RGFA

RGFA is a Ruby library for working with GFA files. It allows to parse, validate, edit and write GFA files.

This manual explains how to access the information in GFA files using the library. It is completed by the more technical API library, which documents each class, method and constant defined by the library.

A test suite makes sure that the functionality described by this manual also works as intented. However, if this is not the case, please report any bug using the Github issues tracked (https://github.com/ggonnella/rgfa/issues).

GFA specifications

The library is based on the official GFA specifications version 1 and 2, available at https://github.com/GFA-spec/GFA-spec. See the Versions chapter for an overview of the differences of the two versions and methods for the conversion from one version to the other.

The RGFA object

The main class of the library is RGFA. An object of the class RGFA represents the content of a GFA file.

A RGFA instance can be created directly (using the `RGFA.new` method, or the method `RGFA.from_file(filename)` can be used to parse a GFA file and create a RGFA instance from it.

The `to_s` method converts the RGFA instance into its textual representation. Writing all information to a GFA file can be done directly using the `to_file(filename)` method.

Retrieving the lines

For many line times, iterating between all lines of the type can be done using a method which is named after the record type, in plural (`segments`, `paths`, `edges`, `links`, `containments`, `groups`, `fragments`, `comments`, `custom_lines`). The access to the header is done using a single line, which is retrieved using the `header` method.

Some lines use identifiers: segments, gaps, edges, paths and sets. Given an identifier, the line can be retrieved using the `line(id)` method. Note that identifier are represented in RGFA by Ruby symbols. The list of all identifier can be retrieved using the `names` method; for the identifiers of a single line type, use `segment_names`, `edges_names`, `gap_names`, `path_names` and `set_names`. The identifiers of external sequences in fragments are not part of the same namespace and can be retrieved using the `external_names` method.

Segments

Segment lines are available in both GFA1 and GFA2. They they represent the pieces of molecules, whose relations to other segments are coded by other line types.

In GFA1 a segment contains a segment name and a sequence (and, eventually, optional tags). In GFA2 the syntax is slightly different, as the segment contain an additional segment length field, which represent an eventually approximate length, which can be taken as a drawing indication for segments in graphical programs.

Relationships between segments

Segments are put in relation to each other by edges lines (E lines in GFA2, L and C Lines in GFA1), as well as gaps. RGFA allows to convert edges lines from one spefication version to the other (subject to limitations, see the Versions chapter). Gap lines cannot be converted, as no GFA1 specification exist for them.

Relationships to external sequences

Fragments represent relationships of segments to external sequences, i.e. sequences which are not represented in the GFA file itself. The typical application is to put contigs in relationship with the reads from which they are constructed.

The set of IDs of the external sequences may overlap the IDs of the GFA file itself (ie. the namespaces are separated). The list of external IDs referenced to by fragment lines can be retrieved using the `external_names` method of RGFA instances.

To find all fragments which refer to an external ID, the `fragments_for_external(ID)` method is used. As an external sequence can refer to different segments in different F lines, the result is always an array of F lines.

Conversely, to find all fragments for a particular segment, you may use the `fragments` method on the segment instance (see the References chapter).

Groups

Groups are lines which combine different other lines in an ordered (paths) or unordered (sets) way. RGFA supports both GFA1 paths and GFA2 paths and sets. Paths have a different syntax in the two specification versions. Methods are provided to edit the group components also without disconnecting the line instance (see the References chapter).

Other line types

The header contain metadata in a single or multiple lines. For ease of access to the header information, all its tags are summarized in a single line instance. See the Header chapter for more information. All lines which start by the symbol `#` are comments; they are handled in the Comments chapter. Custom lines are lines of GFA2 files which start with a non-standard record type. RGFA provides a limited support for accessing the information in custom lines.

Adding new lines

New lines can be added to a GFA file using the `add_line(line)` method or its alias `<<(line)`. The argument may be a string describing a line with valid GFA syntax, or an instance of the class `RGFA::Line` - if a string is added, a line instance is created and then added. A line instance can be created manually before adding it, using the `to_rgfa_line` string method.

Editing the lines

Accessing the information stored in the fields of a line instance is described in the `Positional fields` and `Tags` chapters.

Once a line instance has been added to a RGFA, either directly, or using its string representation, the line is said to be *connected* to the RGFA. Reading the information in fields is always allowed, while changing the content of some fields (fields which refer to other lines) is only possible for instances which are not connected.

In some cases, methods are provided to modify the content of reference fields of connected line (see the References chapter).

Removing lines

Removing a line can be done using the `rm(line)` method. The argument can be a line instance or a symbol (in which case the line is searched using the `line(name)` method, then eliminated). A line instance can also be disconnected using the `disconnect` method on it. Disconnecting a line may trigger other operations, such as the disconnection of other lines (see the References chapter).

Renaming lines

Lines with an identifier can be renamed. This is done simply by editing the corresponding field (such as segment_name). This field is not a reference to another line and can be freely edited also in line instances connected to a RGFA. All references to the line from other lines will still be up to date, as they will refer to the same instance (whose name has been changed) and their string representation will use the new name.

Validation

Different validation levels are available. They represent different compromises between speed and warrant of validity. The validation level can be specified when the RGFA object is created, using the `vlevel` parameter of `RGFA.new` and `RGFA.from_file`. Four levels of validation are defined (0 = no validation, 1 = validation by reading, 2 = validation by reading and writing, 3 = continuous validation). The default validation level value is 1.

Manual validation

Independently from the validation level choosen, the user can always check the value of a field calling `validate_field(fieldname)` on the line instance. If no exeption is raised, the field content is valid.

To check if the entire content of the line is valid, the user can call `validate` on the line instance. This will check all fields and perform cross-field validations, such as comparing the length of the sequence of a GFA1 segment, to the value of the LN tag (if present).

It is also possible to validate the structure of the GFA, for example to check if there are unresolved references to lines. To do this, use the `validate` method of the `RGFA` class.

No validations

If the validation is set to 0, RGFA will try to accept any input and never raise an exception. This is not always possible, and in some cases, an exception will still be raised, if the data is invalid.

Validation when reading

If the validation level is set to 1 or higher, basic validations will be performed, such as checking the number of positional fields, the presence of duplicated tags, the tag datatype of predefined tags. Additionally, all tags will be validated, either during parsing or on first access. Record-type cross-field validations will also be performed.

In other words, a validation of 1 means that RGFA guarantees (as good as it can) that the GFA content read from a file is valid, and will raise an exception on accessing the data if not.

The user is supposed to run `validate_field(fieldname)` when changing a field content to something which can be potentially invalid, or `validate` if potentially cross-field validations could fail.

Validation when writing

Setting the level to 2 will perform all validations described above, plus validate the fields content when their value is written to string.

In other words, a validation of 2 means that RGFA guarantee (as good as it can) that the GFA content read from a file and written to a file is valid and will raise an exception on accessing the data or writing to file if not.

Continuous validation

If the validation level is set to 3, all validations for lower levels described above are run, plus a validation of fields contents each time a setter method is used.

A validation of 3 means that RGFA guarantees (as good as it can) that the GFA content is always valid.

Summary of validation related methods

```
RGFA#validate
RGFA::Line#validate
RGFA::Line#validate_field(fieldname)
```

Positional fields

Most lines in GFA have positional fields (Headers are an exception). During parsing, if a line is encountered, which has too less or too many positional fields, an exception will be thrown. The correct number of positional fields is record type-specific.

Positional fields are recognized by its position in the line. Each positional field has an implicit field name and datatype associated with it.

Field names

The field names are derived from the specification. Lower case versions of the field names are used and spaces are subsituted with underscores.

The following tables shows the field names used in RGFA, for each kind of line. Headers have no positional fields. Comments and custom lines follow particular rules, see the respective chapters.

