Internal Report 2010–18 October 2010

Universiteit Leiden
Opleiding Informatica

Parsing and Conversion of

SMILES–strings to Molecular Graphs

Anton den Hoed

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Parsing and Conversion of SMILES-strings to Molecular Graphs

Anton den Hoed
Supervisors: Michael Emmerich, Johannes Kruisselbrink

October 29, 2010

Abstract

This paper presents a parser for Daylight SMILES that was created for use in the Molecule Evolution Tool ([1], [2]). Many existing SMILES parsers are fully hand–written, which results in source–code that is difficult to maintain. The parser in this project is implemented by means of a tokenizer and parser generator. Because the Molecule Evolution Tool requires that the input molecules are in Kekulé–form, a transformation algorithm was developed to convert from an aromatic definition to Kekulé–form.

Contents

1 SMILES **4**
- 1.1 OpenSMILES . . . 4
- 1.2 Other formats . . . 5
- 1.3 Specification . . . 5
  - 1.3.1 Atoms . . . 6
  - 1.3.2 Bonds . . . 6
  - 1.3.3 Branches . . . 7
  - 1.3.4 Cyclic structures . . . 8
  - 1.3.5 Disconnected structures . . . 9
  - 1.3.6 Isomeric SMILES . . . 9
- 1.4 Conventions . . . 12
  - 1.4.1 Hydrogen . . . 12
  - 1.4.2 Aromaticity . . . 12

2 Formal Languages and Parsing **14**
- 2.1 Formal Languages . . . 14
  - 2.1.1 Context–Free Grammars . . . 15
  - 2.1.2 Backus–Naur Form . . . 15
- 2.2 Parsing Tools . . . 16

3 Implementation **17**
- 3.1 Grammar and tokens . . . 17
  - 3.1.1 Tokens . . . 18
  - 3.1.2 Grammar . . . 19
- 3.2 Parser . . . 20
- 3.3 Post–processing . . . 21

4 Experiments **27**
- 4.1 Test Cases . . . 27
- 4.2 Performance . . . 29

5 Conclusions and Future Work **32**

A Source Code **35**

Introduction

Developing a medicine is very hard. There are millions of potential drug molecules, while most of the times only one molecule has the right properties. Testing all molecules until the right one is found takes a lot of time. Such amounts of time are not available because if a competing company finds the molecule first, they will get the patent. It is also too expensive to test each molecule, because the development cost should be earned back in a limited amount of time.

To avoid this brute–force method of finding the right molecule, researchers predict properties of molecules to find molecules with the desired properties. Doing this by hand is still very time consuming, so automated computer tools were developed. These tools are relatively simple compared to the class of tools which use optimization techniques to find potential good molecules.

One of the used optimization techniques are evolutionary algorithms [3]. These algorithms use the evolution principle from nature to search for the solution of a problem. In the context of finding drug–molecules this means: taking a *population* of molecules and let the best molecules *reproduce*. During the reproduction process the offspring is mutated and a new *population* is formed. This process can be repeated, and in theory a good working molecule will eventually be found.

In the last few years tools like the Molecule Evoluator [4] and the Molecule Evolution Tool [1] were created.

This project is part of the development of the Molecule Evolution Tool (METool). The METool uses an evolutionary algorithm to find molecules. Together with Pipeline Pilot [5] a loop (Figure 1) is created. After each iteration of the algorithm Pipeline Pilot will evaluate the results and gives feedback about the fitness of the molecules to the algorithm for the next iteration.

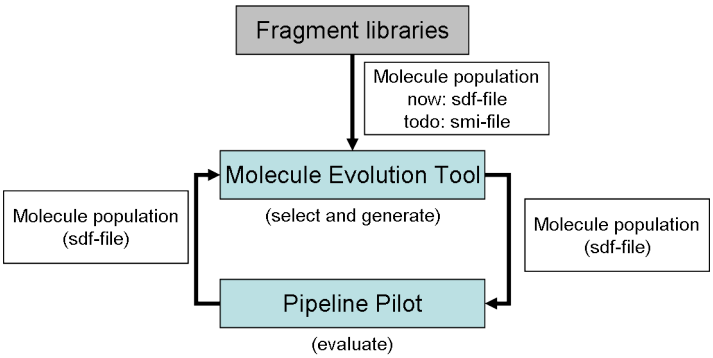


Figure 1: The loop of the Molecule Evolution Tool

METool contains a database of molecule fragments for use by the evolutionary algorithm. In the initial version these files are stored in the SDF–format (MDL Molfile), which is not a very efficient or intuitive way to store molecules (see Section 1.2). While looking for a better storage format for the molecule fragments Daylight SMILES [6] was selected. Daylight SMILES is a format that is more compact and intuitive (see Section 1.2 for a comparison of different formats).

A tool is needed to convert the SMILES–notation to the internal format of the METool. The internal format consists of a list of atoms and a list of bonds between atoms (Figure 2). This tool is called a parser.

Atoms

id	*symbol*
1	C
2	C
3	C

Bonds

$(Atom_1, Atom_2)$
(1, 2)
(2, 3)

Figure 2: The internal format for propane

Many chemistry libraries and toolkits have a SMILES parser. Some examples are CDK [7], RDKit [8], Daylight [9] and OpenEye [10]. All of these parsers are hand–written, and thus hard to maintain. This project has the goal of creating a SMILES parser using a context–free grammar and the standard parser toolchain from GNU (Flex [11], Bison [12]).

Chapter 1 will describe the SMILES–language, accompanied with examples of the explained features. Chapter 2 will introduce basic knowledge about context–free languages and parsers. In Chapter 3 the SMILES grammar and the implementation of the parser will be described. To test the working of the parser some experiments are done in Chapter 4. The conclusions can be found in Chapter 5.

Chapter 1

SMILES

SMILES (Simplified Molecular Input Line Entry System) is a way to represent molecular structures in a compact way as strings. Weininger [6] published the specification in 1970, and many providers of chemical databases (such as ZINC [13], ChemSpider [14], and ChEBI [15]) provide their database in SMILES–format. The standard that is used by most of these databases is the proprietary SMILES standard which was created by Daylight Chemical Information Systems Inc. [9]. The documentation on the Daylight website [16] is used as a reference work in the chemoinformatics community.

In the SMILES–format, a simple molecule like water can be expressed as `[H]O[H]`. A hydrogen atom, connected to a oxygen atom, connected to a hydrogen atom. Because a SMILES–string is linear, rings need to be encoded. For this language, all rings are broken and the removed bonds are numbered. For example this will result in `C1CCCCC1` for cyclohexane. It is also possible to specify properties of atoms such as isotope, chirality and charge by using square brackets around the atom. Uranium–235 is described as `[235U]`. These and other properties of SMILES will be described throughout this chapter.

1.1 OpenSMILES

In 2007, Dalke and James [17] started a project called OpenSMILES to create an open standard for SMILES. They were not satisfied by the fact that the SMILES standard is owned by a company, and that the standard is not perfect on some points, such as cis–trans definitions with slashes and notation for polymers and crystals which is not implemented. Their plan was to add several extensions (see [18]), provide a context–free grammar for the improved language and provide good documentation for the parser. After three years the project is still in progress, and not all parts of the documentation are finished. No documented cases of the use of OpenSMILES in a project could be found.

1.2 Other formats

SMILES is not the only format to store chemical structures. The different formats can be divided into two categories: connection table based and linear string based. Connection table based formats like MDL Molfile [19], PDB [20] and CML [21] usually contain a list of atoms and a matrix describing the bonds between those atoms. Linear string based formats like SMARTS [22], SLN [23], WLN [24] and InChI [25] describe the molecule structure as a single line string, from which the structure is inferred by a computer algorithm.

Both categories have their advantages and disadvantages. Connection table based formats are easier to interpret by a computer, however hard to interpret by a human. On the other hand, linear string based formats are easier to interpret by a human, and harder to interpret by a computer. Another advantage of linear string based formats is the compactness of notation.

To illustrate the differences in notation between the different notations, the notation of benzene will be given in a number of different languages (Figure 1.1).

```
 benzene
ACD/Labs0812062058

  6  6  0  0  0  0  0  0  0  0  1 V2000
    1.9050   -0.7932    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.9050   -2.1232    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7531   -0.1282    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7531   -2.7882    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.3987   -0.7932    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.3987   -2.1232    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  2  1  1  0  0  0  0
  3  1  2  0  0  0  0
  4  2  2  0  0  0  0
  5  3  1  0  0  0  0
  6  4  1  0  0  0  0
  6  5  2  0  0  0  0
M  END
$$$$


c1ccccc1


1/C6H6/c1-2-4-6-5-3-1/h1-6H
```

Figure 1.1: Benzene defined in MDL Molfile, SMILES and InChI

1.3 Specification

In the following part the notation of Daylight SMILES will be explained. This specification is largely based on the description of [16]. It is divided into sections

that cover feature of the language. Each section has a definition of the syntax, an explanation of the feature in a chemical context and several examples.

1.3.1 Atoms

In SMILES atoms are represented by their symbol in the periodic table enclosed in square brackets. If the symbol consists of two characters the second character should be lower case. Several properties can be defined inside the square brackets. These properties are:

- Isotope,
- Chirality,
- Hydrogen count,
- Charge,
- Class.

