`smiles_parser` design document

Laurent Fasnacht

May 23, 2013

Contents

List of Figures

1 Introduction

1.1 Intended audience

The indended audience of this document is both developers interested in using this library, and developers wanting to understand and improve the internals. We assume that the reader is already familiar with SMILES lines.

Section 2 is probably enough for the first category, while the latter should definitely also read section 3.

1.2 Purpose

This library, `smiles_parser`, is a TCL library to parse *Simplified Molecular Input Line Entry Specification* (SMILES [2]) into generic and simple to use TCL objects. It was originally written to improve LipidBuilder [1].

Unfortunately there are multiple non-compatible interpretation of what a SMILES is [3]. Therefore, this library was written to support the OpenSMILES specification [4], which is well-defined and quite common.

1.3 Example

Throughout this document, we will use the example of Vanilline, which is represented in figure 1. We will also use the following color convention:

- Chain numbers: black
- Atom numbers: red
- Bond numbers: blue

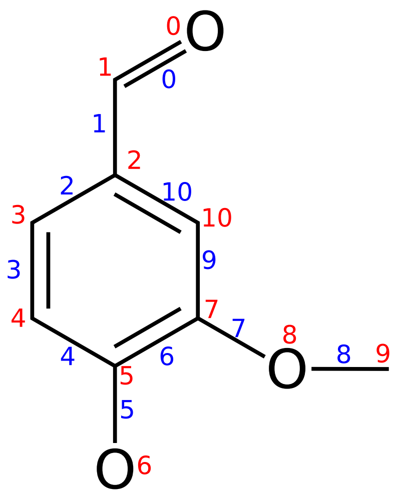

Figure 1: Representation of the vanillin molecule, `O=Cc1ccc(O)c(OC)c1`.

2 Interface

2.1 Representations

There are various way of representing a parsed SMILES string. Two of them were chosen, and will be presented thereafter.

2.1.1 Tree structure

The most natural way is a tree structure, because it directly arise from the specification. It is also the easiest structure to create (except finding the neighbors, which is quite difficult), and correspond closely to the string.

It's important to notice that it has the major drawback that it's difficult and computationnaly expensive to locate an element (i.e. "where's atom 9?"), because the only straightforward way to do this is recursive.

You can see in figure 2 the tree representation of the vanillin molecule (fig. 1).

2.1.2 Flat structure

To be able to quickly use the data, it is required to have a more efficient alternative. Since TCL doesn't have pointers, we need to duplicate data.

Therefore, we can define 3 lists:

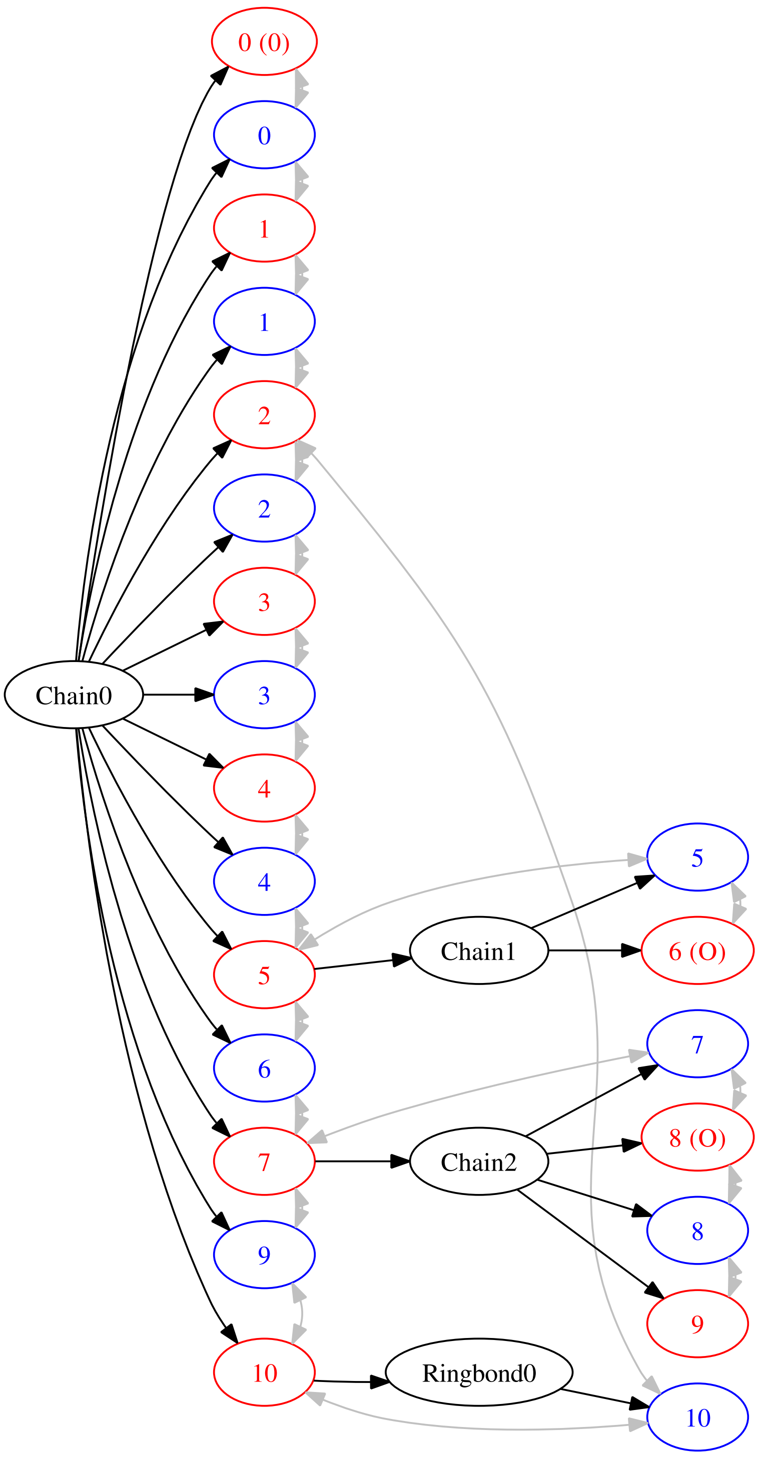

Figure 2: Tree representation of the vanillin molecule, `O=Cc1ccc(O)c(OC)c1`. The atoms are carbon, unless otherwise specified. The gray arrows are "neighbors" indication.

- List of chains (`chain[0]` is the full tree, like in figure 2)
- List of bonds
- List of atoms, including their subtree.

To have an intuitive view, you can look at figure 3.

2.2 Objects

Every object is implemented as a list. The first entry of the list is the type:

- `"chain"`
- `"bond"`
- `"ringbond"`
- `"atom"`

The next elements depend on the type, and will be defined in the corresponding subsection below. A general remark is that you shouldn't do direct indexing (except for the first entry, the type), and instead use the constants defined at the beginning of `smiles_parser.tcl`. By doing this way, you ensure that your code would not break if some field is added to the structure later on. For instance:

```
#this is correct
puts [lindex $::smiles_parser::atom_id $atom]

#This is WRONG
puts [lindex 1 $atom]
```

2.2.1 Chain

A chain is an list of bond and atoms. The object has the following attributes:

- `chain_id`: ID of the chain. Unique. Chain 0 is at the highest level.
- `chain_atomcount`: Number of atoms in the whole subtree of this chain.
- `chain_bondcount`: Number of bonds in the whole subtree of this chain.
- `chain_chaincount`: Number of chain in the whole subtree of this chain (includes this element. E.g. a chain containing no subchain will have a `chaincount` of 1).
- `chain_list`: List of elements (atom or bonds) which belong directly to this chain.