GFA1 field names

Record Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
Segment	`name`	`sequence`				
Link	`from`	`from_orient`	`to`	`to_orient`	`overlap`	
Containment	`from`	`from_orient`	`to`	`to_orient`	`pos`	`overlap`
Path	`path_name`	`segment_names`	`overlaps`			

GFA2 field names

Record Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6	Field 7	Field 8
Segment	`sid`	`slen`	`sequence`					
Edge	`eid`	`sid1`	`sid2`	`beg1`	`end1`	`beg2`	`end2`	`alignment`
Fragment	`sid`	`external`	`s_beg`	`s_end`	`f_beg`	`f_end`	`alignment`	
Gap	`gid`	`sid1`	`d1`	`d2`	`sid2`	`disp`	`var`	
U Group	`pid`	`items`						
O Group	`pid`	`items`						

Datatypes

The datatype of each positional field is described in the specification and cannot be changed (differently from tags). Here is a short description of the Ruby classes used to represent data for different datatypes. For some complex cases, more details are found in the following chapters.

Placeholders

The positional fields in GFA can never be empty. However, there are some fields with optional values. If a value is not specified, a placeholder character is used instead (`*`). Such undefined values are represented in RGFA by the Placeholder class, which is described more in detail in the Placeholders chapter.

Arrays

The `items` field in unordered and ordered groups and the `segment_names` and `overlaps` fields in paths are lists of objects and are represented by Array instances.

Orientations

Orientations are represented by symbols. Applying the `invert` method on an orientation symbol returns the other orientation, e.g.

```
:+.invert # => :-
```

Identifiers

The identifier of the line itself (available for S, P, E, G, U, O lines) can always be accessed in RGFA using the `name` alias and is represented in RGFA by a Symbol. If it is optional (E, G, U, O lines) and not specified, it is represented by a Placeholder instance. The fragment identifier is also a Symbol.

Identifiers which refer to other lines are also present in some line types (L, C, E, G, U, O, F). These are never placeholders and in stand-alone lines are represented by symbols. In connected lines they are references to the Line instances to which they refer to (see the References chapter).

Oriented identifiers

Oriented identifiers (e.g. `segment_names` in GFA1 paths) are represented by elements of the class `RGFA::OrientedLine`. The `segment` method of the oriented segments returns the segment identifier (or segment reference in connected path lines) and the `orient` method returns the orientation symbol. The `name` method returns the symbol of the segment, even if this is a reference to a segment. A new oriented line can be created using the `OL[line, orientation]` method.

Calling `invert` returns an oriented segment, with inverted orientation. To set the two attributes the methods `segment=` and `orient=` are available.

Examples:

```
p = "P\tP1\ta+,b-\t*".to_rgfa_line
p.segment_names # => [OrientedLine(:a,:+),OrientedLine(:b,:-)]
p[0].segment # => :a
p[0].name # => :a
p[0].orient # => :+
p[0].invert # => OrientedLine(:a,:-)
p[0].orient = :-
p[0].segment = "S\tX\t*".to_rgfa_line
p[0] # => OrientedLine(RGFA::Line("S\tX\t*"), :-)
```

```
p[0].name # => :X
p[0] = OL[RGFA::Line("S\tY\t*"), :+]
```

Sequences

Sequences (S field sequence) are represented by strings in RGFA. Depending on the GFA version, the alphabet definition is more or less restrictive. The definitions are correctly applied by the validation methods.

The method `rc` is provided to compute the reverse complement of a nucleotidic sequence. The extended IUPAC alphabet is understood by the method. Applied to non nucleotidic sequences, the results will be meaningless:

```
"gcat".rc # => "atgc"
"*".rc # => "*" (placeholder)
"yatc".rc # => "gatr" (wildcards)
"gCat".rc # => "atGc" (case remains)
"ctg".rc(rna: true) # => "cug"
```

Integers and positions

The C lines `pos` field and the G lines `disp` and `var` fields are represented by integers. The `var` field is optional, and thus can be also a placeholder. Positions are 0-based coordinates.

The position fields of GFA2 E lines (`beg1, beg2, end1, end2`) and F lines (`s_beg, s_end, f_beg, f_end`) contain a dollar symbol as suffix if the position is equal to the segment length. For more information, see the Positions chapter.

Alignments

Alignments are always optional, ie they can be placeholders. If they are specified they are CIGAR alignments or, only in GFA2, trace alignments. For more details, see the Alignments chapter.

GFA1 datatypes

Datatype	Record Type	Fields
Identifier	Segment	`name`
	Path	`path_name`
	Link	`from, to`
	Containment	`from, to`
[OrientedIdentifier]	Path	`segment_names`
Orientation	Link	`from_orient, to_orient`
	Containment	`from_orient, to_orient`

Datatype	Record Type	Fields
Sequence	Segment	`sequence`
Alignment	Link	`overlap`
	Containment	`overlap`
[Alignment]	Path	`overlaps`
Position	Containment	`pos`

GFA2 datatypes

Datatype	Record Type	Fields
Itentifier	Segment	`sid`
	Fragment	`sid`
OrientedIdentifier	Edge	`sid1, sid2`
	Gap	`sid1, sid2`
	Fragment	`external`
OptionalIdentifier	Edge	`eid`
	Gap	`gid`
	U Group	`oid`
	O Group	`uid`
[Identifier]	U Group	`items`
[OrientedIdentifier]	O Group	`items`
Sequence	Segment	`sequence`
Alignment	Edge	`alignment`
	Fragment	`alignment`
Position	Edge	`beg1, end1, beg2, end2`
	Fragment	`s_beg, s_end, f_beg, f_end`
Integer	Gap	`disp, var`

Reading and writing positional fields

The `RGFA::Line#positional_fieldnames` method returns the list of the names (as symbols) of the positional fields of a line.

The positional fields can be read using a method on the RGFA line object, which is called as the field name. Setting the value is done with an equal sign version of the field name method (e.g. segment.slen = 120). In alternative, the `set(fieldname, value)` and `get(fieldname)` methods can also be used.

When a field is read, the value is converted into an appropriate object. The string representation of a field can be read using the `field_to_s(fieldname)` method.

When setting a value, the user can specify the value of a tag either as a Ruby object, or as the string representation of the value.

Note that setting the value of reference and backreferences-related fields is generally not allowed, when a line instance is connected to a RGFA object (see the References chapter).

Validation

The content of all positional fields must be a correctly formatted string according to the rules given in the GFA specifications (or a Ruby object whose string representation is a correctly formatted string).

Depending on the validation level, more or less checks are done automatically (see the Validation chapter). Not regarding which validation level is selected, the user can trigger a manual validation using the `validate_field(fieldname)` method for a single field, or using `validate`, which does a full validation on the whole line, including all positional fields.

Aliases

For some fields, aliases are defined, which can be used in all contexts where the original field name is used (i.e. as parameter of a method, and the same setter and getter methods defined for the original field name are also defined for each alias, see below).

Name

Different record types have an identifier field: segments (name in GFA1, sid in GFA2), paths (path_name), edge (eid), fragment (sid), gap (gid), groups (pid).

All these fields are aliased to `name`. This allows the user for example to set the identifier of a line using the `name=(value)` method using the same syntax for different record types (segments, edges, paths, fragments, gaps and groups).

Version-specific field names

For segments the GFA1 name and the GFA2 sid are equivalent fields. For this reason an alias `sid` is defined for GFA1 segments and `name` for GFA2 segments.

Crypical field names

The definition of from and to for containments is somewhat cryptical. Therefore following aliases have been defined for containments: container[_orient] for from[_orient]; contained[_orient] for to[_orient]

Summary of positional fields-related API methods

```
RGFA::Line#<fieldname>/<fieldname>=
RGFA::Line#get/set
RGFA::Line#validate_field/validate
Symbol#invert
String#rc
```

Placeholders

Some positional fields may contain an undefined value S: sequence; L/C: overlap; P: overlaps; E: eid, alignment; F: alignment; G: gid, var; U/O: pid. In GFA this value is represented by a `*`.

In RGFA instances of the class RGFA::Placeholder (and its subclasses) represent the undefined value.

Distinguishing placeholders

The method #placeholder? is defined for placeholders and all classes whose instances can be used as a value for fields where a placeholder is allowed. It allows to check if a value is a placeholder instance or an equivalent value (such as an empty array, or the string representation of the placeholder).