The notation of an atom is valid if it complies with the following regular expression:

$$'[' \; isotope? \; symbol \; chiral? \; hcount? \; charge? \; class? \; ']'$$

.

In Table 1.1 some examples of atoms with specified properties can be found.

`[Ti]`	Titanium
`[235U]`	Uranium-235
`[C@]`	Carbon with counter-clockwise chirality
`[CH3]`	Carbon with 3 Hydrogen atoms attached to it
`[O-]`	Oxygen with a negative charge
`[N:42]`	Nitrogen with a class number attached to it

Table 1.1: Examples of atom properties

Of course combinations are possible, like `[14C@H2:56]`. An exception to the requirement that all symbols are enclosed by square brackets is that the atoms in the *organic subset* (B, C, N, O, P, S, F, Cl, Br and I) can be written without square brackets. This is only possible if the number of attached hydrogens conforms to the *lowest normal valence* of the atom minus the number of explicitly defined bonds. The lowest normal valences can be seen in Table 1.2.

In some cases it might be needed to define a molecule which is not entirely known. To support this, it is possible to replace an atom with a wildcard (`*`) symbol. This symbol will be treated as *any symbol*.

1.3.2 Bonds

Bonds between atoms can be defined by connecting them with a bond character. The bond characters are '−', '=', '#' and ':' for respectively single, double, triple

Atom	Valence
B	3
C	4
N	$3, 5$
O	2
P	$3, 5$
S	$2, 4, 6$
Cl	1
Br	1
I	1

Table 1.2: Lowest normal valences of the organic subset

and aromatic bonds. In OpenSMILES it is also possible to define quadruple bonds using '$'. If there is no character between two atoms, SMILES assumes that the atoms are bonded by a single bond.

`CC`	ethane	CH_3CH_3
`C-C`	ethane	CH_3CH_3
`C=O`	formaldehyde	CH_2O
`O=C=O`	carbon dioxide	CO_2
`C#N`	hydrogen cyanide	HCN
`[H][H]`	molecular hydrogen	H_2

Table 1.3: Examples of molecules with different bond orders

1.3.3 Branches

Branches in a molecule can be defined by placing the branches between parentheses, `(...)`. Branches can be nested `(...(...))` and stacked `(...)(...)`. Some examples of molecules with branches can be found in Figure 1.2.

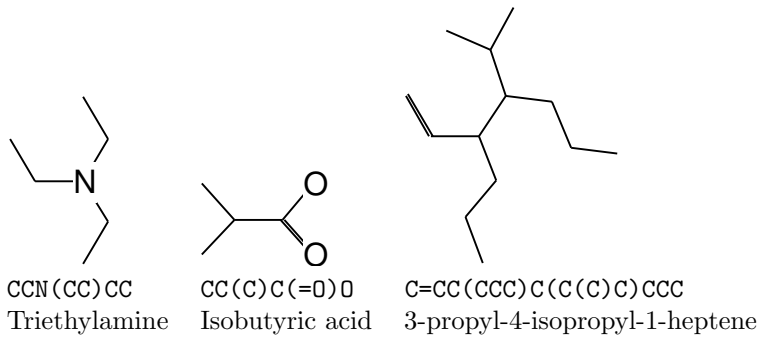
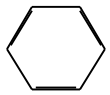


Figure 1.2: Examples of molecules with branches

1.3.4 Cyclic structures

To represent cyclic structures in a string, it is needed to break one bond in each ring. The atoms at each end of the broken bond get a number, which is appended to the atom definition in the string. A simple example is benzene, shown in Figure 1.3.



`c1ccccc1`

Figure 1.3: benzene

It is possible to have several rings opened at the same time. The first nine rings can be numbered by 1 to 9. When larger numbers are needed a '%' character should be prepended to the number. While the maximum number of open rings is 99 (`%99`), it is still possible to create structures with more than 99 rings because ring number can be reused.

Molecules can have many different equally valid SMILES strings, as can be seen in Figure 1.4. The first option *(a)* follows the ring, while the second option *(b)* considers the ring past `Br` as a branch.

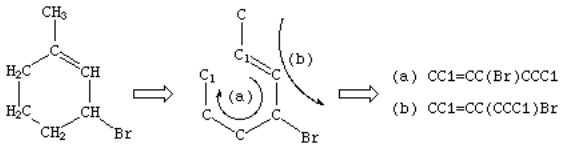


Figure 1.4: Different SMILES strings for a single molecule

An example of a molecule with an atom with multiple ring numbers is cubane, which can be seen in Figure 1.5. The corresponding SMILES string is `C12C3C4C1C5C4C3C25`

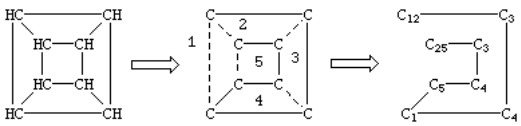


Figure 1.5: Atoms with multiple ring numbers

An example of a molecule with ring number `1` that is re–used can be seen in Figure 1.6

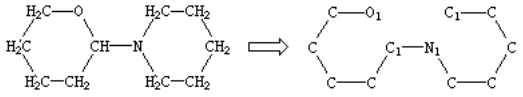


Figure 1.6: Re–use of ring numbers

1.3.5 Disconnected structures

Disconnected compounds can be written with a dot (`.`) as separator. For example, two protons are written as `[H+].[H+]`. The order and charge of the molecules in a disconnected compound does not matter. When a dot is encountered inside a molecule, there is no bond between the atoms on the left and right side of the dot. This does not imply that the molecule is disconnected. If a ring number appears in different molecules in a string, then there is a bond between the atoms. For example, `C1.C1` is the same as `CC`.

1.3.6 Isomeric SMILES

In SMILES it is possible to specify the isotopes, configuration around double bonds (*cis-trans*) and chirality. This section will give a short overview of Isomeric SMILES, because these features are rarely used in finding drug molecules.

Isotopic specification

Isotopes can be specified by prepending them to the atomic symbol of a bracketed atom. If there is no isotope defined, there is no mass defined at all. SMILES does not take a common isotope into account. Examples of isotopic specification can be found in Table 1.4.

SMILES	Name
`[12C]`	carbon-12
`[13C]`	carbon-13
`[C]`	carbon (unspecified mass)
`[13CH4]`	C-13 methane

Table 1.4: Examples of isotopic specification

Configuration around double bonds

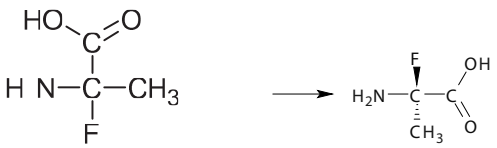
To discriminate between *cis* and *trans* configurations of a molecule, special bond characters (`\` and `/`) are available. If the slash is a forward slash (ex. `F/C`) or a backward slash (ex. `F\C`), it means that `F` respectively is 'below' or 'above' `C`. So if both slashes are in the same direction the atoms are at different sides. If the slashes are different the atoms are at the same side.

Z-1,2-difluorethene	E-1,2-difluorethene
`F/C=C\F`	`F/C=C/F`
`F\C=C/F`	`F\C=C\F`

Figure 1.7: Examples of configuration around double bonds

Configuration around tetrahedral centers

The configuration around tetrahedral centers can be defined by appending a `@` or `@@` to the atom symbol in a bracketed atom. A single `@` means that the order of connected atoms is counter–clockwise, while `@@` means clockwise.



NC(C)(F)C(=O)O	N[C@](C)(F)C(=O)O
NC(F)(C)C(=O)O	N[C@@](F)(C)C(=O)O
unspecified	specified

Figure 1.8: Examples of configuration around tetrahedral centers

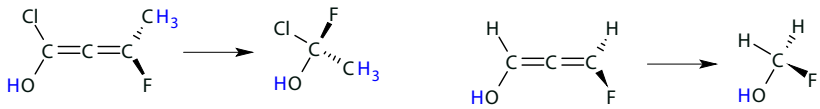
When looking from the `N` to the `C` attached to it, the connected atoms are listed counter–clockwise in the first case. In the second case the atoms are listed clockwise.

General chiral specification

Because tetrahedral chirality is not the only type of chirality, SMILES also supports other types. These supported types are:

Tetrahedral
Configurations around tetrahedral centers can be denoted by `@TH1` (counter–clockwise)and `@TH2` (clockwise). The codes `@` and `@@` are shorthands for `@TH1` and `@TH2` respectively.

Allene–like
This type defines configurations around an atom of degree 2 (the chiral center is the central atom with double bonds). Allene-like chirality is defined by `@AL1` and `@AL1`. If an atom has degree two, this type of chiral specification can be defined by `@` for `@AL1` and `@@` for `@AL2`

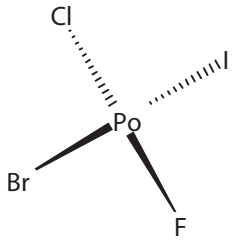


OC(Cl)=[C@]=C(C)F
OC(Cl)=[C@AL1]=C(C)F

OC=[C@]=CF
OC([H])=[C@AL1]=C([H])F

Figure 1.9: Examples of Allene–like configurations

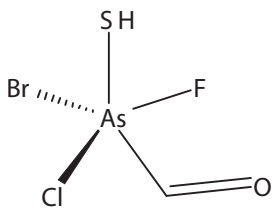
Square–planar
This type is not the default chiral specification for atoms of degree 4. It can be specified by `@SP1`, `@SP2` and `@SP3`. Shorthands are not allowed in this case, because square–planar is not the default for degree 4.



