 Class GtTransTable

`GtStrArray* gt_trans_table_get_scheme_descriptions(void)`

Returns a `GtStrArray` of translation scheme descriptions, each of the format ”`gt_translator_set_translation_scheme()` and the string is the scheme name.

`GtTransTable* gt_trans_table_new(unsigned int scheme, GtError *err)`

Returns a translation table as given by `scheme` which refers to the numbers as reported by `gt_translator_get_translation_table_descriptions()` or the list given at the NCBI web site *http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi*. Returns NULL if an error occurred, see `err` for details.

`GtTransTable* gt_trans_table_new_standard(GtError *err)`

Returns the standard translation table.

`const char* gt_trans_table_description(const GtTransTable *tt)`

Returns the description of `tt`.

`int gt_trans_table_translate_codon(const GtTransTable *tt, char c1, char c2, char c3, char *amino, GtError *err)`

Writes the translation for the codon `c1`,`c2`,`c3` to the position pointed to by `amino`. The current translation scheme set in `translator` is used. Returns a negative value if an error occurred, see `err` for details. Otherwise, 0 is returned.

`void gt_trans_table_delete(GtTransTable *tt)`

Deletes `tt`.

2.90 Class GtTranslator

The `GtTranslator` can be used to produce 3-frame translations of DNA sequences via an iterator interface.

Methods

`GtTranslator* gt_translator_new_with_table(GtTransTable *tt, GtCodonIterator *ci)`

Creates a new `GtTranslator`, starting its translation at the current position of `ci`. The current reading frame is also taken from the state of `ci`. The translation table `tt` is used.

`GtTranslator* gt_translator_new(GtCodonIterator *ci)`

Creates a new `GtTranslator`, starting its translation at the current position of `ci`. The current reading frame is also taken from the state of `ci`. The standard translation table is used.

`void gt_translator_set_codon_iterator(GtTranslator *translator, GtCodonIterator *ci)`

Reinitializes `translator` with the position and frame status as given in `ci`.

`void gt_translator_set_translation_table(GtTranslator *translator, GtTransTable *tt)`

Selects the translation scheme in `translator` to the one identified by translation table `tt`.

`GtTranslatorStatus gt_translator_next(GtTranslator *translator, char *translated, unsigned int *frame, GtError *err)`

Returns the translation of the next codon. The currently translated character is put in `translated` while the current reading frame is put in `frame`. Returns GT_TRANSLATOR_ERROR if an error occurred, see `err` for details. If the end of the sequence region to translate has been reached, GT_TRANSLATOR_END is returned. Otherwise, GT_TRANSLATOR_OK (equal to 0) is returned.

`GtTranslatorStatus gt_translator_find_startcodon(GtTranslator *translator, unsigned long *pos, GtError *err)`

Moves the `translator` to the beginning of the first codon in `dnaseq` (of length `dnalen`) which is a start codon according to the selected translation scheme in `translator`. The offset is written to the location pointed to by `pos`. Returns GT_TRANSLATOR_ERROR if an error occurred, see `err` for details. If the end of the sequence region to scan has been reached without finding a start codon, GT_TRANSLATOR_END is returned. Otherwise, GT_TRANSLATOR_OK (equal to 0) is returned.

`GtTranslatorStatus gt_translator_find_stopcodon(GtTranslator *translator, unsigned long *pos, GtError *err)`

Moves the `translator` to the beginning of the first codon in `dnaseq` (of length `dnalen`) which is a stop codon according to the selected translation scheme in `translator`. The offset is written to the location pointed to by `pos`. Returns GT_TRANSLATOR_ERROR if an error occurred, see `err` for details. If the end of the sequence region to scan has been reached without finding a stop codon, GT_TRANSLATOR_END is returned. Otherwise, GT_TRANSLATOR_OK (equal to 0) is returned.