Compatibility methods

Some methods are defined for placeholders, which allow them to respond to the same methods as defined values. For example, for all placeholders, #empty? returns true; #validate does nothing; #length returns 0; #[] returns self; #+ returns self. Thus in many cases the code can be written in a generic way, without explicitely handling the different cases where a value is a placeholder or not.

Summary of API methods related to placeholders

```
RGFA::Placeholder#to_s
RGFA::Placeholder#placeholder?
String/Symbol/Array/Integer#placeholder?
RGFA::Placeholder#empty?
RGFA::Placeholder#validate
RGFA::Placeholder#length
RGFA::Placeholder#[]
RGFA::Placeholder#+
```

Position fields

The only position field in GFA1 is the `pos` field in the C lines. This represents the starting position of the contained segment in the container segment and is 0-based.

Some fields in GFA2 E lines (`beg1, beg2, end1, end2`) and F lines (`s_beg, s_end, f_beg, f_end`) are positions. According to the specification, they are 0-based and represent virtual ticks before and after each symbol in the sequence. Thus ranges are represented similarly to the Python range conventions: e.g. a 1-character prefix of a sequence will have begin 0 and end 1.

GFA2 last position symbol

The GFA2 positions must contain an additional symbol (`$`) appended to the integer, if (and only if) they are the last position in the segment sequence. These particular positions are represented in RGFA as instances of the class RGFA::LastPos.

To create a lastpos instance, `to_lastpos` can be called on an integer, or `to_pos` can be called on the string representation:

```
12.to_lastpos # => RGFA::LastPos with value 12
"12".to_pos   # => 12
"12$".to_pos  # => RGFA::LastPos with value 12
```

Subtracting an integer from a lastpos returns a lastpos if 0 subtracted, an integer otherwise. This allows to do some arithmetic on positions without making them invalid.

```
12.to_lastpos - 0 # => RGFA::LastPos(value: 12)
12.to_lastpos - 1 # 11
```

The methods first? and last? allow to determine if a position value is 0 (first?), or if it is a last position (last?), using the same syntax fo lastpos and integer instances.

```
0.first?  # true
0.last?   # false
12.first? # false
12.last?  # false
"12".to_pos.first? # false
"12$".to_pos.last? # true
```

Summary of position-related API methods

```
String#to_pos
Integer#to_lastpos
```

```
Integer/RGFA::LastPos#first?
Integer/RGFA::LastPos#last?
RGFA::LastPos.-
```

Alignments

Some fields contain alignments and lists of alignments (L/C: overlap; P: overlaps; E/F: alignment). If an alignment is not given, the placeholder symbol `*` is used instead. In GFA1 the alignments can be given as CIGAR strings, in GFA2 also as Dazzler traces.

RGFA uses different classes (in module RGFA::Alignment) for representing the two possible alignment styles (cigar strings and traces) and undefined alignments (placeholders).

Creating an alignment

An alignment instance is usually created from its GFA string representation by using the `String#to_alignment` method:

```
"*".to_alignment        # => RGFA::Alignment::Placeholder
"10,10,10".to_alignment # => RGFA::Alignment::Trace
"30M2I30M".to_alignment # => RGFA::Alignment::CIGAR
```

The alignment classes also provide a `to_alignment` method (returning self), so that is always safe to call the method on a variable which can contain a string or an alignment instance:

```
RGFA::Alignment::Placeholder.new.to_alignment
RGFA::Alignment::Trace.new([12,13,0]).to_alignment
```

Recognizing undefined alignments

The `placeholder?` method is available for strings and alignment instances and is the correct way to understand if an alignment field contains a defined value (cigar, trace) or not (placeholder).

```
"30M".to_alignment.placeholder? # => false
"10,10,10".to_alignment.placeholder? # => false
"*".to_alignment.placeholder? # => true
"*".placeholder? # => true
RGFA::Alignment::CIGAR.new([]).placeholder? # => true
RGFA::Alignment::Trace.new([]).placeholder? # => true
RGFA::Alignment::Placeholder.new.placeholder? # => true
```

Reading and editing CIGARs

CIGARs are represented by arrays of cigar operation objects. Each cigar operation provides the methods `len`/`len=` and `code`/`code=`. Len is the length of the operation (Integer).

```
cigar = "30M".to_alignment
cigar.kind_of?(Array) # => true
operation = cigar[0]
operation.class # => RGFA::Alignment::CIGAR::Operation
operation.code # => :M
operation.len # => 30
operation.to_s # => "30M"
operation.code = :D
operation.len = 10
operation.to_s # => "10D"
```

CIGAR values can be edited using the methods `len=` and `code=` of the single operations or editing the array itself (which allows e.g. to add or remove operations). If the array is emptied, its string representation will be `*`.

```
cigar = "30M".to_alignment
cigar << RGFA::Alignment::CIGAR::Operation.new(12, :D)
cigar.to_s # "30M12D"
cigar.delete(cigar[1])
cigar.to_s # "30M"
cigar.delete(cigar[0])
cigar.to_s # "*"
```

CIGARs consider one sequence as reference and another sequence as query. The `length_on_reference` and `length_on_query` methods compute the length of the alignment on the two sequences. These methods are used by the library e.g. to convert GFA1 L lines to GFA2 E lines (which is only possible if CIGARs are provided).

```
cigar = "30M10D20M5I10M".to_alignment
cigar.length_on_reference # => 70
cigar.length_on_query # => 65
```

Validation

The `validate` method checks if all operations in a cigar use valid codes and length values (which must be non-negative) The codes can be M, I, D or P. For GFA1 the other codes are formally accepted (no exception is raised), but their use is discouraged. An error is raised in GFA2 on validation, if the other codes are used.

```
cigar = "30M10D20M5I10M".to_alignment
```

```
cigar.validate # no exception raised
cigar = "-30M".to_alignment
cigar.validate # raises an exception
cigar = "30X".to_alignment
cigar.validate # raises an exception
cigar = "10=".to_alignment(version: :gfa1)
cigar.validate # no exception raised
cigar = "10=".to_alignment(version: :gfa2)
cigar.validate # raises an exception
```

Reading and editing traces

Traces are arrays of non-negative integers. The values are interpreted using a trace spacing value. If traces are used, a trace spacing value must be defined in a TS integer tag, either in the header, or in the single lines which contain traces.

```
gfa.header.TS     # => the global TS value
gfa.edges(:x).TS # => an edge''s own TS tag
```

Complement alignment

CIGARs are dependent on which sequence is taken as reference and which is taken as query. For each alignment, a complement CIGAR can be computed (using the method `complement`), which is the CIGAR obtained when the two sequences are switched. This method is used by the library e.g. to compare links, as they can be expressed in different ways, by switching the two sequences.

```
cigar = "2M1D3M".to_alignment
cigar.complement.to_s # => "3M1I2M"
```

The current version of RGFA does not provide a way to compute the alignment in RGFA, thus the trace information can be accessed and edited, but not used for this purpose. Because of this there is currently no way in RGFA to compute a complement trace (trace obtained when the sequences are switched).

```
trace = "1,2,3".to_alignment
trace.complement.to_s # => "*"
```

The complement of a placeholder is a placeholder:

```
"*".to_alignment.complement.to_s # => "*"
```

Tags

Each record in GFA can contain tags. Tags are fields which consist in a tag name, a datatype and data. The format is `NN:T:DATA` where `NN` is a two-letter tag name, `T` is an one-letter datatype symbol and `DATA` is a string representing the data according to the specified datatype. Tag names must be unique for each line, i.e. each line may only contain a tag once.

```
# Examples of GFA tags of different datatypes:
aa:i:-12
bb:f:1.23
cc:Z:this is a string
dd:A:X
ee:B:c,12,3,2
ff:H:122FA0
gg:J:["A","B"]
```

Custom tags

Some tags are explicitely defined in the specification (these are named *predefined tags* in RGFA), and the user or an application can define its own custom tags.