F[Po@SP1](Cl)(Br)I	SP1 lists in a "U shape"
F[Po@SP2](Br)(Cl)I	SP2 lists in a "4-shape"
F[Po@SP3](Cl)(I)Br	SP3 lists in a "Z shape"

Figure 1.10: Example of a Square–planar configuration

Trigonal–bipyramidal
This is the default chiral class for degree 5. It can be specified by `@TB1`, `@TB2`, ..., `@TB20`.

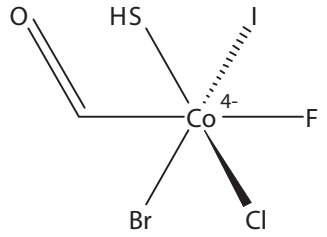


s[As@@](F)(Cl)(Br)C=O
O=C[As@](F)(Cl)(Br)S

Figure 1.11: Example of a Trigonal–bipyramidal configuration

Octahedral
The default chiral class for degree 6 is Octahedral. It can be specified by `@OH1`,

`@OH2`, ..., `@OH30`.



S[Co@@](F)(Cl)(Br)(I)C=O
O=C[Co@](F)(Cl)(Br)(I)S

Figure 1.12: Example of a Octahedral configuration

1.4 Conventions

In this section the conventions for explicitly defining hydrogen atoms and defining aromaticity are explained.

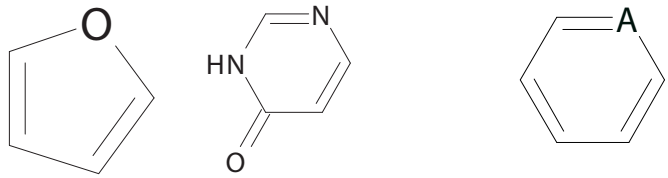
1.4.1 Hydrogen

In most organic structures it is not necessary to specify hydrogen atoms explicitly, because the number of attache hydrogen atoms can be deduced using the valences of the atoms. If it is necessary to specify the hydrogen atoms there are two ways to do it:

- **Explicitly by count**: If an atom is written inside brackets, the number of attached hydrogens can be specified by writing `Hn`, where `n` is the number of attached hydrogens
- **As explicit atoms**: By writing `[H]` for each hydrogen atom

1.4.2 Aromaticity

In SMILES aromaticity can be defined in two ways, the first is using lowercase atom symbols and the second is defining aromatic bonds by using the : symbol. Because aromatic definitions can be difficult to handle, another notation called the Kekulé form can be used. If the Kekulé form is used, one of the configurations of an aromatic molecule is explicitly written out using single and double bonds. If the aromatic structures are written using lowercase atom symbols or using :–bonds, the parser interprets the bonds between atoms written using lowercase symbols and :–bonds as aromatic bonds. If the aromatic structures are written using the Kekulé form the parser interprets the bonds as normal single and double bonds. A simple example to show the difference between the different notations is benzene. Benzene can be written as `c1ccccc1`, `C:1C:C:C:C:C1` or `C1=CC=CC=C1`.



C1=COC=C1	C1=CN=C[NH]C(=O)1	C1=C*=CC=C1
c1cocc1	c1cnc[nH]c(=O)1	c1c*ccc1

Figure 1.13: Examples of aromatic molecules

Chapter 2

Formal Languages and Parsing

This chapter will introduce some basic concepts of formal languages and parsing.

Parsing is the process of matching an input stream of characters or tokens with a defined formal grammar. The parser determines if the input is valid according to the grammar, and thus if the input is part of the language. The parser also generates a parse–tree which contains the derivations that are used to match the input. This parse–tree can be used by computer programs to do processing and computation with the input and its structure.

In practice the parsing process is a pipeline (Figure 2.1). The input stream is given to a tokenizer, which splits the input into separate tokens. These tokens are given to the parser, which has code attached to each grammar production to process the input. In the case of the SMILES–parser a separate post–processing step will Kekulize the molecular graph that results from the parsing stage.

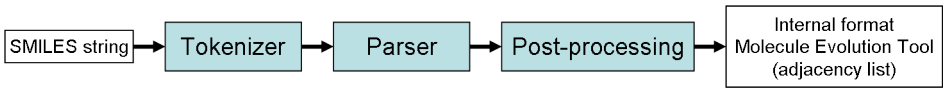


Figure 2.1: The parser pipeline of the SMILES–parser

2.1 Formal Languages

A formal language can be seen as a set of words. These words are a sequence of characters or tokens. Formal grammars are used to define which words are part of a language. Such a grammar defines a set of substitution rules of the form $\alpha \longrightarrow \beta$, which can be read as substitute α with β in the string at hand. The grammar also defines a starting symbol, which is used to form words in the language in one or more applications of rules from the grammar. Strings like α consist of terminal and non–terminal symbols. Non–terminal symbols

are symbols that can be replaced by a production rule, while terminal symbols cannot.

There are different types of formal languages and formal grammars. The classification of formal languages is known as the Chomsky–hierarchy [26], and contains the following language classes:

Type	**Languages**	**Productions**
Type–0	Recursively enumerable (RE)	$\alpha \longrightarrow \beta$
Type–1	Context–sensitive (CSL)	$\alpha A \beta \longrightarrow \alpha \gamma \beta$
Type–2	Context–free (CFL)	$A \longrightarrow \alpha$
Type–3	Regular (Reg.)	$A \longrightarrow aB$, $A \longrightarrow a$

RE grammars are the least restrictive and each type is a subset of the type above ($Reg. \subseteq CFL \subseteq CSL \subseteq RE$). When describing programming languages or languages like SMILES Context–free grammars are used.

2.1.1 Context–Free Grammars

Context–free grammars are grammars with rules like $S \longrightarrow \alpha$. The left hand side of the arrow contains only a single non–terminal, and the right hand side of the arrow can contain λ or a string of terminals and non–terminals. Not every language is context–free, because not every language can be defined by rules with a single non–terminal as left hand side. Languages for which it is possible to describe them with a context–free grammar are called context–free languages.

To define a context–free grammar (CFG) a mathematical structure is needed. A CFG is defined by a tuple $G = (V, \Sigma, S, P)$. Where V is a set of non–terminals, Σ is an alphabet, S is the start symbol for the derivations and P is a relation that defines all possible productions.

A simple example of a context–free language is the language $\{a^n b^n | n \geq 1\}$, which is the language of all strings with at least one a followed by the same amount of b's. The grammar will have $V = S$, $\Sigma = a, b$, $S = S$ and P is defined as:
$S \longrightarrow aSb$
$S \longrightarrow ab.$

2.1.2 Backus–Naur Form

To have a structured way of describing the productions of a CFG, Backus and Naur [27] developed a language called BNF (Backus–Naur Form) to describe CFG's. A BNF–definition consists of rules of the following form.

```
<symbol> ::= _expression_
```

Expressions consist of non–terminals written like `<symbol>`, terminals written like strings `"term"` and the | symbol to indicate a choice between several rules. The example language from the previous section can be described in BNF as follows:

```
<start> ::= "a" <start> "b" | "a" "b"
```

In the Extended–BNF (EBNF) standard several improvements were made to the BNF standard. The main advantages of EBNF are that it is possible to define zero or one occurrences of a symbol as `<symbol>?` and zero or more occurrences as `<symbol>*`.

2.2 Parsing Tools

This section will introduce the concepts of lexical analyzers and parser generators.

Lexical Analyzer

To parse a language like SMILES it is needed to split the strings in small parts called lexemes. Some examples of lexemes are `C`, `[` and `@`. To make these lexemes usable for the parser, a lexical analyzer matches all lexemes in the string with pre–defined patterns (regular expressions). When there is a match the lexical analyzer will produce a token. A token is a symbol that indicates the type of the lexeme.

For the SMILES–parser Flex (fast lexical analyzer generator) was used. More information can be found at the website of Flex [11].

Parser generator

A parser generator is a tool that converts a grammar to a working parser. The grammar is defined in a certain syntax and programming code can be added to each production to define the semantics. The parser generator generally converts the definition to a pre–calculated table with all parsing decisions and a part with the programming code.

For the SMILES–parser the GNU Bison parser generator was used. More information can be found at the website of GNU Bison [12].

Chapter 3

Implementation

This chapter will describe the implementation of the SMILES–parser. First the grammar and the tokens that are used are introduced and explained. After that the design considerations are discussed. Then the conversion from the graph with aromatic bonds to a graph in Kekulé–form will be described.

3.1 Grammar and tokens

The grammar that is used to create the SMILES–parser is derived from the OpenSMILES grammar [28]. OpenSMILES is not identical to Daylight SMILES in several aspects. A minor difference is that OpenSMILES supports quadruple bonds, where Daylight SMILES does not. A more serious difference was that OpenSMILES only supports ringbonds and branches in a specific order,

`branched_atom ::= atom ringbond* branch*`,

while Daylight SMILES has no specific order. This problem can be solved by changing the production into

`branched_atom ::= atom ringbond* branch* ringbond*`.