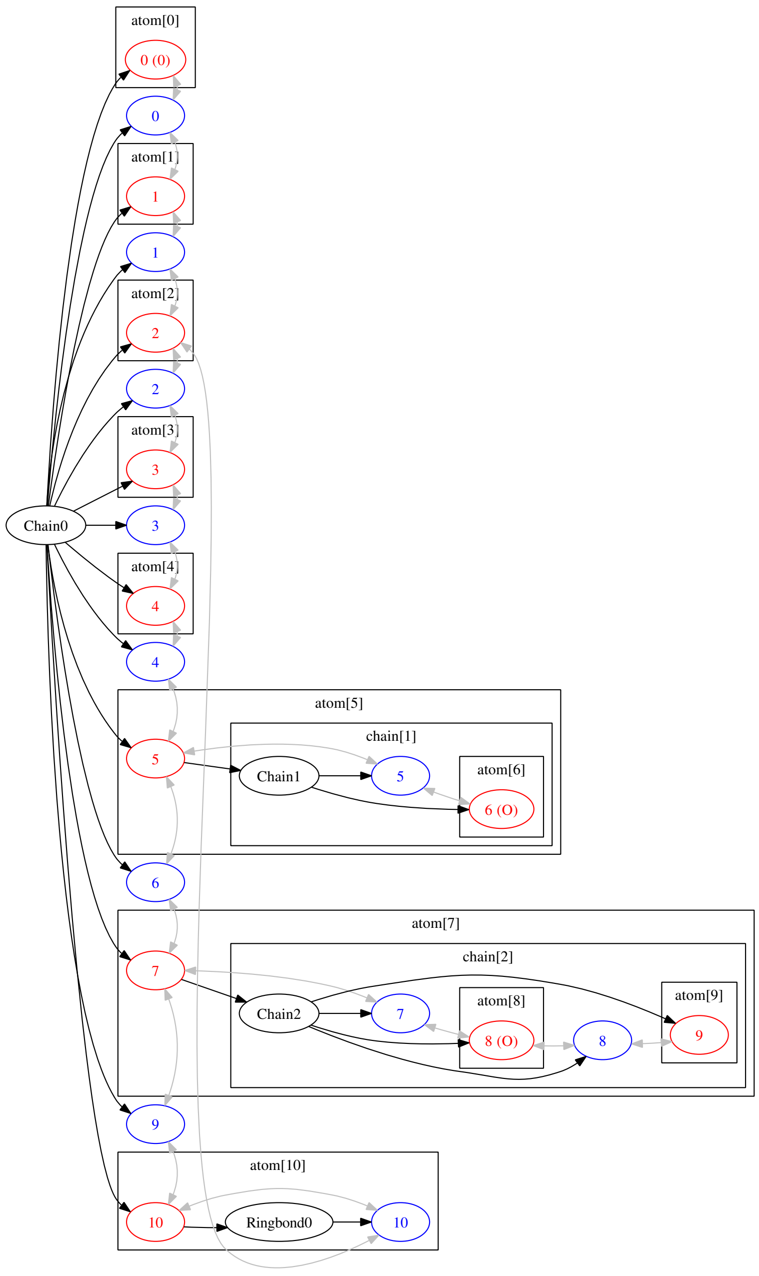

Figure 3: Flat representation of the vanillin molecule, `O=Cc1ccc(O)c(OC)c1`. The clusters represents the different table elements. Clusters for bonds were omitted (as they only return the bond).

2.2.2 Bond

- `bond_id`: ID of the bond. Unique.
- `bond_count`: Multiplicity of the bond.
  - `-`, `/`, `:`, `\`, (implicit), ⇒ 1
  - `=` ⇒ 2
  - `#` ⇒ 3
  - `$` ⇒ 4
- `bond_aromatic`: Is the bond aromatic?
  - yes (explicit `:`, or implicit between two aromatic atoms) ⇒ 1
  - no (explicit `-`, or implicit not between two aromatic atoms) ⇒ 0
- `bond_direction`: Contains information for cis-trans like `/` or `\` (`-` if not specified, even for multiple bonds or aromatic)
- `bond_linking`: List containing the two atom IDs linked by this bond.