Custom tags are user or program specific and may of course collide with the tags used by other users or programs. For this reasons, if you write scripts which employ custom tags, you should always check that the values are of the correct datatype and plausible.

```ruby
if line.get_datatype(:xx) != :i
  raise "I expected the tag xx to contain an integer!"
end
myvalue = line.xx
if (myvalue > 120) or (myvalue % 2 == 1)
  raise "The value in the xx tag is not an even value <= 120"
end
# ... do something with myvalue
```

Also it is good practice to allow the user of the script to change the name of the custom tags. For example, RGFATools employs the +or+ custom tag to track the original segment from which a segment in the final graph is derived. All methods which read or write the +or+ tag allow to specify an alternative tag name to use instead of +or+, for the case that this name collides with the custom tag of another program.

```ruby
# E.g. a method which does something with myvalue, usually stored in tag xx
# allows the user to specify an alternative name for the tag
# @param mytag [Symbol] <i>(defaults to: +:xx+)</i> tag where value is stored
def mymethod(line, mytag: :xx)
  myvalue = line.xx
  # .... do something with myvalue
end
```

Tag names in GFA1

According to the GFA1 specification, custom tags are lower case, while predefined tags are upper case (in both cases the second character in the name can be a number). There is a number of predefined tags in the specification, different for each kind of line.

```
VN:Z:1.0 # VN is upcase => predefined tag
z5:Z:1.0 # z5 first char is downcase => custom tag

# not forbidden, but not reccomended:
zZ:Z:1.0 # => mixed case, first char downcase => custom tag
Zz:Z:1.0 # => mixed case, first char upcase => custom tag
vn:Z:1.0 # => same name as predefined tag, but downcase => custom tag
```

Besides the tags described in the specification, in GFA1 headers, the TS tag is allowed, in order to simplify the translation of GFA2 files.

Tag names in GFA2

The GFA2 specification is currently not as strict regarding tags: anyone can use both upper and lower case tags, and no tags are predefined except for VN and TS.

However, RGFA follows the same conventions as for GFA1: i.e. it allows the tags specified as predefined tags in GFA1 to be used also in GFA2. No other upper case tag is allowed in GFA2.

Datatypes

The following table summarizes the datatypes available for tags:

Symbol	Datatype	Example	Ruby class
Z	string	This is a string	String
i	integer	-12	Fixnum
f	float	1.2E-5	Float
A	char	X	String
J	JSON	[1,{“k1”:1,“k2”:2},“a”]	Array/Hash
B	numeric array	f,1.2,13E-2,0	RGFA::NumericArray
H	byte array	FFAA01	RGFA::ByteArray

Validation

The tag name is validated according the the rules described above: except for the upper case tags indicated in the GFA1 specification, and the TS header tag, all other tags must contain at least one lower case letter.

```
VN # => in header: allowed, elsewhere: error
TS # => allowed in headers and GFA2 Edges
KC # => allowed in links, containments, GFA1/GFA2 segments
xx # => custom tag, always allowed
xxx # => error: name is too long
x # => error: name is too short
11 # => error: at least one letter must be present
```

The datatype must be one of the datatypes specified above. For predefined tags, RGFA also checks that the datatype given in the specification is used.

```
xx:X # => error: datatype X is unknown
VN:i # => error: VN must be of type Z
```

The data must be a correctly formatted string for the specified datatype or a Ruby object whose string representation is a correctly formatted string.

```
# current value: xx:i:2
line.xx = 1   # OK
line.xx = "1" # OK, value is set to 1
line.xx = "A" # error
```

Depending on the validation level, more or less checks are done automatically (see validation chapter). Per default - validation level (1) - validation is performed only during parsing or accessing values the first time, therefore the user must perform a manual validation if he changes values to something which is not guaranteed to be correct. To trigger a manual validation, the user can call the method `validate_field(fieldname)` to validate a single tag, or `validate` to validate the whole line, including all tags.

```
line.xx = "A"
line.validate_field(:xx) # validates xx
# or, to validate the whole line, including tags:
line.validate
```

Reading and writing tags

Tags can be read using a method on the RGFA line object, which is called as the tag (e.g. line.xx). A banged version of the method raises an error if the tag was not available (e.g. line.LN!), which the normal method returns `nil` in this case. Setting the value is done with an equal sign version of the tag name method (e.g. line.TS = 120). In alternative, the `set(fieldname, value)`, `get(fieldname)` and `get!(fieldname)` methods can also be used. To remove a tag from a line, use the `delete(fieldname)` method, or set its value to `nil`.

```
# line is "H xx:i:12"
line.xx   # => 1
line.xy   # => nil
line.xx! # => 1
line.xy! # => error: xy is not defined
line.get(:xx)  # => 1
line.get!(:xy) # => error, xy is not defined
line.xx = 2    # => value of xx is changed to 2
line.xx = "a"  # => error: not compatible with existing type (i)
line.xy = 2    # => xy is created and set to 2, type is auto-set to i
line.set(:xy, 2) # => sets xy to 2
line.delete(:xy) # => tag is eliminated
line.xx = nil    # => tag is eliminated
```

The `RGFA::Line#tagnames` method, returns the list of the names (as symbols) of all defined tags for a line. Alternatively, to test if a line contains a tag, it is possible to use the not-banged get method (e.g. line.VN), as this returns nil if the tag is not defined, and a non-nil value if the tag is defined.

```
puts "Line contains the following tags:"
line.tagnames.each do |tagname|
  puts tagname
end
if line.VN
  # do something with line.VN value
end
```

When a tag is read, the value is converted into an appropriate object (see Ruby classes in the datatype table above). When setting a value, the user can specify the value of a tag either as a Ruby object, or as the string representation of the value.

```
# line is: H xx:i:1 xy:Z:TEXT xz:J:["a","b"]
line.xx # => 1 (Integer)
line.xy # => "TEXT" (String)
line.xz # => ["a", "b"] (Array)
```

The string representation of a tag can be read using the `field_to_s(fieldname)` method. The default is to only output the content of the field. By setting "tag: true"', the entire tag is output (name, datatype, content, separated by colons). An exception is raised if the field does not exist.

```
# line is: H xx:i:1
line.xx # => 1 (Integer)
line.field_to_s(:xx) # => "1" (String)
line.field_to_s(:xx, tag: true) # => "xx:i:1"
```

Datatype of custom tags

The datatype of an existing custom field (but not of predefined fields) can be changed using the `set_datatype(fieldname, datatype)` method. The current datatype specification can be read using `get_datatype(fieldname)`. Thereby the fieldname and datatype arguments are Ruby symbols.

```
# line is: H xx:i:1
line.get_datatype(:xx) # => :i
line.set_datatype(:xx, :Z)
```

If a new custom tag is specified, RGFA selects the correct datatype for it: i/f for numeric values, J/B for arrays, J for hashes and Z for strings and symbols. If the user wants to specify a different datatype, he may do so by setting it with `set_datatype` (this can be done also before assigning a value, which is necessary if full validation is active).

```
# line has not tags
line.xx = "1" # => "xx:Z:1" created
line.xx # => "1"
line.set_datatype(:xy, :i)
line.xy = "1" # => "xy:i:1" created
line.xy # => 1
```

Arrays of numerical values

`B` and `H` tags represent array with particular constraints (e.g. they can only contain numeric values, and in some cases the values must be in predefined ranges). In order to represent them correctly and allow for validation, Ruby classes have been defined for both kind of tags: `RGFA::ByteArray` for `H` and `RGFA::NumericArray` for `B` fields.

Both are subclasses of Array. Object of the two classes can be created by converting the string representation (using `to_byte_array` and `to_numeric_array`). The same two methods can be applied also to existing Array instances containing numerical values.

```
# create a byte array instance
[12,3,14].to_byte_array
"A012FF".to_byte_array
# create a numeric array instance
"c,12,3,14".to_numeric_array
[12,3,14].to_numeric_array
```

Instances of the classes behave as normal arrays, except that they provide a #validate method, which checks the constraints, and that their #to_s method computes the GFA string representation of the field value.

```
[12,1,"1x"].to_byte_array.validate # => error: 1x is not a valid value
[12,3,14].to_numeric_array.to_s # => "c,12,3,14"
```

For numeric values, the `compute_subtype` method allows to compute the subtype which will be used for the string representation. Unsigned subtypes are used if all values are positive. The smallest possible subtype range is selected. The subtype may change when the range of the elements changes.

```
[12,13,14].to_numeric_value.compute_subtype # => "C"
```

Special cases: custom records, headers, comments and virtual lines.