Another problem is that Bison does not support the repetition operators of EBNF (`?` and `*`). These operators need to be converted. To convert the zero or one operator the following transformation can be applied:

`<sym>?` $\longrightarrow$ `<sym_opt>`,
`<sym_opt> ::= <sym> | ""`,

where `""` is the empty string.

And for the zero or more operator:

`<sym>*` $\longrightarrow$ `<sym_kleene>`,
`<sym_kleene> ::= <sym_kleene> <sym> | <sym> | ""`.

Testing the basic grammar resulted in a shift/reduce problem, where the parser could not decide between two options. The problem was located in the grammar

rule for isotopes. `isotope: NUMBER`, where `NUMBER` was a token with the pattern `{DIGIT}+`. It was solved by creating a new grammar rule `number: number DIGIT | DIGIT` and changing the isotope rule into `isotope: number`.

3.1.1 Tokens

This section defines the tokens that are used in the SMILES–parser. These tokens resemble features in SMILES. For example, `BOND_2` represents a double bond and `CHARGE_PLUS` represents the plus–sign for positive charges.

First the available regular expressions are given, then the available tokens will be given. With the regular expressions and the tokens the patterns and the resulting tokens are described.

Regular expressions

The regular expressions that are available to match the tokens in the input are defined in table 3.1. The elements of the periodic table are divided into nine sub–expressions (`SYM_1` ... `9`). These nine expressions are merged, and together with the aromatic symbols they form the expression `SYMBOL`.

```
DIGIT        [0-9]
NUMBER       {DIGIT}+

SYM_1        ("H"|"He")
SYM_2        ("Li"|"Be"|"B"|"C"|"N"|"O"|"F"|"Ne")
SYM_3        ("Na"|"Mg"|"Al"|"Si"|"P"|"S"|"Cl"|"Ar")
SYM_4        ("K"|"Ca"|"Sc"|"Ti"|"V"|"Cr"|"Mn"|"Fe"|"Co"|"Ni"|"Cu"|"Zn"|"Ga"|"Ge"|"As"|"Se"|"Br"|"Kr")
SYM_5        ("Rb"|"Sr"|"Y"|"Zr"|"Nb"|"Mo"|"Tc"|"Ru"|"Rh"|"Pd"|"Ag"|"Cd"|"In"|"Sn"|"Sb"|"Te"|"I"|"Xe")
SYM_6        ("Cs"|"Ba"|"Hf"|"Ta"|"W"|"Re"|"Os"|"Ir"|"Pt"|"Au"|"Hg"|"Tl"|"Pb"|"Bi"|"Po"|"At"|"Rn")
SYM_7        ("Fr"|"Ra"|"Rf"|"Db"|"Sg"|"Bh"|"Hs"|"Mt"|"Ds"|"Rg")
SYM_8        ("La"|"Ce"|"Pr"|"Nd"|"Pm"|"Sm"|"Eu"|"Gd"|"Tb"|"Dy"|"Ho"|"Er"|"Tm"|"Yb"|"Lu")
SYM_9        ("Ac"|"Th"|"Pa"|"U"|"Np"|"Pu"|"Am"|"Cm"|"Bk"|"Cf"|"Es"|"Fm"|"Md"|"No"|"Lr")
AROMATIC     ("c"|"n"|"o"|"p"|"s"|"se"|"as")
SYMBOL       ({SYM_1}|{SYM_2}|{SYM_3}|{SYM_4}|{SYM_5}|{SYM_6}|{SYM_7}|{SYM_8}|{SYM_9}|{AROMATIC})
CHREGEX      ("TH1"|"TH2"|"AL1"|"AL2"|"SP1"|"SP2"|"SP3")
BOND         ("="|"#"|"$"|":"|"/"|"\\")
DEPR_MIN     "--"
DEPR_PLUS    "++"
PRING        "%"
RINGBOND     ({BOND}{PRING}{DIGIT}{DIGIT}|{PRING}{DIGIT}{DIGIT})
```

Table 3.1: The regular expressions in the tokenizer

Tokens

The tokens that are available in the parser and the tokenizer are given in Figure 3.1. The tokens are grouped by type, and the description of the type is mentioned behind each group.

1. ELEMENT, AROMATIC (*symbols*)
2. DEPR_MIN, DEPR_PLUS, CHARGE_MINUS, CHARGE_PLUS (*charges*)
3. DIGIT, NUMBER (*numeric values*)
4. CHIRAL, CHIRAL_CODE (*chiralty definitions*)
5. BOND_1, BOND_2, BOND_3, BOND_4, BOND_ARO, RINGBOND (*bonds*)
6. CLASS_COLON (`:`)
7. DOT (`.`)
8. LPAREN, RPAREN (`(`, `)`)
9. LBRACKET, RBRACKET (`[`, `]`)
10. WILDCARD (`*`)

Figure 3.1: Available tokens in the parser and tokenizer

Patterns

The patterns that are formed using the available tokens and regular expressions are shown in table 3.2.

RegExp	Returns
{SYMBOL}	ELEMENT
{DIGIT}	DIGIT
{CHREGEX}	CHIRAL_CODE
"-"	in brackets? CHARGE_MINUS else BOND_1
"+"	CHARGE_PLUS
"="	BOND_2
"#"	BOND_3
"$"	BOND_4
":"	in brackets? CLASS_COLON else BOND_ARO
"/"	BOND_1
"\\"	BOND_1
"."	DOT
"("	LPAREN
")"	RPAREN
"*"	WILDCARD
{RINGBOND}	RINGBOND
{DEPR_MIN}	DEPR_MIN
{DEPR_PLUS}	DEPR_PLUS
"@"	CHIRAL
"\["	LBRACKET
"\]"	RBRACKET
.	any character not matched by another pattern

Table 3.2: Patterns and resulting tokens

3.1.2 Grammar

The grammar for the SMILES–parser is defined as follows:

```
smiles                ::= chain

chain                 ::= branched_atom
                        | branched_atom chain
                        | branched_atom bond chain
                        | branched_atom DOT chain

bond                  ::= BOND_1
                        | BOND_2
                        | BOND_3
                        | BOND_4
                        | BOND_ARO

branched_atom         ::= atom kleene_ringbond kleene_branch kleene_ringbond

kleene_ringbond       ::= kleene_ringbond ringbond
                        | ringbond
                        | ""

kleene_branch         ::= kleene_branch branch
                        | branch
                        | ""
```

```
ringbond              ::= RINGBOND
                        | DIGIT

branch                ::= LPAREN chain RPAREN
                        | LPAREN bond chain RPAREN
                        | LPAREN DOT chain RPAREN

atom                  ::= bracket_atom
                        | aliphatic_organic
                        | aromatic_organic
                        | WILDCARD

bracket_atom          ::= LBRACKET isotope symbol chiral hcount charge class RBRACKET

aliphatic_organic     ::= ELEMENT

aromatic_organic      ::= AROMATIC

isotope               ::= number
                        | ""


symbol                ::= ELEMENT
                        | AROMATIC
                        | WILDCARD

chiral                 ::= CHIRAL
                         | CHIRAL CHIRAL
                         | CHIRAL CHIRAL_CODE
                         | ""


hcount                 ::= ELEMENT
                         | ELEMENT DIGIT
                         | ""

charge                 ::= CHARGE_MINUS
                         | CHARGE_MINUS DIGIT
                         | CHARGE_PLUS
                         | CHARGE_PLUS DIGIT
                         | DEPR_MIN
                         | DEPR_PLUS
                         | ""

class                  ::= CLASS_COLON NUMBER
                         | ""

number                 ::= number DIGIT
                         | DIGIT
```

In the case of the `hcount` production, there is a check if the `ELEMENT` token is equal to `H`.

3.2 Parser

Within the system, all grammar rules are defined in the Bison definition file (`smiles.y`, see Appendix A). All tokens are defined in the Flex definition file (`smiles.l`, see Appendix A).

Atoms and bonds need to be tracked by the parser to create a list of atoms and bonds. To achieve this, classes were created to hold all the information about atoms (*Atom*) and bonds (*Bond*). The *RingTable* stores the state of rings that are encountered while parsing.

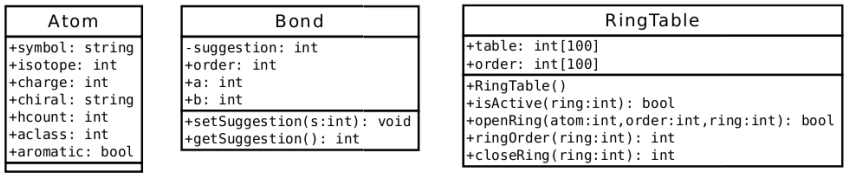


Figure 3.2: Data structures used by the parsing process

Every time an atom definition is encountered in the parsing process, a new instance of *Atom* is created. When a bond is encountered, a new instance of *Bond* is created. To handle (possibly multiple) branches and ring numbers, references to the branches and ring numbers are pushed onto a stack. The stack is processed when the corresponding atom is handled. The rings are handled by storing a reference to the atom where the ring number occurs the first time. When a ring number is encountered for the second time, a simple look–up in the *RingTable* gives the to atoms to bond together.

3.3 Post–processing

After parsing of the SMILES–string is finished a molecular graph — stored as a list of atoms and an adjacency list — is available. However, METool is not able to handle aromatic bonds and the graph still contains aromatic bonds and atoms.