2.2.3 Atom

- `atom_id`: ID of the atom. Unique.
- `atom_atomcount`: (total) number of atom in branches (does NOT include the current atom)
- `atom_bondcount`: (total) number of bond in branches
- `atom_chaincount`: (total) number of chain in branches
- `atom_symbol`: Chemical symbol of the atom (always in capital form, even for aromatic), or `*`
- `atom_aromatic`: Is the atom aromatic?
  - yes ⇒ 1
  - no ⇒ 0
- `atom_isotope`: Isotope of the atom. -1 if not specified.
- `atom_chiral`: Chiral part of the atom specification
- `atom_hcount`: Number of hydrogen
- `atom_charge`: Charge, as an integer

- `atom_class`: Atom class (integer)
- `atom_ringbonds`: List of ringbonds connected to this atom (only exist for the second part of the link, to avoid duplicate)
- `atom_branches`: List of chains, which are branches starting from this atom.
- `atom_linkedby`: List of bond IDs connected to this atom.

2.2.4 Ringbonds

- `ringbond_number`: Number used to declare the ringbond
- `ringbond_ref`: Reference counter (the same ringbond number can be used multiple time according to the specification. This number will increase each time it is reused)
- `ringbond_bond`: The corresponding bond (as defined in 2.2.2)
- `ringbond_toatom`: (**PRIVATE, DO NOT USE!**) Only for internal usage, use bond attribute instead (contains the target atom ID).

2.3 Functions

This library only export two public function. You should not try to call another function, as they are reserved for internal use.

2.3.1 `::smiles_parser::smiles_parse`

This function takes one argument, the SMILES string (which should be valid according to the OpenSMILES specification [4]), and returns the corresponding tree structure.

It may abort (`error`) if something is wrong, like:

- The SMILES string is invalid (two bonds one after the other, invalid atom, etc.)
- An explicit aromatic bond was written between two non-aromatic atoms.

2.3.2 `::smiles_parser::flatten`

This function takes as argument the tree returned by `::smiles_parser::smiles_parse`, and returns the corresponding flat structure.

When used correctly, it should never fail.

2.4 Example

Here's an example of running with the vanillin molecule:

```
lappend auto_path [file dirname [file dirname [file normalize [info script]]]]
package require smiles_parser

set data [::smiles_parser::smiles_parse "O=Cc1ccc(O)c(OC)c1"]
puts $data
puts ""
puts [::smiles_parser::flatten $data]
```

You can run this file by going in the `test` folder, and launching:
`tclsh test_vanillin.tcl`

3 Internals

3.1 Callgraph

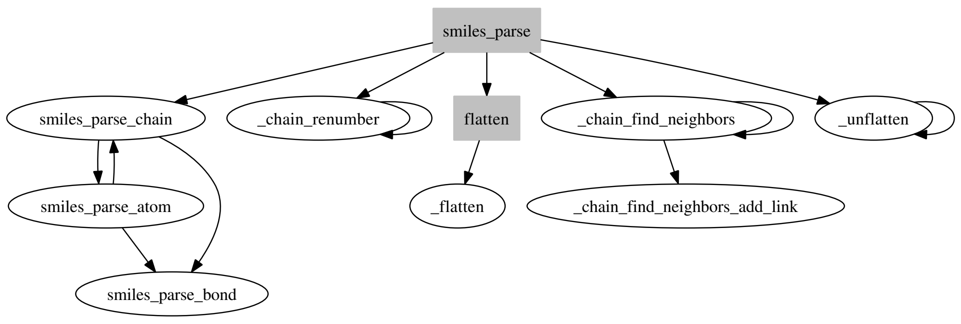

Figure 4: Callgraph the defined functions. Functions with gray background are the ones belonging to the interface. Functions starting with an underscore are private functions. `smiles_parse_chain`, `smiles_parse_atom` and `smiles_parse_bond` are not public, in the sense that they are not really useful by themselves, but may still be used without any side effects.

3.2 Description of the functions

In this section, we will briefly describe the intent which of the functions.

3.2.1 `smiles_parse`

The interface was already described in part 2.3.1. This basically do the following:

1. Parses the SMILES (`smiles_parse_chain` 3.2.3)
2. Number the different elements (`_chain_renumber` 3.2.6)
3. Flatten the structure (`flatten` 3.2.2)
4. Find neighbors (`_chain_find_neighbors` 3.2.7). It's easier to do so on if we can index directly.
5. Update the tree using information from the flattened structure. (`_unflatten` 3.2.10)

3.2.2 `flatten`

The interface was already described in part 2.3.2.