GFA2 allows custom records, introduced by record type symbols other than the predefined ones. RGFA uses a pragmatical approach for identifying tags in custom records, and tries to interpret the rightmost fields as tags, until the first field from the right raises an error; all remaining fields are treated as positional fields.

```
X a b c xx:i:12 # => xx is tag, a, b, c are positional fields
Y a b xx:i:12 c # => all positional fields, as c is not a valid tag
```

For easier access, the entire header of the GFA is summarized in a single line instance. Different GFA header lines can contain the same tag (this was a discussed topic, it is not forbidden by the current specifications, but this may change). A class (`RGFA::FieldArray`) has been defined to handle this special case (see Header chapter for details).

Comment lines are represented by a subclass of the same class (`RGFA::Line`) as the records. However, they cannot contain tags: the entire line is taken as content of the comment.

```
# this is not a tag: xx:i:1 # => not a tag, xx:i:1 is part of the string content
```

Virtual `RGFA::Line` instances (e.g. Segment instances automatically created because of not yet resolved references found in edges) cannot be modified by the user, and tags cannot be specified for them. This includes all instances of the `RGFA::Line::Unknown` class.

References

Some fields in GFA lines contain identifiers or lists of identifiers (sometimes followed by orientation symbols), which reference other lines of the GFA file.

Connecting a line to a RGFA object

In stand-alone line instances, the identifiers which reference other lines are symbols (or, if they are oriented identifiers, then instances of RGFA::OrientedLine containing a symbol). Lists of identifiers are represented by arrays of symbols and oriented segment instances.

When a line is connected to a RGFA object (adding the line using `RGFA#<<(line)` or calling `RGFA::Line#connect(rgfa)`), the symbols in the fields (and in arrays and oriented line instances) are changed into references to the corresponding lines in the RGFA object.

The method `RGFA::Line#connected?` allows to determine if a line is connected to an RGFA instance. The method `RGFA::Line#rgfa` returns the RGFA instance to which the line is connected.

References for each record type

The following tables list the references for each record type. `[]` represent arrays.

GFA1

Record type	Fields	Type of reference
Link	from, to	Segment
Containment	from, to	Segment
Path	segment_names,	[OrientedLine(Segment)]
	links (1)	[OrientedLine(Link)]

(1): paths contain information in the fields segment_names and overlaps, which allow to find the identify from which they depend; these links can be retrieved using `links` (which is not a field).

GFA2

Record type	Fields	Type of reference
Edge	sid1, sid2	Segment
Gap	sid1, sid2	Segment
Fragment	sid	Segment
U Group	items	[Edge/O-Group/U-Group/Segment]
O Group	items	[OrientedLine(Edge/O-Group/Segment)]

Backreferences for each record type

When a line containing a reference to another line is connected to a RGFA object, backreferences to it are created in the targeted line.

For each backreference collection a getter method exist, which is named as the collection (e.g. `RGFA::Line::Segment#dovetails_L`). The methods return frozen arrays (as changing the content of the array directly would invalid other related references in the graph object). To change the reference which generated the backreference, see the section “Editing reference fields” below.

The following tables list the backreferences collections for each record type.

GFA1

Record type	Backreferences
Segment	dovetails_L
	dovetails_R
	edges_to_contained
	edges_to_containers
	paths
Link	paths

GFA2

Record type	Backreferences	Type
Segment	dovetails_L	E
	dovetails_R	E
	edges_to_contained	E
	edges_to_containers	E
	internals	E
	gaps_L	G
	gaps_R	G
	fragments	F
	paths	O
	sets	U
Edge	paths	O
	sets	U
O Group	paths	O
	sets	U
U Group	sets	U

Backreference convenience methods

In some cases, additional methods are available which combine in different way the backreferences information.

The segment `dovetails` and `gaps` methods take an optional argument. Without argument all dovetail overlaps (references to links or dovetail edges) or gaps are returned. If :L or :R is provided as argument, the dovetails overlaps (or gaps) of the left or, respectively, right end of the segment sequence are returned (equivalent to dovetails_L/dovetails_R and gaps_L/gaps_R). The segment `containments` methods returns both containments where the segment is the container or the contained segment. The segment `edges` method returns all edges (dovetails, containments and internals) with a reference to the segment.

Other methods directly compute list of segments from the edges lists mentioned above. In particular, the segment `neighbours` method computes the set of segment instances which are connected by dovetails to the segment. The segment `containers` and `contained` methods similarly compute the set of segment instances which, respectively, contains the segment, or are contained in the segment.

Multiline group definitions

Groups can be defined on multiple lines, by using the same ID for each line defining the group. If multiple RGFA::Line::Group instances with the same ID are connected to the RGFA, the final RGFA will only contain the last instance: all previous one are disconnected and their items list prepended to the last instance. All tags will be copied to the last instance added.

The tags of multiple line defining a group may not contradict each other. Either are the tag names on different lines defining the group all different, or, if the same tag is present on different lines, the value and datatype must be the same.

Induced set and captured path

The item list in GFA2 sets and paths may not contain elements which are implicitly involved. For example a path may contain segments, without specifying the edges connecting them, if there is only one such edge. Alternatively a path may contain edges, without explitely indicating the segments. Similarly a set may contain edges, but not the segments refered to in them, or contain segments which are connected by edges, without the edges themselves. Furthermore groups may refer to other groups (set to sets or paths, paths to paths only), which then indirectly contain references to segments and edges.

RGFA provides methods for the computation of the sets of segments and edges which are implied by an ordered or unordered group. Thereby all references to subgroups are resolved and implicit elements are added, as described in the specification. The computation can, therefore, only be applied to connected lines. For unordered groups, this computation is provided by the method `induced_set`, which returns an array of segment and edge instances. For ordered group, the computation is provided by the method `captured_path`, whcih returns a list of RGFA::OrientedLine instances, alternating segment and edge instances (and starting and ending in segments).

The methods `induced_segments_set`, `induced_edges_set`, `captured_segments` and `captured_edges` return, respectively, the list of only segments or edges, in ordered or unordered groups.

Disconnecting a line from a RGFA object

Lines can be disconnected using `RGFA#rm(line)` or `RGFA::Line#disconnect`.

Disconnecting a line affects other lines as well. Lines which are dependent on the disconnected line are disconnected as well. Any other reference to disconnected lines is removed as well. In the disconnected line, references to lines are transformed back to symbols and backreferences are deleted.

The following tables show which dependent lines are disconnected if they refer to a line which is being disconnected.

GFA1

Record type	Dependent lines
Segment	links (+ paths), containments
Link	paths

GFA2

Record type	Dependent lines
Segment	edges, gaps, fragments, u-groups, o-groups
Edge	u-groups, o-groups
U-Group	groups

Editing reference fields

In connected line instances, it is not allowed to directly change the content of fields containing references to other lines, as this would make the state of the RGFA object invalid.

Besides the fields containing references, some other fields are read-only in connected lines. Changing some of the fields would require moving the backreferences to other collections (position fields of edges and gaps, from_orient and to_orient of links). The overlaps field of connected links is readonly as it may be necessary to identify the link in paths.

Renaming an element

The name field of a line (e.g. segment name/sid) is not a reference and thus can be edited also in connected lines. When the name of the line is changed, no manual editing of references (e.g. from/to fields in links) is necessary, as all lines which refer to the line will still refer to the same instance. The references to the instance in the RGFA lines collections will be automatically updated. Also, the new name will be correctly used when converting to string, such as when the RGFA is written to a GFA file.

Renaming a line to a name which already exists has the same effect of adding a line with that name. That is, in most cases, `RGFA::NotUniqueError` is raised. An exception are GFA2 groups: in this case the line will be appended to the existing line with the same name.

Adding and removing group elements

Elements of GFA2 groups can be added and removed from both connected and non-connected lines, using the following methods.