Before the algorithm is explained in detail, the working is clarified using an example. In the example *2-Benzofuran* will be transformed step by step. In Figure 3.3 the results of each post–processing step are shown. In each structure a bond marked with a star means that the bond is present on the stack. If a bond has a thicker line, it is already processed by the algorithm.

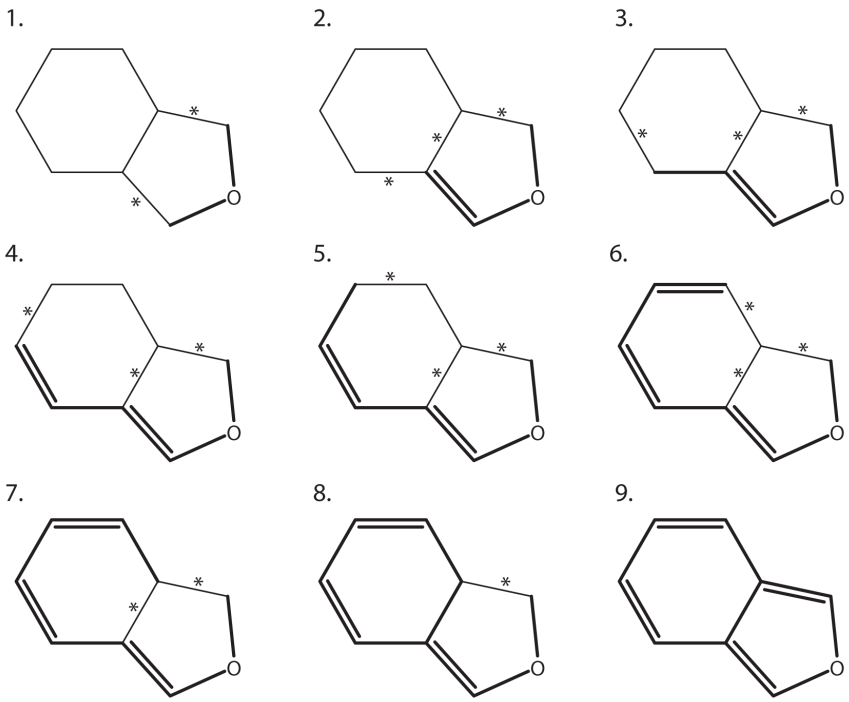


Figure 3.3: The post–processing of 2-Benzofuran

The steps taken by the algorithm are as follows:

Step 1 The preprocessing step detects an oxygen atom in a ring, and replaces the adjacent bonds with single bonds. Bonds that are adjacent to the processed bonds are pushed onto the stack.
Step 2 The order of the bond that is on top of the stack is set to 2 because the adjacent bonds that are already processed have an order of 1.
Steps 3 − 7 In these steps the order of the bond on top of the stack is set to a value so that an interleaving pattern of single and double bonds is obtained.
Step 8 The pattern suggests that the bond processed in this step should be a double bond. However, an atom at the bond of the bond already has a double bond, so it is not possible to set the current bond to a double bond.
Step 9 The last bond is set, and the post–processing algorithm is finished.

To transform all aromatic bonds in the graph a Depth First Search (DFS) algorithm (Algorithm 1) can be used. Aromatic bonds are pushed onto a stack, and the graph is traversed. When a bond is handled, the algorithm determines the new order (1 or 2) by determining if the neighbouring bonds prevent a double bond to be formed (Algorithm 6). This process continues until there are no aromatic bonds left in the graph.

The kekulization algorithm (Algorithm 1) converts molecular graphs with aromatic bonds to molecular graphs with explicitly defined bond orders. To accomplish this the algorithm starts with identifying all rings in the structure (Algorithm 3). Using a Depth First Search algorithm all atoms and bonds are visited. If an atom that is already visited is encountered, the algorithm determines that the path leading to that atom is a cycle. This step is needed because the SMILES–parser sees all bonds between two aromatic atoms as aromatic bonds, so a bond that connects two aromatic rings is also classified as an aromatic bond. To solve this problem the algorithm converts all the aromatic bonds that are not part of a ring.

Molecules can contain some *special cases* (see Figure 3.4). These are patterns that contain atoms with a maximum order of two in an aromatic ring or atoms that are connected to an aromatic ring via a double bond. The Kekulization–algorithm cannot handle these cases, so the Preprocess algorithm (Algorithm 2) finds these cases and converts them to structures containing normal bonds. This algorithm also pushes the encountered atoms on a stack as starting points for the Kekulization–algorithm.

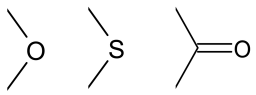


Figure 3.4: Special cases handled by the pre–processing step

The actual algorithm first checks if the *stack* of bonds to be processed is empty. If the stack is empty, the first aromatic bond in the list of bonds will be selected and pushed onto the stack. While the stack is not empty new orders will be calculated using the suggested value and the NewOrder algorithm (Algorithm 6). The suggested value is defined as not being equal to the maximum order

of adjacent bonds of the atom. So if an atom has an adjacent bond with order 2, the suggestion will be 1, and if there is no adjacent bond with order 2, the suggestion will be 2. In this way the interleaving of single and double bonds is accomplished. The new bond order is set and the adjacent bonds are pushed onto the stack together with a suggested bond order. If the stack is empty, the algorithm will try to add new aromatic bonds to the stack until there are no aromatic bonds left.

The worst case time complexity of the Kekulization algorithm can be estimated as $O(|A|+|B|)$, where $|A|$ is the number of atoms and $|B|$ the number of bonds. This is because the algorithm behaves like a Depth First Search algorithm for each graph component consisting of aromatic atoms. Because the total number of atoms and bonds in the aromatic graph components does not exceed the total number of atoms and bonds in the molecule graph, the estimated time complexity is a worst case estimate.

Algorithm 1: The Kekulization algorithm

```
Input: molecule graph containing aromatic bonds
Output: molecule graph in Kekulé format
Find all rings (FindRings)
Convert aromatic bonds that are not part of rings
Process special cases (Preprocess)

while aromatic bonds exist do
    if stack empty then
        find first aromatic bond and push it on the stack
    end
    while stack not empty do
        cur = top of stack
        pop stack

        if cur.order = aromatic then
            set cur.order to calculated order (NewOrder)
            if cur.order = 1 then
                suggestion = 2
            else
                suggestion = 1
            end
            push all adjacent bonds on the stack, together with suggestion
            (Bond, suggestion)
        end
    end
end
```

Algorithm 2: Preprocessing algorithm

```
Input: molecule graph
Output: molecule graph with converted special cases
for all atoms do
    if symbol = "o" then
        set adjacent bonds to order 1 and push on stack
    end
    if symbol = "s" then
        set adjacent bonds to order 1 and push on stack
    end
    if group is attached with double bond aromatic ring then
        set the adjacent bonds in the ring to order 1 and push on stack
    end
end
```

Algorithm 3 (FindRings) finds all rings in the molecular structure, to determine which atoms and bonds are in a ring. This is done using the recursive functions ProcessBond (Algorithm 4) and ProcessAtom (Algorithm 5).

ProcessBond and ProcessAtom are called while traversing the structure. Each function checks if the current bond or atom is visited before by the algorithm. If a visited bond or atom is found, the current path of function calls contains a loop. The function returns *true* and pushes the loop property up in the path of function calls. In this way the function can set ring property on the corresponding atoms and bonds.

Algorithm 3: FindRings

```
Input: molecule graph
while b = bond without cyclic definition do
    ProcessBond(b)
end
```

Algorithm 4: ProcessBond(b)

```
Input: bond b
Output: boolean
if b already visited then
    b is cyclic return true
end
res = false
if atom left of b already visited then
    res = processAtom(b.right, b)
else
    res = processAtom(b.left, b)
end
if res then
    b is cyclic return true
else
    b is not cyclic return false
end
```

Algorithm 5: ProcessAtom(a, p)

Input: atom a (current atom)
Input: atom p (previous bond)
Output: boolean
if *a already visited* **then**
  return $false$
end
$res = false$
for *each bond b adjacent to a and not equal to p* **do**
  $res = res \vee$ processBond(b)
end
if res **then**
  a is cyclic return $true$
else
  a is not cyclic return $false$
end

Algorithm 6 (NewOrder) determines if a bond becomes a single or double bond. If the suggested value is 2, and the atoms at both ends of the bond are able to form a double bond, the suggestion is returned. Otherwise the bond becomes a single bond, and 1 is returned.

Algorithm 6: NewOrder(b)

Input: bond b
Output: new order of the bond
if *GetBondBudget(b.left)* $\geq 1 \wedge$ *GetBondBudget(b.right)* ≥ 1 **then**
  return *b.suggestion*
else
  return 1
end

Algorithm 7 (GetBondBudget) computes the capacity to form new bonds an atom has. The capacity can be calculated by taking the maximum number of bonds a specific atom can have and subtracting the number of existing bonds, the defined charge of the atom and the number of explicitly defined hydrogen atoms.