`GtTranslatorStatus gt_translator_find_codon(GtTranslator *translator, GtStrArray *codons, unsigned long *pos, GtError *err)`

Moves the `translator` to the beginning of the first codon in `dnaseq` (of length `dnalen`) which belongs to the set of codons specified in `codons`. The offset is written to the location pointed to by `pos`. Returns GT_TRANSLATOR_ERROR if an error occurred, see `err` for details. If the end of the sequence region to scan has been reached without finding one of the codons, GT_TRANSLATOR_END is returned. Otherwise, GT_TRANSLATOR_OK (equal to 0) is returned.

`void gt_translator_delete(GtTranslator *translator)`

Delete `translator`.

2.91 Class GtTypeChecker

The `GtTypeChecker` interface, allows one to check the validity of (genome feature) types.

Methods

`GtTypeChecker* gt_type_checker_ref(GtTypeChecker *type_checker)`

Increase the reference count for `type_checker` and return it.

`bool gt_type_checker_is_valid(GtTypeChecker *type_checker, const char *type)`

Return `true` if `type` is a valid type for the given `type_checker`, `false` otherwise.

`void gt_type_checker_delete(GtTypeChecker *type_checker)`

Decrease the reference count for `type_checker` or delete it, if this was the last reference.

2.92 Class GtTypeCheckerOBO

Implements the `GtTypeChecker` interface with types from an OBO file.

Methods

`GtTypeChecker* gt_type_checker_obo_new(const char *obo_file_path, GtError *err)`

Create a new `GtTypeChecker*` for OBO file with given `obo_file_path`. If the OBO file cannot be parsed correctly, `NULL` is returned and `err` is set correspondingly.

2.93 Class GtUniqStream

Implements the `GtNodeStream` interface. A `GtUniqStream` filters out repeated features it retrieves from its node source.

Methods

`GtNodeStream* gt_uniq_stream_new(GtNodeStream*)`

Create a `GtUniqStream` object which filters out repeated feature node graphs it retrieves from the sorted `in_stream` and return all other nodes. Two feature node graphs are considered to be *repeated* if they have the same depth-first traversal and each corresponding feature node pair is similar according to the `gt_feature_node_is_similar()` method. For such a repeated feature node graph the one with the higher score (of the top-level feature) is kept. If only one of the feature node graphs has a defined score, this one is kept.

2.94 Class GtVisitorStream

Implements the `GtNodeStream` interface.

Methods

`GtNodeStream* gt_visitor_stream_new(GtNodeStream *in_stream, GtNodeVisitor *node_visitor)`

Create a new `GtVisitorStream*`, takes ownership of `node_visitor`. This stream applies `node_visitor` to each node which passes through it. Can be used to implement all streams with such a functionality.

2.95 Module Array2dim

`#define gt_array2dim_malloc(ARRAY2DIM, ROWS, COLUMNS)`

Allocates a new 2-dimensional array with dimensions `ROWS` x `COLUMNS` and assigns a pointer to the newly allocated space to `ARRAY2DIM`. The size of each element is determined automatically from the type of the `ARRAY2DIM` pointer.

`#define gt_array2dim_calloc(ARRAY2DIM, ROWS, COLUMNS)`

Allocates a new 2-dimensional array with dimensions `ROWS` x `COLUMNS` and assigns a pointer to the newly allocated space to `ARRAY2DIM`. The allocated space is initialized to be filled with zeroes. The size of each element is determined automatically from the type of the `ARRAY2DIM` pointer.

`int gt_array2dim_example(GtError*)`

An example for usage of the `Array2dim` module.

`#define gt_array2dim_delete(ARRAY2DIM)`

Frees the space allocated for the 2-dimensional array pointed to by `ARRAY2DIM`.

2.96 Module Assert

`#define gt_assert(expression)`

The `gt_assert()` macro tests the given `expression` and if it is false, the calling process is terminated. A diagnostic message is written to `stderr` and the `exit(3)` function is called (with error code 2 as argument), effectively terminating the program. If `expression` is true, the `gt_assert()` macro does nothing.

2.97 Module Bsearch

`void* gt_bsearch_data(const void *key, const void *base, size_t nmemb, size_t size, GtCompareWithData, void *data)`

Similar interface to `bsearch(3)`, except that the `GtCompareWithData` function gets an additional `data` pointer.