It is really simple: it initialize the internal variables for `_flatten` (3.2.9), calls it, and returns the result.

3.2.3 `smiles_parse_chain`

This function accepts a SMILES string and a boolean as parameters. The boolean indicates if the string should start with a bond (like in a branch) or not (main SMILES string).

The function returns a list containing the length parsed, and a `chain` (2.2.1) object.

`smiles_parse_chain` tries to alternatively parse an atom (`smiles_parse_atom` 3.2.4) and a bond (`smiles_parse_bond` 3.2.5). If for some reason one of these function fails, it either throws an error, or stops and returns what was parsed till now.

3.2.4 `smiles_parse_atom`

This function accepts a SMILES string as an argument, which is expected to start by an atom. It returns the length parsed and an `atom` (2.2.3) object.

Basically, it first tries to parse the real "atom" part, and then tries to find ringbonds (using `smiles_parse_bond` 3.2.5 to parse them), and finally parses the branches (using `smiles_parse_chain` 3.2.3 with the chain between the parentheses, and 1 as the second argument, since the branch should start by a bond).

3.2.5 `smiles_parse_bond`

This function accepts a SMILES string as an argument, which is expected to start by an bond (which may be implicit). It returns the length parsed and an `bond` (2.2.2) object. Note that the length parsed may be 0 (if the bond is implicit).

3.2.6 `_chain_renumber`

This function numbers the object provided using a recursive DFS strategy. It also resolves the ringbonds (only the second reference to a number remains, which mean that "open" ringbonds are discarded.

It returns the numbered object.

It uses the following global variables:

- `::smiles_parser::cur_id_atom`
- `::smiles_parser::cur_id_bond`
- `::smiles_parser::cur_id_chain`
- `::smiles_parser::cur_ringbond_links`.

While these three first variables only hold the id for the next element, the latter is a cache (dictionnary) to resolve the ringbonds. All these variables should be initialized before calling this function. See the source code of `smiles_parse` to see how.

3.2.7 `_chain_find_neighbors`

This function accepts a chain, and (facultative, if this is a branch) the previous atom ID. It traveses everything (DFS strategy), calling `_chain_find_neighbors_add_link` (3.2.8) for any link it discovers.

3.2.8 `_chain_find_neighbors_add_link`

This function has three arguments:

1. The first atom ID
2. The bond ID
3. The second atom ID

It updates in the cache (`::smiles_parser::cur_flatten_*`, see 3.2.9):

- For the bond: `bond_linking attribute`
- `For atoms: atom_linkedby attribute`

It also updates the bond to set it aromatic if both atoms are aromatic and the bond was implicit, and checks if the bond was specified aromatic that both ends are indeed aromatic.

Note that you'll have to call `_unflatten` (3.2.10) afterwards to recreate the full tree, with the new information.

3.2.9 `_flatten`

This function takes object provided as an argument, and by doing recursively a DFS, update the cache for the flattened structure. It also returns the new state of the cache. (list containing three elements: the list of atoms, the list of bonds, and the list of chains)

The cache is composed of three global variables:

- `::smiles_parser::cur_flatten_atom`
- `::smiles_parser::cur_flatten_bond`
- `::smiles_parser::cur_flatten_chain`

Basically, the idea is to set `::smiles_parser::cur_flatten_atom[x]` to be the atom with id `x`.

The cache should be initialized to a list of the correct size before calling this function. See for instance the source code of `flatten` (3.2.2).

3.2.10 `_unflatten`

This function takes as argument an element, and updates every element using the data from the cache (see 3.2.9).

It uses a DFS strategy. It is very important to first replace the element, and then do the recursion, as the cache may be unconsistent (we only update the top level object in the cache, so the subtree of one of its element is may not correspond).

References

[1] LipidBuilder,
http://lipidbuilder.epfl.ch/

[2] SMILES,
http://www.epa.gov/med/Prods_Pubs/smiles.htm

[3] SMILES Line Notation, OEChem TK,
http://www.eyesopen.com/docs/toolkits/current/html/OEChem_TK-python/SMILES.html

[4] OpenSMILES Specification,
http://www.opensmiles.org/opensmiles.html