To add an item to or remove an item from an unordered group, use the methods `add_item(item)` and `rm_item(item)`, which take as argument either a symbol (identifier) or a line instance.

To append or prepend an item to an ordered group, use the methods `append_item(item)` and `prepend_item(item)`. To remove the first or the last item of an ordered group use the methods `rm_first_item` and `rm_last_item`.

Editing read-only fields of connected lines

Editing the read-only information of edges, gaps, links, containments, fragments and paths is more complicated. These lines shall be disconnected before the edit and connected again to the RGFA object after it. Before disconnecting a line, you should check if there are other lines dependent on it (see tables above). If so, you will have to disconnect these lines first, eventually update their fields and reconnect them at the end of the operation.

Virtual lines

The order of the lines in GFA is not prescribed. Therefore, during parsing, or constructing a RGFA in memory, it is possible that a line is referenced to, before it is added to the RGFA instance. Whenever this happens, RGFA creates a "virtual" line instance.

Users do not have to handle with virtual lines, if they work with complete and valid GFA files.

Virtual lines are similar to normal line instances, with some limitations (they contain only limited information and it is not allowed to add tags to them). To check if a line is a virtual line, one can use the `RGFA::Line#virtual?` method.

As soon as the parser founds the real line corresponding to a previously introduced virtual line, the virtual line is exchanged with the real line and all references are corrected to point to the real line.

Summary of references-related API methods

```
RGFA#<<(line)/rm(line)
RGFA::Line#connect(rgfa)
RGFA::Line#disconnect
RGFA::Line#connected?
RGFA::Line#rgfa
RGFA::Line#virtual?
RGFA::Line::Segment::GFA1/GFA2#dovetails(_L|_R)
RGFA::Line::Segment::GFA1/GFA2#dovetails
RGFA::Line::Segment::GFA1/GFA2#neighbours
RGFA::Line::Segment::GFA1/GFA2#contain(ed|ers)
RGFA::Line::Segment::GFA1/GFA2#edges_to_contain(ed|ers)
RGFA::Line::Segment::GFA1/GFA2#containments
RGFA::Line::Segment::GFA1/GFA2#internals
RGFA::Line::Segment::GFA1/GFA2#edges
RGFA::Line::Segment::GFA2#gaps(_L|_R)
RGFA::Line::Segment::GFA2#gaps
RGFA::Line::Segment::GFA2#fragments
RGFA::Line::Segment::GFA1/GFA2#paths
RGFA::Line::Segment::GFA2#sets
RGFA::Line::Fragment#sid
RGFA::Line::Edge::Containment/Link#from/to
RGFA::Line::Gap/Edge::GFA2#sid1/sid2
RGFA::Line::Gap/Edge::GFA2#sets/paths
RGFA::Line::Group::Path#segment_names
RGFA::Line::Group::Path#links
RGFA::Line::Group::Unordered#items
RGFA::Line::Group::Unordered#paths
RGFA::Line::Group::Unordered#add_item(item)
RGFA::Line::Group::Unordered#rm_item(item)
RGFA::Line::Group::Ordered#items
RGFA::Line::Group::Ordered#paths
RGFA::Line::Group::Ordered#append_item(item)
RGFA::Line::Group::Ordered#prepend_item(item)
RGFA::Line::Group::Ordered#rm_first_item
RGFA::Line::Group::Ordered#rm_last_item
```

```
RGFA::Line::Group::Ordered#captured_paths
RGFA::Line::Group::Ordered#captured_segments
RGFA::Line::Group::Ordered#captured_edges
RGFA::Line::Group::Unordered#induced_set
RGFA::Line::Group::Unordered#induced_segments_set
RGFA::Line::Group::Unordered#induced_edges_set
```

The Header

GFA files may contain one or multiple header lines (record type: H). These lines may be present in any part of the file, not necessarily at the beginning.

Although the header may consist of multiple lines, its content refers to the whole file. Therefore in RGFA the header is accessed using a single line instance (accessible by the `header` method). Header lines contain only tags. If not header line is present in the GFA, then the header line object will be empty (i.e. contain no tags).

Header lines cannot be connected to the RGFA as other lines (i.e. calling `connect` on them raises an exception). Instead they are merged to the existing header, when the `add_line(line)` method is called on the RGFA.

Multiple definitions of the predefined header tags

For the predefined tags (`VN` and `TS`), the presence of multiple values in different lines is an error, unless the value is the same in each instance (in which case the repeated definitions are ignored).

```
H VN:Z:1.0
# other lines
# ...
# the following raises an exception:
H VN:Z:2.0
```

Multiple definitions of custom header tags

If the tags are present only once in the header in its entirety, the access to the tags is the same as for any other line (see Tags chapter).

However, the specification does not forbid custom tags to be defined with different values in different header lines (which we name “multi-definition tags”). This particular case is handled in the next sections.

Reading multi-definitions tags

Reading, validating and setting the datatype of multi-definition tags is done using the same methods as for all other lines (see Tags chapter). However, if a tag is defined multiple times on multiple H lines, reading the tag will return an array of the values on the lines. This array is an instance of the subclass `RGFA::FieldArray` of Array.

```
H xx:i:1
H xx:i:2
H xx:i:3
# => gfa.header.xx value is RGFA::FieldArray[1,2,3]
```

Setting a tag

Calling set, if a tag was already defined, overwrites its value. For this reason, another method is defined, for supporting multi-definition tags: `add`. When `add(tagname, value)` is called on the RGFA header, if the tag does not exist, add will be a synonymous of set and simply create it. If it exists, it creates a field array (if a single value was present) or adds the new value to the existing field array (if multiple values were present).

```
# header.xx is not set
gfa.header.add(:xx, 1)
# header.xx is 1
gfa.header.add(:xx, 2)
# header.xx is a field array [1,2]
```

Modifying field array values

Field arrays can be modified directly (e.g. adding new values or removing some values). However, if this is done, some additional work is sometimes needed.

First, if values are added to the array, or its values are modified, the user is responsible to check that the array values remain compatible with the datatype of the tag (which can be checked by calling `validate_field(tagname)` on the header).

```
gfa.header.xx # => RGFAFieldArray[1,2,3]
gfa.header.xx << 4
gfa.header.xx << 5
gfa.validate_field(:xx)
```

Second, if the field array is modified using array methods (such as `map`) which return an Array class instance, this must be transformed back into a field array calling `to_rgfa_field_array(datatype)` method; thereby datatype can be set to the value returned by calling `get_datatype(tagname)` on the header.

```
gfa.header.map = gfa.header.map {|elem| elem + 1}.
                   to_rgfa_field_array(gfa.header.get_datatype(:xx))
```

String representation of the header

Note that when converting the header line to string, a single-line string is returned, eventually with multiple instances of the tag (in which case it is not standard-compliant). Similarly when calling #field_to_s on a field array tag, the output string will contain the instances of the tag, separated by tabs. However, when the RGFA is output to file or string, the header is splitted into multiple H lines with single tags, so that standard-compliant GFA is output. These can be retrieved using the `headers` method on the RGFA:

```
gfa.header.to_s # H VN:Z:1.0 xx:i:1 xx:i:2 (compact, but invalid GFA)
gfa.header.field_to_s(:xx) # => xx:i:1 xx:i:2
gfa.headers # => [] of three Header instances, with a single tag each
gfa.to_s # => (valid GFA)
         # H VN:Z:1.0
         # H xx:i:1
         # H xx:i:2
```

Custom records

According to the GFA2 specification, each line which starts with a non-standard record type shall be considered an user- or program-specific record.

RGFA allows to retrieve custom records and access their data using a similar interface to that for the predefined record types. It assumes that custom records consist of tab-separated fields and that the first field is the record type.

Validation of custom records is very limited; therefore, if you work with custom records, you may define your own validation method and call it when you read or write custom record contents.

Retrieving, adding and deleting custom records

The custom records contained in a RGFA object can be retrieved using its `custom_records` method. If no argument is provided, all custom records are returned. If a record type symbol is provided (e.g. `g.custom_records(:X)`), records of that type will be returned.