`void gt_bsearch_all(GtArray *members, const void *key, const void *base, size_t nmemb, size_t size, GtCompareWithData, void *data)`

Similar interface to`gt_bsearch_data()`, except that all members which compare as equal are stored in the `members` array. The order in which the elements are added is undefined.

`void gt_bsearch_all_mark(GtArray *members, const void *key, const void *base, size_t nmemb, size_t size, GtCompareWithData, void *data, GtBittab*)`

Similar interface to`gt_bsearch_all()`. Additionally, if a bittab is given (which must be of size `nmemb`), the bits corresponding to the found elements are marked (i.e., set).

2.98 Module Countingsort

`void gt_countingsort(void *out, const void *in, size_t elem_size, unsigned long size, unsigned long max_elemvalue, void *data, GtGetElemvalue get_elemvalue)`

Sort the array of elements pointed to by `in` containing `size` many elements of size `elem_size` and store the result in the array `out` of the same size. `max_elemvalue` denotes the maximum value an element can have. `get_elemvalue` should return an integer value for the given element `elem`.
Implements the counting sort algorithm. For a description see for example page 175 to page 177 of the book:
T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms*. MIT Press: Cambridge, MA, 1990.

`unsigned long gt_countingsort_get_max(const void *in, size_t elem_size, unsigned long size, void *data, GtGetElemvalue get_elemvalue)`

If `max_elemvalue` is not known, it can be determined with this function.

2.99 Module Cstr

`char* gt_cstr_dup(const char *cstr)`

Creates a duplicate of string `cstr` using the GenomeTools memory allocator.

`char** gt_cstr_split(const char *cstr, char sep)`

Splits the \0-terminated `cstr` at all positions where `sep` occurs and returns a C string array in which each element is a separate string between the occurrences of `sep`. The string array is terminated by NULL. The caller is responsible to free the result.

`char* gt_cstr_dup_nt(const char *cstr, unsigned long length)`

Creates a duplicate of string `cstr` using the GenomeTools memory allocator. The string needs not be \0-terminated, instead its `length` must be given.

`void gt_cstr_rep(char *cstr, char f, char t)`

Replace each occurence of `f` in `cstr` to `t`.

`void gt_cstr_show(const char *cstr, unsigned long length, FILE *outfp)`

Outputs the first `length` characters of the string `cstr` to file pointer `outfp`.

`unsigned long gt_cstr_length_up_to_char(const char *cstr, char c)`

Returns the length of the prefix of `cstr` ending just before `c`, if `cstr` does not contain `c`, strlen(cstr) is returned.

`char* gt_cstr_rtrim(char* cstr, char remove)`

Removes all occurrences of `remove` from the right end of `cstr`.

2.100 Module Endianess

`bool gt_is_little_endian(void)`

Returns `true` if host CPU is little-endian, `false` otherwise.

2.101 Module Fileutils

`bool gt_file_exists(const char *path)`

Returns true if the file with the given `path` exists, false otherwise.

`bool gt_file_is_newer(const char *a, const char *b)`

Returns true if the file with path `a` has a later modification time than the file with path `b`, false otherwise.

`unsigned long gt_file_number_of_lines(const char*)`

Returns the number of lines in a file.

`const char* gt_file_suffix(const char *path)`

Returns the suffix of `path`, if there is any. Returns ”” otherwise. The suffix is the part after and including the last ’.’ but after the last ’/’. Except if `path` ends with ”.gz” or ”.bz2”, then the suffix is the part after and including the second last ’.’.

`void gt_file_dirname(GtStr *path, const char *file)`

Set `path` to the dirname of `file`, if it has one, to ”” otherwise.