Algorithm 7: GetBondBudget(a)

```
Input: atom a
Output: number of bonds that can be formed at the atom
if a = "c" then
|  budget = 4
end
if a = "n" ∨ a = "p" then
|  if neighbours(a) + a.charge ≥ 4 then
|  |  budget = 5
|  else
|  |  budget = 3
|  end
end
if a = "o" then
|  budget = 2
end
if a = "s" then
|  if neighbours(a) + a.charge ≥ 5 then
|  |  budget = 6
|  else if neighbours(a) + a.charge ≥ 3 then
|  |  budget = 4
|  else
|  |  budget = 2
|  end
end
return budget − neighbours(a) − a.charge − a.hydrogens
```

Chapter 4

Experiments

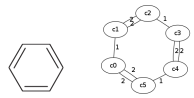
In this chapter it will be demonstrated that the parser works. The performance of the parser will also be demonstrated.

4.1 Test Cases

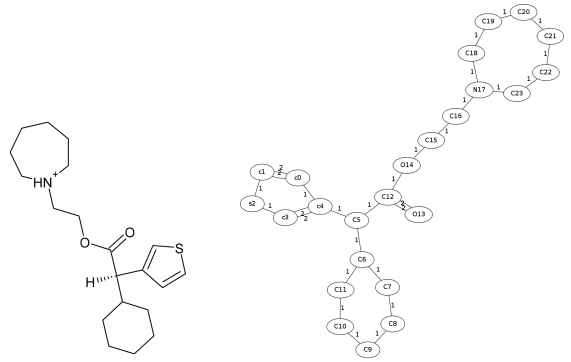
In order to test the correctness of the parser a debug output mode is used. In this debug output mode the lists of atoms and bonds are printed and a GraphViz [29] definition is generated. This definition can then be rendered by GraphViz to visualize the molecular graph.

This section presents ten of the test cases that were used to test the parser. The SMILES–string, the molecule rendered by ACD/ChemSketch [30] and the molecule–structure rendered by GraphViz [29] are shown.

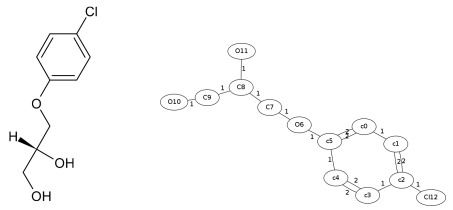
`c1ccccc1`



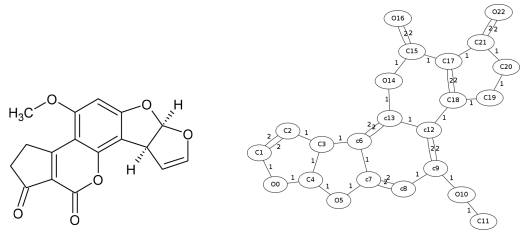
`c1cscc1[C@@H](C2CCCCC2)C(=O)OCC[NH+]3CCCCCC3`



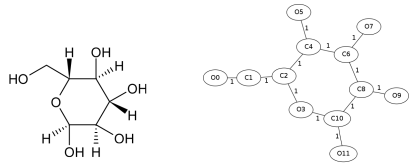
`c1cc(ccc1OC[C@@H](CO)O)Cl`



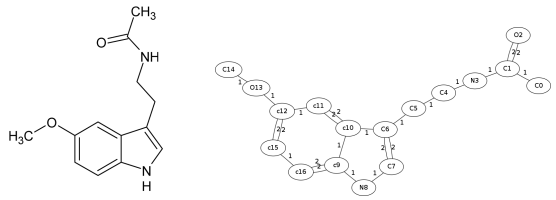
`O1C=C[C@H]([C@H]1O2)c3c2cc(OC)c4c3OC(=O)C5=C4CCC(=O)5`



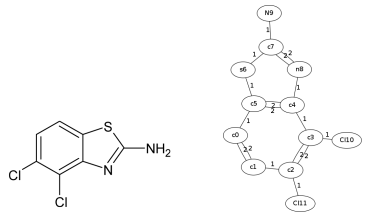
`OC[C@@H](O1)[C@@H](O)[C@H](O)[C@@H](O)[C@@H](O)1`



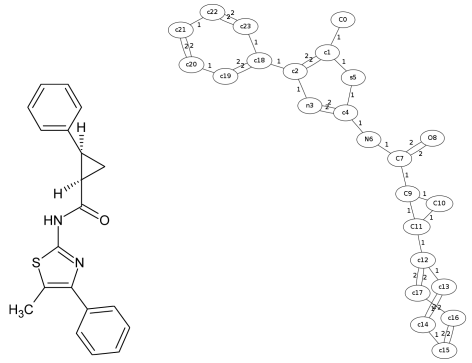
`CC(=O)NCCC1=CNc2c1cc(OC)cc2`



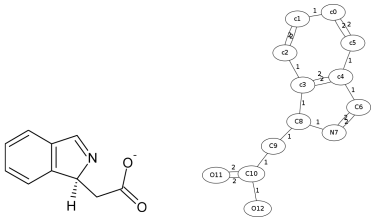
`c1cc(c(c2c1sc(n2)N)Cl)Cl`



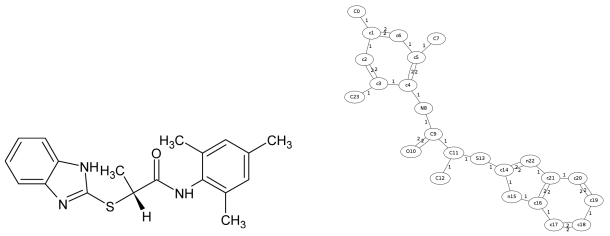
`Cc1c(nc(s1)NC(=O)[C@@H]2C[C@@H]2c3ccccc3)c4ccccc4`



`c1ccc2c(c1)C=N[C@H]2CC(=O)[O-]`



`Cc1cc(c(c(c1)C)NC(=O)[C@@H](C)Sc2[nH]c3ccccc3n2)C`



4.2 Performance

To test the performance of the parser, a dataset of 1000 SMILES–string from the ZINC–database [13] was parsed by the parser. The time needed to process each string is plotted against the length of the SMILES–string.

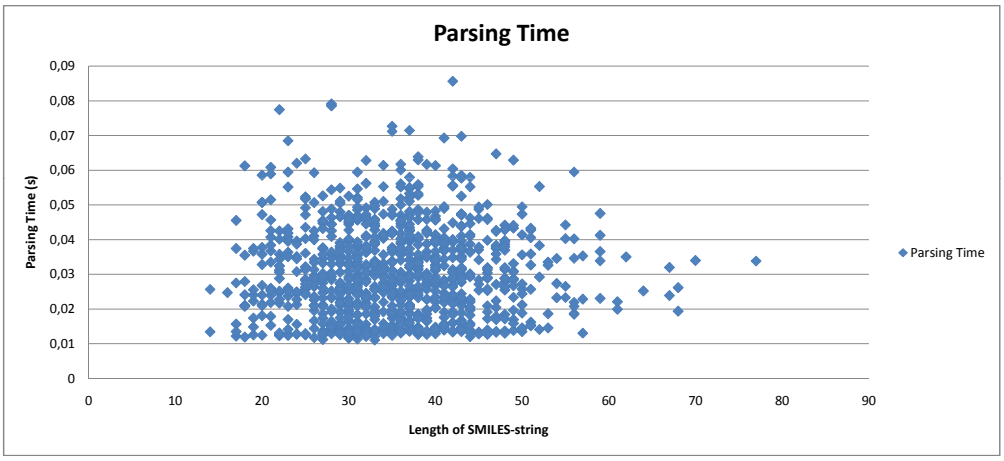


Figure 4.1: Graph of total time against string length

It is not possible to find a pattern in this graph, so it can be concluded that there is no strong correlation between the length of the SMILES–string and the parsing time. To see if this is caused by the parsing or the post–processing step, the graph is split in separate parts for the parsing (Figure 4.2) and the post–processing (Figure 4.3) steps.

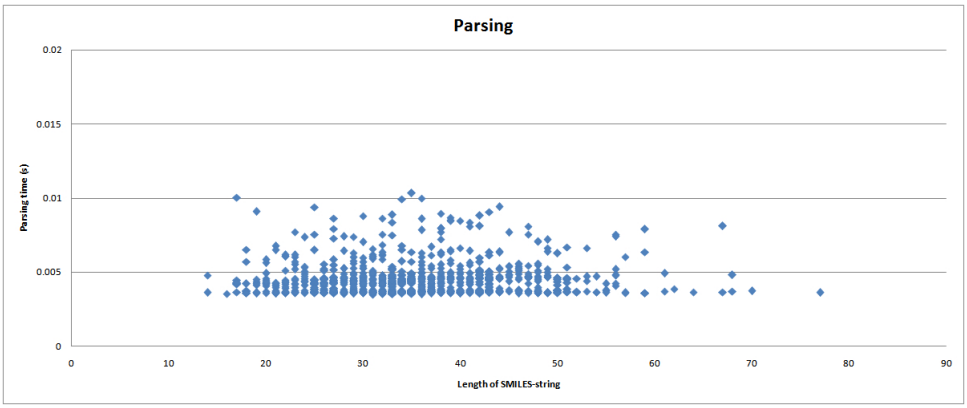


Figure 4.2: Graph of parsing time

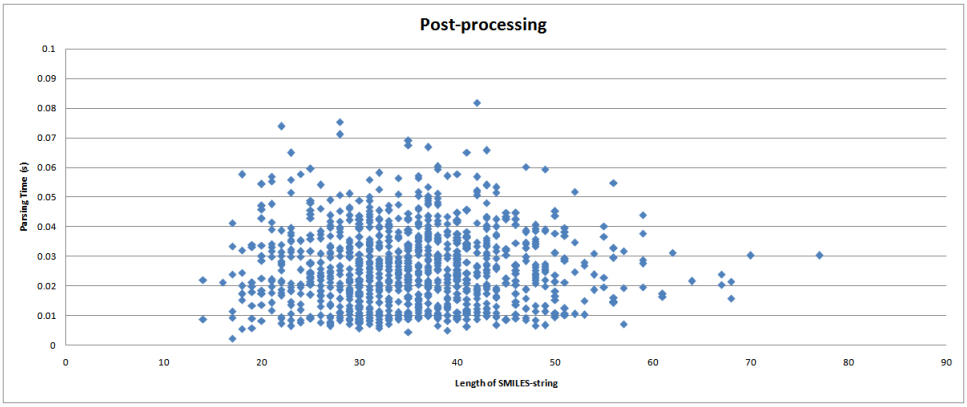


Figure 4.3: Graph of post–processing time

These graphs show that the parsing time takes only about 15% (0.004 seconds on average) of the total time. The rest of the time (85%, 0.026 seconds on average) is consumed by the post–processing algorithm.