Adding custom records to and removing them from a RGFA instance is similar to any other line. So to delete a custom record, `disconnect` is called on the instance, or `rm(custom_record_line)` on the RGFA object. To add a custom record line, the instance or its string representation is added using `<<` on the RGFA, e.g. `g << "X\ta\tb"`.

Tags

As RGFA cannot know how many positional fields are present when parsing custom records, an heuristic approach is followed, to identify tags.

A field resembles a tag if it starts with `tn:d:` where `tn` is a valid tag name and `d` a valid tag datatype (see Tags chapter).

The fields are parsed from the last to the first. As soon as a field is found which does not resemble a tag, all remaining fields are considered positionals (even if another field parsed later resembles a tag).

This parsing heuristics has some consequences on validations. Tags with an invalid tag name (such as starting with a number, or with a wrong number of letters), or an invalid tag datatype (wrong letter, or wrong number of letters) are considered positional fields. The only validation available for custom records tags is thus the validation of the content of the tag, which must be valid according to the datatype.

Positional fields

The positional fields in a custom record are called `:field1, :field2, ...`. The user can iterate over the positional field names using the array obtained by calling `positional_fieldnames` on the line.

Positional fields are allowed to contain any character (including non-printable characters and spacing characters), except tabs and newlines (as they are structural elements of the line).

Due to the parsing heuristics mentioned in the Tags section above, invalid tags are sometimes wrongly taken as positional fields. Therefore, the user shall validate the number of positional fields (`line.positional_fieldnames.size`).

Extensions

The support for custom fields is limited, as RGFA does not know which and how many fields are there and how shall they be validated. It is possible to create an extension of RGFA, which defines new record types: this will allow to use these record types in a similar way to the built-in types. However, extending the library requires sligthly more advanced programming than just using the predefined record types. In the chapter Extending RGFA these extensions are discussed and an example is made.

Summary of custom-record related API methods

```
RGFA#custom_records
RGFA#custom_records(record_type)
RGFA#rm(custom_record_line)
RGFA#<<(custom_record_string)
RGFA::Line::CustomRecord#disconnect
RGFA::Line::CustomRecord#positional_fieldnames
RGFA::Line::CustomRecord#field1/field2/...
```

Comments

GFA lines starting with a `#` symbol are considered comments. In RGFA comments are represented by instances of RGFA::Line::Comment. They have a similar interface to other line instances (see below), with some differences, e.g. they do not support tags.

Comments in RGFA objects

Adding a comment to a RGFA object is done similary to other lines, by using the `RGFA#<<(line)` method. The comments of a RGFA object can be accessed using the `comments` method. This returns an array of comment line instances. To remove a comment from the RGFA, first find the instance (using the #comments array), then call `disconnect` on the line instance or "rm(line)"' on the RGFA object (passing the instance as parameter).

Examples:

```
g << "# this is a comment"
g.comments.map(&:to_s) # => ["# this is a comment"]
g.comments[0].disconnect # or g.rm(g.comments[0])
g.comments # => []
```

Accessing the comment content

The content of the comment line, excluding the initial +#+ and eventual initial spacing characters, is included in the field +content+.

The initial spacing characters can be read/changed using the +spacer+ field. The default value is a single space.

Tags are not supported by comment lines. If the line contains tags, these are nor parsed, but included in the +content+ field. Trying to set or get tag values raises exceptions.

Summary of comments-related API methods

```
RGFA#<<(comment_line)
RGFA#comments
RGFA::Line::Comment#disconnect
RGFA#rm(comment_line)
RGFA::Line::Comment#content/content=
RGFA::Line::Comment#spacer/spacer=
```

Errors

All exception raised in the library are subclasses of RGFA::Error. This means that `rescue RGFA::Error` catches all library errors.

Different types of errors are defined and are summarized in the following table:

Error	Description	Examples
Version	An unknown or wrong version is specified or implied	"GFA0"; or GFA1 in GFA2 context
Value	The value of an object is invalid	a negative position is used
Type	The wrong type has been used or specified	Z instead of i used for VN tag; Hash for an i tag
Format	The format of an object is wrong	a line does not contain the expected number of fields
NotUnique	Something should be unique but is not	duplicated tag name or line identifier
Inconsistency	Pieces of information collide with each other	length of sequence and LN tag do not match
Runtime	The user tried to do something which is not allowed	editing from/to field in connected links
Argument	Problem with the arguments of a method	wrong number of arguments in dynamically created method
Assertion	Something unexpected happened	there is a bug in the library

Some error types are generic (such as RuntimeError and ArgumentError), and their definition may overlap that of more specific errors (such as ArgumentError, which overlaps ValueError and TypeError). The user should not rely on the type of error alone, but rather take it as an indication. The error message tries to be informative and for this reason often prints information on the internal state of the relevant variables.

Assertion errors are reserved for those situation where something is implied by the programmer (e.g. a value is implied to be positive at a certain point of the code). It the checks fails, an assertion error is raised. The user may report the problem, as this may indicate a bug (unless the user did something he was not supposed to do, such as calling an API private method).

Graph operations

Some graph operations are provided by the RGFA library. These are described in the RGFA1 paper and in the API documentation. Note that some operations are completed by additional features, which are available when RGFATools is used.

Extending RGFA

The RGFA library is designed to be easily extended, although its extensions requires more knowledge of the Ruby languange, than what is necessary for merely using the library.

The GFA2 format can be extended by defining new line types. These are handled using the custom records functionality, but the support is limited: e.g. validation, parsing of the field content, references to other lines and access to fields by name are not possible. All this is made possible by extensions.

An example of user-specific record types

This chapter gives an example on how to extend the library to define an user-specific record type and custom field datatypes. As an example, we will define a record type for metagenomics applications with code M. This will have the role to define taxon-specific subgraphs, by putting segments in relation with a taxon. The taxa themselves will be declared in lines with code T:

Each T line will contain: - tid: a taxon ID - name: an organism name (text field) - the tags may contain an URL tag, which will point to a website, describing the organism (UL tag, string)

Each M line will contain: - mid: an optional assignment ID - tid: a taxon ID - sid: a reference to a segment - score: an optional Phred-style integer score, which will define an error probability of the assignment of the segment to a taxon

Here is an example of GFA containing the new line types:

```
S A 1000 *
T B12_c
M 1 taxon:123 A 40 xx:Z:cjaks536
M 2 taxon:123 B * xx:Z:cga5r5cs
```

```
S B 1000 *
M * B12_c B 20
T taxon:123 UL:http://www.taxon123.com
```

Subclassing RGFA::Line

Defining a new record type for RGFA requires to create a new subclass of the RGFA::Line class. Thereby some constants must be defined:

- `RECORD_TYPE` must contain the record type as symbol.
- `POSFIELDS` is an array of symbols, indicating the sequence of positional fields in the record
- `PREDEFINED_TAGS` contain an array of predefined optional tag names.
- `DATATYPE` is an hash. Each key is a symbol, either contained in POSFIELDS or in PREDEFINED_TAGS. The value is a datatype symbol: see the RGFA::Field module for a list of possible datatypes.
- `NAME_FIELD` is the field which contains the name of the line, if any
- `STORAGE_KEY` is the field which shall be used as a key for storing references of the line in RGFA; for custom subclasses, set it to `:name` if the line has a name field, to `nil` otherwise
- `FIELD_ALIAS` ia an hash which contain aliases to field names; it may be empty
- `REFERENCE_FIELDS` is a list of fields which contain references (or arrays of references) to other lines. The references may contain an orientation.
- `BACKREFERENCE_RELATED_FIELDS` is a list of fields which shall not be changed in a connected line without potentially invaliding backreferences to the line. In the predefined line types, these are the fields containing match coordinates in GFA2 edges (as they change their nature as internal, dovetails or containments) and the orientation and overlap fields in GFA1 links.
- `DEPENDENT_LINES` and `OTHER_REFERENCES` are lists of names of references collections, which will contain backreferences to other line types (which refer the line type in their fields). E.g. for a segment, the list contain the `:fragments` symbol, indicating that a collection shall be initialized, which will contain backreferences to the fragments which reference the segment. Disconnection is cascaded to lines in the collections named in DEPENDENT_LINES but not to those named in OTHER_REFERENCES.