`int gt_file_find_in_path(GtStr *path, const char *file, GtError*)`

Find `file` in *PATH*,*ifithasnodirname*;*set*`path`*todirnameotherwise*.*Sets*`path`*totheemptystringif*`file`*coula*

`int gt_file_find_in_env(GtStr *path, const char *file, const char *env, GtError*)`

Find `file` in the ’:’-separated directory list specified in environment variable `env`,*ifithasnodirname*;*set*`path`*todirnameotherwise*.*Sets*`path`*totheemptystringif*`file`*couldnotbefoundine*

`off_t gt_file_estimate_size(const char *file)`

Return the (estimated) size of `file`. If `file` is uncompressed, the exact size is returned. If `file` is compressed, an estimation which assumes that `file` contains a DNA sequence is returned.

`off_t gt_files_estimate_total_size(const GtStrArray *filenames)`

Return the (estimated) total size of all files given in `filenames`. Uses `gt_file_estimate_size()`.

`int gt_files_guess_if_protein_sequences(const GtStrArray *filenames, GtError *err)`

Guesse if the sequences contained in the files given in `filenames` are protein sequences. Returns 1 if the guess is that the files contain protein sequences. Returns 0 if the guess is that the files contain DNA sequences. Returns -1 if an error occurs while reading the files (`err` is set accordingly).

2.102 Module FunctionPointer

`int (*GtCompare)(const void *a, const void *b)`

Functions of this type return less than 0 if `a` is *smaller* than `b`, 0 if `a` is *equal* to `b`, and greater 0 if `a` is *larger* than `b`. Thereby, the operators *smaller*, *equal*, and *larger* are implementation dependent. Do not count on these functions to return -1, 0, or 1!

`int (*GtCompareWithData)(const void*, const void*, void *data)`

Similar to `GtCompare`, but with an additional `data` pointer.

`void (*GtFree)(void*)`

The generic free function pointer type.

2.103 Module Grep

`int gt_grep(bool *match, const char *pattern, const char *line, GtError*)`

Set `match` to `true` if `pattern` matches `line`, to `false` otherwise.

2.104 Module Init

`void gt_lib_init(void)`

Initialize this *GenomeTools* library instance. This has to be called before the library is used!

`void gt_lib_reg_atexit_func(void)`

Registers exit function which calls `gt_lib_clean()` at exit.

`int gt_lib_clean(void)`

Returns 0 if no memory map, file pointer, or memory has been leaked and a value != 0 otherwise.

2.105 Module Log

`void gt_log_enable(void)`

Enable logging.

`bool gt_log_enabled(void)`

Returns true if logging is enabled, false otherwise

`void gt_log_log(const char *format, ...)`

Prints the log message obtained from format and following parameters according if logging is enabled. The logging output is prefixed with the string ”debug: ” and finished by a newline.

`void gt_log_vlog(const char *format, va_list)`

Prints the log message obtained from format and following parameter according to if logging is enabled analog to`gt_log_log()`. But in contrast to `gt_log_log()``gt_log_vlog()` does not accept individual arguments but a single va_list argument instead.

`FILE* gt_log_fp(void)`

Return logging file pointer.

`void gt_log_set_fp(FILE *fp)`

Set logging file pointer to `fp`.

2.106 Module MemoryAllocation

`#define gt_malloc(size)`

Allocate **uninitialized** space for an object whose size is specified by `size` and return it. Besides the fact that it never returns `NULL` analog to `malloc(3)`.

`#define gt_calloc(nmemb, size)`

Allocate contiguous space for an array of `nmemb` objects, each of whose size is `size`. The space is initialized to zero. Besides the fact that it never returns `NULL` analog to `calloc(3)`.

`#define gt_realloc(ptr, size)`

Change the size of the object pointed to by `ptr` to `size` bytes and return a pointer to the (possibly moved) object. Besides the fact that it never returns `NULL` analog to `realloc(3)`.

`#define gt_free(ptr)`

Free the space pointed to by `ptr`. If `ptr` equals `NULL`, no action occurs. Analog to `free(3)`.

`void gt_free_func(void *ptr)`

Analog to `gt_free()`, but usable as a function pointer.

2.107 Module Msort

`void gt_msort(void *base, size_t nmemb, size_t size, GtCompare compar)`

Sorts an array of `nmemb` elements, each of size `size`, according to compare function `compar`. Uses the merge sort algorithm, the interface equals `qsort(3)`.