Because the length of the test strings appears to be too small to notice an increase in parsing time, another experiment with chains of carbon atoms up to length 1000 is done (Figure 4.4).

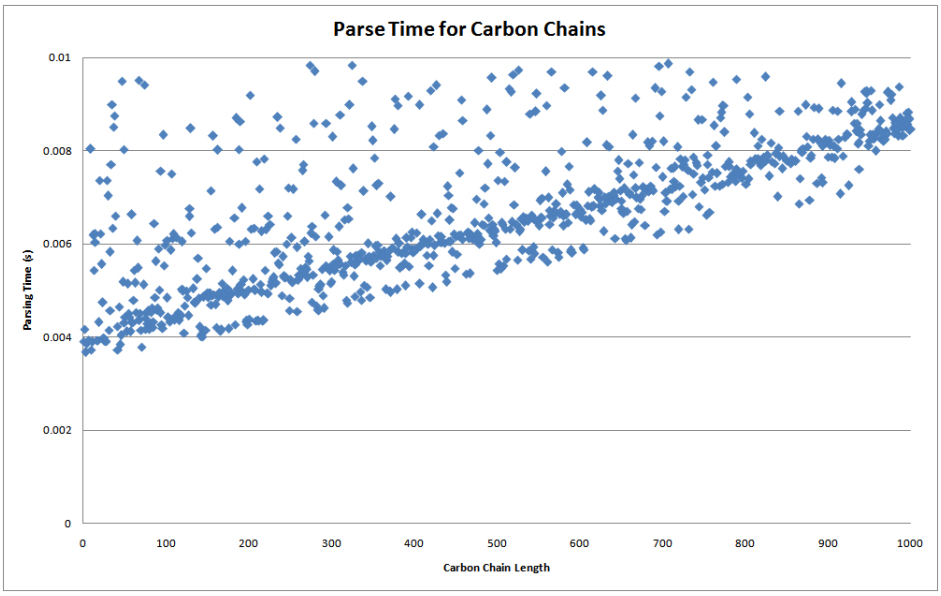


Figure 4.4: Graph of parsing times for carbon chains

This graph shows a clear increase in the parsing time for longer strings. The parsing times that deviate from the line, are most likely caused by the scheduler of the operating system and interrupts during the experiments.

Chapter 5

Conclusions and Future Work

In this paper it is demonstrated that it is possible to implement a SMILES–parser using Bison and Flex, that parses drug–like molecules. Although the SMILES–standard was not designed with the specific goal to make it suitable for these kinds of parsers, it is possible to translate the semantics of SMILES to semantic code in the parser. For example, it is not possible to parse rings in a natural way, but with some additional bookkeeping (*RingTable*) it is possible.

Using the OpenSMILES–grammar to create a SMILES–parser, proves to be a way to save a lot of time. OpenSMILES and SMILES are equivalent at many points, so only small changes have to be made. Something that takes more time is converting the obtained grammar to a form that is suitable for the Bison parser generator. The Backus–Naur Form facilitates a large number of short hand notations to provide a concise way to write grammars. Some of these short hand notations need to be converted in another format to conform to the Bison input specification.

Kekulization is important because the METool can not handle molecule structures that contain bonds that are specifically designated as aromatic bonds. The algorithm is tested with drug–like molecules, and works well. Because the algorithm finds a good configuration with alternating single and double bonds, possible faulty results can be solved by extending the preprocessing function.

Because the parser has the goal to parse molecule fragments for the Molecule Evolution Tool, the features of SMILES that are used by the tool are tested more thoroughly than the other features. Therefore it can not be guaranteed that the parser parses *all* molecules correctly. In this phase it is possible to integrate the parser in other tools that need a SMILES–parser as long as the tool only handles drug–like molecules.

In the future it is possible to test all features of SMILES and fix possible shortcomings of the parser, to create a SMILES parser that can be used for *general purpose* molecule parsing. It may be considered to make the parser available to the bio–informatics community.

Bibliography

[1] J.W. Kruisselbrink, A. Aleman, M. Emmerich, A.P. IJzerman, A. Bender, T. Baeck, and E. van der Horst. Enhancing search space diversity in multi-objective evolutionary drug molecule design using niching. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 217–224. ACM, 2009.

[2] J. Kruisselbrink, M. Emmerich, T. Baeck, A. Bender, A. IJzerman, and E. van der Horst. Combining Aggregation with Pareto Optimization: A Case Study in Evolutionary Molecular Design. In *Evolutionary Multi-Criterion Optimization*, pages 453–467. Springer, 2009.

[3] T. Bäck, D.B. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. Taylor & Francis, 1997.

[4] E.W. Lameijer, J.N. Kok, T. Bäck, and A.P. IJzerman. The molecule evoluator. an interactive evolutionary algorithm for the design of drug-like molecules. *J. Chem. Inf. Model*, 46(2):545–552, 2006.

[5] Pipeline pilot. retrieved from `http://accelrys.com/products/pipeline-pilot/` on 03–07–2010.

[6] D. Weininger. SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules. In *Proc. Edinburgh Math. SOC*, volume 17, pages 1–14, 1970.

[7] Chemistry development kit (cdk). retrieved from `http://sourceforge.net/apps/mediawiki/cdk/index.php` on 13–09–2010.

[8] Rdkit. retrieved from `http://www.rdkit.org/` on 13–09–2010.

[9] Daylight chemical information systems inc. retrieved from `http://www.daylight.com/` on 09–03–2010.

[10] Openeye. retrieved from `http://eyesopen.com/` on 13–09–2010.

[11] Flex — fast lexical analyzer generator. retrieved from `http://flex.sourceforge.net/` on 14–06–2010.

[12] Bison — gnu parser generator. retrieved from `http://www.gnu.org/software/bison/` on 09–03–2010.

[13] Zinc — a free database for virtual screening. retrieved from http://zinc.docking.org/ on 14–06–2010.

[14] Chemspider — database of chemical structures and property predictions. retrieved from http://www.chemspider.com/ on 14–06–2010.

[15] Chebi — chemical entities of biological interest. retrieved from http://www.ebi.ac.uk/chebi on 14–06–2010.

[16] Smiles — a simplified chemical language. retrieved from http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html on 09–03–2010.

[17] Opensmiles. retrieved from http://www.opensmiles.org/ on 09–03–2010.

[18] Opensmiles specification — 6. proposed extensions. retrieved from http://www.opensmiles.org/spec/open-smiles-6-extensions.html on 14–06–2010.

[19] Ctfile formats. retrieved from http://www.symyx.com/downloads/public/ctfile/ctfile.jsp on 06–07–2010.

[20] Atomic coordinate entry format description. retrieved from http://www.wwpdb.org/documentation/format32/v3.2.html on 06–07–2010.

[21] Chemical markup language. retrieved from http://sourceforge.net/projects/cml/ on 06–07–2010.

[22] Smarts – a language for describing molecular patterns. retrieved from http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html on 06–07–2010.

[23] S. Ash, M.A. Cline, R.W. Homer, T. Hurst, and G.B. Smith. SYBYL Line Notation (SLN): A Versatile Language for Chemical Structure Representation. *J. Chem. Inf. Comput. Sci*, 37(1):71–79, 1997.

[24] W.J. Wiswesser. How the WLN began in 1949 and how it might be in 1999. *Journal of Chemical Information and Computer Sciences*, 22(2):88–93, 1982.

[25] The iupac international chemical identifier. retrieved from http://www.iupac.org/inchi/ on 06–07–2010.

[26] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

[27] Algol–60 bnf. retrieved from http://www.lrz.de/ bernhard/Algol-BNF.html on 08–07–2010.

[28] Opensmiles specification — 2. grammar. retrieved from http://www.opensmiles.org/spec/open-smiles-2-grammar.html on 16–06–2010.

[29] Graphviz. retrieved from http://www.graphviz.org/ on 15–06–2010.

[30] Chemsketch. retrieved from http://www.acdlabs.com/products/draw_nom/draw/chemsketch/ on 08–07–2010.