For our example, we will define the subclasses for record types T and M.

```ruby
class RGFA::Line::Taxon < RGFA::Line

  RECORD_TYPE = :T
  POSFIELDS = [:tid, :desc]
  PREDEFINED_TAGS = [:UL]
  DATATYPE = {
    :tid => :identifier_gfa2,
    :desc => :Z,
    :UL => :Z,
  }
  NAME_FIELD = :tid
  STORAGE_KEY = :name
  FIELD_ALIAS = {}
  REFERENCE_FIELDS = []
  BACKREFERENCE_RELATED_FIELDS = []
  DEPENDENT_LINES = [:metagenomic_assignments]
  OTHER_REFERENCES = []

  apply_definitions

end

class RGFA::Line::MetagenomicAssignment < RGFA::Line

  RECORD_TYPE = :M
  POSFIELDS = [:mid, :tid, :sid, :score]
  PREDEFINED_TAGS = []
  DATATYPE = {
    :mid => :optional_identifier_gfa2,
    :tid => :identifier_gfa2,
    :sid => :identifier_gfa2,
    :score => :optional_integer,
  }
  NAME_FIELD = :mid
  STORAGE_KEY = :name
  FIELD_ALIAS = {}
  REFERENCE_FIELDS = [:tid, :sid]
  BACKREFERENCE_RELATED_FIELDS = []
  DEPENDENT_LINES = []
  OTHER_REFERENCES = []

  apply_definitions

end
```

Enabling the references

If reference fields have been defined (as in the previous example of M, where tid is a reference to a taxon line and sid is a reference to a segment line), a private `initialize_references` method shall be provided, which is called when a line of the type is connected to a RGFA instance.

In particular, the method shall change all identifiers in the reference fields into references to lines in the GFA (either existing lines or virtual lines, which is the way RGFA handles forward-pointing references).

If the referenced line is not yet available, but it may be defined by the GFA at a later time, the method will create a virtual line. In our example, we know that the reference is to a segment or a taxon line. If we would not know that we would instantiate RGFA::Line::Unknown.

When the field content itself is a reference, the content cannot be changed directly (using set would raise an exception, as the line is already connected when the initialize_referneces method is called). Therefore, the private line method set_existing_field shall be used, with `set_reference: true`. If the reference field contains an oriented line or an array instead, references can be edited directly.

```
class RGFA::Line::MetagenomicAssignment

  def initialize_references
    s = @rgfa.segment(sid)
    if s.nil?
      s = RGFA::Line::Segment::GFA2.new([sid.to_s, "1", "*"],
                                        virtual: true, version: :gfa2)
      s.connect(@rgfa)
    end
    set_existing_field(:sid, s, set_reference: true)
    s.add_reference(self, :metagenomic_assignments)

    t = @rgfa.line(tid)
    if t.nil?
      t = RGFA::Line::Taxon.new([tid.to_s, ""],
                                virtual: true, version: :gfa2)
      t.connect(@rgfa)
    end
    set_existing_field(:tid, t, set_reference: true)
    t.add_reference(self, :metagenomic_assignments)
  end
  private :initialize_references

end
```

The method defined backreferences to the new line in the segment and taxon instances, using :metagenomic_assignments as name for the collection of backreferences in S or T lines to lines of type M. For taxa, this collection has been defined in the class definition above. For segments, we will need to add this collection to the segment definition and redefine the reference getters methods. As lines of type M will be dependent on S lines (ie they shall be deleted if the referred segment line is deleted), we will add it to the DEPENDENT_LINES list. In case of no dependency, we would use the OTHER_REFERENCES list instead.

```
class RGFA::Line::Segment::GFA2
  DEPENDENT_LINES << :metagenomic_assignments
  define_reference_getters
end
```

Recognizing the record type code

When parsing lines starting with the code for the new record type, we want RGFA to return an instance of the correct subclass of Line.

To obtain this, the `subclass` class Method of `RGFA::Line` must be extended to handle the new record_type symbol, for GFA2 or unknown version records. It must return a class (the new subclass of RGFA::Line). The new record symbols must also be added to the gfa2 specific symbols list in `RECORD_TYPE_VERSIONS[:specific][:gfa2]`.

In our example the method `subclass` will be patched as follows:

```
class RGFA::Line
  class << self
    alias_method :orig_subclass, :subclass
    def subclass_GFA2(record_type, version: nil)
      if version.nil? or version == :gfa2
        case record_type.to_sym
        when :M then return RGFA::Line::MetagenomicAssignment
        when :T then return RGFA::Line::Taxon
        end
      end
      orig_subclass(record_type, version: version)
    end
  end
  RECORD_TYPE_VERSIONS[:specific][:gfa2] << :M
  RECORD_TYPE_VERSIONS[:specific][:gfa2] << :T
end
```

Allowing to find records

Both record types T and M define a name field. This allows to find record of the types using the `line()` method of the `RGFA` class, as well as allowing to replace virtual T lines created while parsing M lines, with real T lines, when these are found. For this to work, the codes must be added to the list `RECORDS_WITH_NAME` of the `RGFA` class:

```
RGFA::RECORDS_WITH_NAME << :T
RGFA::RECORDS_WITH_NAME << :M
```

Defining a field datatype

When new subclasses of line are created, it may be necessary or useful to create new datatypes for its fields. For example, we used :identifier_gfa2 for the tid field in the M and T records. However, we could made the field syntax stricter, and require that the content of the field must be either a reference to the NCBI taxonomy database or a custom identifier. In the first case, it will need to be in the form `taxon:<n>`, where `<n>` is a positive integer. In the second case, it will need to be a combination of letters, numbers and underscores (thereby `:` will not be allowed).

A module must be created, which handles the parsing and writing of fields with the new datatype. The module shall define six module functions (see the API documentation of the RGFA::Field module for more detail). Decode and unsafe_decode take a string as argument and return an appropriate Ruby object. Encode and unsafe_encode take a string representation or another ruby object and converts it into the correct string representation. Validate_encoded validates the string representation. Validate_decoded validates a non-string content of the field. The unsafe version of the decode and encode methods may provide faster results and are used if the parameters are guaranteed to be valid. The safe version must check the validity of the provided data.

```
module RGFA::Field::TaxonID

  def validate_encoded(string)
    if string !~ /^taxon:(\d+)$/ and string !~ /^[a-zA-Z0-9_]+$/
      raise RGFA::ValueError, "Invalid taxon ID: #{string}"
    end
  end
  module_function :validate_encoded

  def unsafe_decode(string)
    string.to_sym
  end
  module_function :unsafe_decode

  def decode(string)
    validate_encoded(string)
    unsafe_decode(string)
  end
  module_function :decode

  def validate_decoded(object)
```

```
    case object
    when RGFA::Line::Taxon
      validate_encoded(object.name.to_s)
    when Symbol
      validate_encoded(object.to_s)
    else
      raise RGFA::TypeError,
        "Invalid type for taxon ID: #{object.inspect}"
    end
  end
  module_function :validate_decoded

  def unsafe_encode(object)
    object = object.name if object.kind_of?(RGFA::Line::Taxon)
    object.to_s
  end
  module_function :unsafe_encode

  def encode(object)
    validate_decoded(object)
    unsafe_encode(object)
  end
  module_function :encode

end
```

The new datatype must have a symbol which identifies it. The symbol must be added to the `GFA2_POSFIELD_DATATYPE` list of the `RGFA::Field` module. An entry must be added to the `RGFA::Field::FIELD_MODULE` hash, where the symbol of the new datatype is the key and the value is the module.

```
RGFA::Field::GFA2_POSFIELD_DATATYPE << :taxon_id
RGFA::Field::FIELD_MODULE[:taxon_id] = RGFA::Field::TaxonID
```

Now the new datatype can be put into use by changing the datatype for the tid fields of the M and T lines:

```
RGFA::Line::Taxon::DATATYPE[:tid] = :taxon_id
RGFA::Line::MetagenomicAssignment::DATATYPE[:tid] = :taxon_id
```