`void gt_msort_r(void *base, size_t nmemb, size_t size, void *comparinfo, GtCompareWithData compar)`

Identical to `gt_msort()` except that the compare function is of `GtCompareWithData` type accepting `comparinfo` as arbitrary data.

2.108 Module POSIX

`char* gt_basename(const char *path)`

This module implements the functiongt_basename() according to the specifications in http://www.unix-systems.org/onlinepubs/7908799/xsh/basename.html and http://www.opengroup.org/onlinepubs/009695399/
gt_basename() is equivalent to the function basename(3) which is available on most unix systems, but in different libraries and with slightly different functionality.
gt_basename() takes the pathname pointed to by path and returns a pointer to the final component of the pathname, deleting any trailing '/' characters.
If path consists entirely of the '/' character, thengt_basename() returns a pointer to the string "/".
If path is a null pointer or points to an empty string,gt_basename() returns a pointer to the string ".".
See the implementation ofgt_basename_unit_test() for additional examples.
The caller is responsible for freeing the received pointer!

2.109 Module Parseutils

`int gt_parse_int(int *out, const char *nptr)`

Parse integer from nptr and store result in out. Returns 0 upon success and -1 upon failure.

`int gt_parse_uint(unsigned int *out, const char *nptr)`

Parse unsigned integer from nptr and store result in out. Returns 0 upon success and -1 upon failure.

`int gt_parse_long(long *out, const char *nptr)`

Parse long from nptr and store result in out. Returns 0 upon success and -1 upon failure.

`int gt_parse_ulong(unsigned long *out, const char *nptr)`

Parse unsigned long from nptr and store result in out. Returns 0 upon success and -1 upon failure.

`int gt_parse_double(double *out, const char *nptr)`

Parse double from nptr and store result in out. Returns 0 upon success and -1 upon failure.

`int gt_parse_range(GtRange *rng, const char *start, const char *end, unsigned int line_number, const char *filename, GtError*)`

Parse a range given by start and end, writing the result into rng. Enforces that start is smaller or equal than end. Give filename and line_number for error reporting. Returns 0 upon success and -1 upon failure.

int gt_parse_range_tidy(GtRange *rng, const char *start, const char *end, unsigned int line_number, const char *filename, GtError*)

Like gt_parse_range, but issues a warning if start is larger then end and swaps both values. It also issues a warning, if start and/or end is not-positive and sets the corresponding value to 1.

void gt_fasta_show_entry(const char *description, const char *sequence, unsigned long sequence_length, unsigned long width, GtFile *outfp)

Print a fasta entry with optional description and mandatory sequence to outfp. If width is != 0 the sequence is formatted accordingly.

2.110 Module Qsort

void gt_qsort_r(void *a, size_t n, size_t es, void *data, GtCompareWithData cmp)

Like qsort(3), but allows an additional data pointer passed to the GtCompareWithData comparison function cmp.

2.111 Module Strcmp

int gt_strcmp(const char *s1, const char *s2)

Returns 0 if s1 == s2, otherwise the equivalent of strcmp(s1,s2). Useful as a performance improvement in some cases (for example, to compare symbols).

2.112 Module Symbol

const char* gt_symbol(const char *cstr)

Return a symbol (a canonical representation) for cstr. An advantage of symbols is that they can be compared for equality by a simple pointer comparison, rather than using strcmp() (as it is done in gt_strcmp()). Furthermore, a symbol is stored only once in memory for equal cstrs, but keep in mind that this memory can never be freed safely during the lifetime of the calling program. Therefore, it should only be used for a small set of cstrs.

2.113 Module Undef

#define GT_UNDEF_BOOL

The undefined bool value.

#define GT_UNDEF_CHAR

The undefined char value.

#define GT_UNDEF_DOUBLE

The undefined double value.

#define GT_UNDEF_FLOAT

The undefined float value.

#define GT_UNDEF_INT

The undefined int value.

#define GT_UNDEF_LONG

The undefined long value.

#define GT_UNDEF_UCHAR

The undefined <unsigned char> value.

#define GT_UNDEF_UINT

The undefined <unsigned int> value.

#define GT_UNDEF_ULONG

The undefined <unsigned long> value.

2.114 Module Unused

#define GT_UNUSED

Unused function arguments should be annotated with this macro to get rid of compiler warnings.

2.115 Module Version

const char* gt_version_check(unsigned int required_major, unsigned int required_minor, unsigned int required_micro)

Check that the *GenomeTools* library in use is compatible with the given version. Generally you would pass in the constants GT_MAJOR_VERSION, GT_MINOR_VERSION, and GT_MICRO_VERSION as the three arguments to this function.
Returns NULL if the *GenomeTools* library is compatible with the given version, or a string describing the version mismatch, if the library is not compatible.

2.116 Module Warning

void (*GtWarningHandler)(void *data, const char *format, va_list ap)

Handler type used to process warnings.

void gt_warning(const char *format, ...)

Print a warning according to format and ..., if a handler is set.

`void gt_warning_disable(void)`
: Disable that warnings are shown. That is, subsequent `gt_warning()` calls have no effect.

`void gt_warning_set_handler(GtWarningHandler warn_handler, void *data)`
: Set `warn_handler` to handle all warnings issued with `gt_warning()`. The `data` is passed to `warning_handler` on each invocation.

`void gt_warning_default_handler(void *data, const char *format, va_list ap)`
: The default warning handler which prints on `stderr`. ”warning: ” is prepended and a newline is appended to the message defined by `format` and `ap`. Does not use `data`.

`GtWarningHandler gt_warning_get_handler(void)`
: Return currently used `GtWarningHandler`.

`void* gt_warning_get_data(void)`
: Return currently used `data` which is passed to the currently used `GtWarningHandler`.

2.117 Module XANSI

`void gt_xatexit(void (*function)`
: Similar to `atexit(3)`, terminates on error.

`void gt_xfclose(FILE*)`
: Similar to `fclose(3)`, terminates on error.

`void gt_xfflush(FILE*)`
: Similar to `fflush(3)`, terminates on error.

`int gt_xfgetc(FILE*)`
: Similar to `fgetc(3)`, terminates on error.

`char* gt_xfgets(char *s, int size, FILE *stream)`
: Similar to `fgets(3)`, terminates on error.

`void gt_xfgetpos(FILE*, fpos_t*)`
: Similar to `fgetpos(3)`, terminates on error.

`FILE* gt_xfopen(const char *path, const char *mode)`
: Similar to `fopen(3)`, terminates on error.

`void gt_xfputc(int, FILE*)`
: Similar to `fputc(3)`, terminates on error.

`void gt_xfputs(const char*, FILE*)`
: Similar to `fputs(3)`, terminates on error.

`size_t gt_xfread(void *ptr, size_t size, size_t nmemb, FILE*)`
: Similar to `fread(3)`, terminates on error.

`#define gt_xfread_one(ptr, fp)`
: shortcut to gt_xfread

`void gt_xfseek(FILE*, long offset, int whence)`
: Similar to `fseek(3)`, terminates on error.

`void gt_xfsetpos(FILE*, const fpos_t*)`
: Similar to `fsetpos(3)`, terminates on error.

`void gt_xfwrite(const void *ptr, size_t size, size_t nmemb, FILE*)`
: Similar to `fwrite(3)`, terminates on error.

`#define gt_xfwrite_one(ptr, fp)`
: shortcut to gt_xfwrite

`void gt_xputchar(int)`
: Similar to `putchar(3)`, terminates on error.

`void gt_xputs(const char*)`
: Similar to `puts(3)`, terminates on error.

`void gt_xremove(const char*)`
: Similar to `remove(3)`, terminates on error.

`void gt_xungetc(int, FILE*)`
: Similar to `ungetc(3)`, terminates on error.

`void gt_xvfprintf(FILE *stream, const char *format, va_list ap)`
: Similar to `vfprintf(3)`, terminates on error.

`int gt_xvsnprintf(char *str, size_t size, const char *format, va_list ap)`
: Similar to `vsnprintf(3)`, terminates on error.