Appendix A

Source Code

smiles.l

```
%{
#include <stdio.h>
#include <string>
#include <list>
#include "atom.h"
#include "bond.h"
#include "ring.h"
#include "y.tab.h"


std::string lastString;
std::string prevString;
int lastBond;
int lastNumber;

bool inBrackets = false;

std::string smiles_input;
int smiles_pos = 0;

#define YY_INPUT(buf,result,max_size) \
  { \
  result = (smiles_pos < smiles_input.length()) ? (buf[0] =
    smiles_input[smiles_pos++], 1) : YY_NULL; \
  }
%}

DIGIT [0-9]
NUMBER {DIGIT}+

SYM_1 ("H"|"He")
SYM_2 ("Li"|"Be"|"B"|"C"|"N"|"O"|"F"|"Ne")
SYM_3 ("Na"|"Mg"|"Al"|"Si"|"P"|"S"|"Cl"|"Ar")
SYM_4 ("K"|"Ca"|"Sc"|"Ti"|"V"|"Cr"|"Mn"|"Fe"|"Co"|"Ni"|"Cu"|"Zn"|
    "Ga"|"Ge"|"As"|"Se"|"Br"|"Kr")
SYM_5 ("Rb"|"Sr"|"Y"|"Zr"|"Nb"|"Mo"|"Tc"|"Ru"|"Rh"|"Pd"|"Ag"|
    "Cd"|"In"|"Sn"|"Sb"|"Te"|"I"|"Xe")
SYM_6 ("Cs"|"Ba"|"Hf"|"Ta"|"W"|"Re"|"Os"|"Ir"|"Pt"|"Au"|"Hg"|
    "Tl"|"Pb"|"Bi"|"Po"|"At"|"Rn")
SYM_7 ("Fr"|"Ra"|"Rf"|"Db"|"Sg"|"Bh"|"Hs"|"Mt"|"Ds"|"Rg")
```

```
SYM_8  ("La"|"Ce"|"Pr"|"Nd"|"Pm"|"Sm"|"Eu"|"Gd"|"Tb"|"Dy"|"Ho"|
      "Er"|"Tm"|"Yb"|"Lu")
SYM_9  ("Ac"|"Th"|"Pa"|"U"|"Np"|"Pu"|"Am"|"Cm"|"Bk"|"Cf"|"Es"|
      "Fm"|"Md"|"No"|"Lr")
AROMATIC   ("c"|"n"|"o"|"p"|"s"|"se"|"as")

SYMBOL   ({SYM_1}|{SYM_2}|{SYM_3}|{SYM_4}|{SYM_5}|{SYM_6}|{SYM_7}|
      {SYM_8}|{SYM_9}|{AROMATIC})
CHREGEX  ("TH1"|"TH2"|"AL1"|"AL2"|"SP1"|"SP2"|"SP3")

BOND   ("="|"#"|"$"|":"|"/"|"\\")

DEPR_MIN   "--"
DEPR_PLUS  "++"
PRING      "%"
RINGBOND   ({BOND}{PRING}{DIGIT}{DIGIT}|{PRING}{DIGIT}{DIGIT})


%%

{SYMBOL}      { prevString = lastString;
         lastString = (std::string) yytext;
         return ELEMENT; }
{DIGIT}       { lastNumber = atoi(yytext); return DIGIT; }
{CHREGEX}     { lastString = (std::string) yytext;
         return CHIRAL_CODE; }
"-"      { if (inBrackets) { return CHARGE_MINUS; }
         else { return BOND_1; }}
"+"      { return CHARGE_PLUS; }
"="      { return BOND_2; }
"#"      { return BOND_3; }
"$"      { return BOND_4; }
":"      { if (inBrackets) { return CLASS_COLON; }
         else { return BOND_ARO; } }
"/"      { return BOND_1; }
"\\"       { return BOND_1; }
"."      { return DOT; }
"("      { return LPAREN; }
")"      { return RPAREN; }
"*"      { return WILDCARD; }
{RINGBOND}      { yytext[0] = '0';
         lastBond = atoi(yytext);
         return RINGBOND; }
{DEPR_MIN}      { return DEPR_MIN; }
{DEPR_PLUS}     { return DEPR_PLUS; }

"@"      { if (inBrackets) { return CHIRAL; }
         else { return yytext[0]; } }
"\["       { inBrackets = true; return LBRACKET; }
"\]"       { inBrackets = false; return RBRACKET; }
.      { return yytext[0]; }
%%
```

smiles.y

```
%{
#include <stdio.h>
#include <string>
#include <cstring>
#include <map>
#include <vector>
#include <list>
```

```
#include <stack>
#include <set>
#include <iostream>

#include "atom.h"
#include "bond.h"
#include "ring.h"

#include "kekulize.h"

#define VERBOSITY 1

int debug = 1;

/* Prototypes */
static void yyerror(const char *);
static void yywarning(const char *);

/* Import from smiles.l */
extern int yylex(void);
extern std::string lastString;
extern std::string prevString;
extern int lastBond;
extern int lastNumber;
extern std::string smiles_input;

RingTable rt;

std::stack<int> atomStack;

int atomCount = 0;

std::map<int,Atom> atomList;
std::map<int,Atom>::iterator it;

std::vector<Bond> bondList;
std::vector<Bond>::iterator bit;

bool step_done = false;



bool aromatic_symbols(std::string s) {
  if (s.compare("c") == 0) return true;
  if (s.compare("n") == 0) return true;
  if (s.compare("o") == 0) return true;
  if (s.compare("p") == 0) return true;
  if (s.compare("s") == 0) return true;
  if (s.compare("se") == 0) return true;
  if (s.compare("as") == 0) return true;
  return false;
}

%}

%start smiles

%token ELEMENT AROMATIC
%token DEPR_MIN DEPR_PLUS
%token DIGIT NUMBER
%token SPACE TAB LINEFEED CARRIAGE_RETURN END_OF_STRING
%token CHIRAL CHIRAL_CODE
```

```
%token BOND_1 BOND_2 BOND_3 BOND_4 BOND_ARO RINGBOND
%token CHARGE_MINUS CLASS_COLON CHARGE_PLUS
%token DOT LPAREN RPAREN WILDCARD LBRACKET RBRACKET

%union {
  std::list<Bond> *groups;
  std::list<int> *rings;
  Bond *bond;
  int i;
  std::string *str;
}

%%

smiles:
    chain { return true; }
  ;


chain:
    branched_atom { $<i>$ = $<i>1; }
  | branched_atom chain {
    Bond b;
    b.order = atomList[$<i>2].aromatic &&
        atomList[$<i>1].aromatic ? -1 : 1;
    b.a = $<i>2;
    b.b = $<i>1;
    bondList.push_back(b);
    $<i>$ = $<i>1;
    }
  | branched_atom bond chain {
    Bond b;
    b.order = atomList[$<i>3].aromatic &&
    atomList[$<i>1].aromatic ? -1 : $<i>2;
    b.a = $<i>3;
    b.b = $<i>1;
    bondList.push_back(b);
    $<i>$ = $<i>1;
          }
  | branched_atom DOT chain { $<i>$ = $<i>1; }
  ;

bond:
    BOND_1 { rule("bond ::= '-'"); $<i>$ = 1;}
  | BOND_2 { rule("bond ::= '='"); $<i>$ = 2;}
  | BOND_3 { rule("bond ::= '#'"); $<i>$ = 3;}
  | BOND_4 { rule("bond ::= '$'"); $<i>$ = 4;}
  | BOND_ARO { rule("bond ::= ':'"); $<i>$ = -1;}
  ;

branched_atom:
   atom kleene_ringbond kleene_branch kleene_ringbond {

    std::list<Bond>::iterator git;
    if ($<groups>3 != NULL && !$<groups>3->empty()) {
        std::list<Bond> gr = *$<groups>3;
          for ( git = gr.begin(); git != gr.end(); git++) {
            if ((*git).order != -1) {
              (*git).a = $<i>1;
            } else {
              (*git).a = $<i>1;
              (*git).order = aromatic_symbols(
```

```
            atomList[$<i>1].symbol) ? -1 : 1;
              }
              bondList.push_back(*git);
        }
           }

    std::list<int>::iterator rit;
    if ($<rings>2 != NULL && !$<rings>2->empty()) {
        std::list<int> ring = *$<rings>2;
        for ( rit = ring.begin();
         rit != ring.end(); rit++) {
            if (!rt.isActive(*rit)) {
              rt.openRing($<i>1,
         atomList[$<i>1].aromatic ?
         -1 : 1, *rit);
        } else {
              Bond temp;
              temp.order = rt.ringOrder(*rit) == -1
               && atomList[$<i>1].aromatic ? -1 : 1;
              temp.a = rt.closeRing(*rit);
              temp.b = $<i>1;
              bondList.push_back(temp);
            }
        }
          }

    if ($<rings>4 != NULL && !$<rings>4->empty()) {
        std::list<int> ring = *$<rings>4;
        for ( rit = ring.begin();
        rit != ring.end(); rit++) {

            if (!rt.isActive(*rit)) {
              rt.openRing($<i>1,
         atomList[$<i>1].aromatic ? -1 : 1, *rit);
            } else {
              Bond temp;
              temp.order = rt.ringOrder(*rit) == -1
        && atomList[$<i>1].aromatic ? -1 : 1;
        temp.a = rt.closeRing(*rit);
              temp.b = $<i>1;
              bondList.push_back(temp);
            }
        }
    }

    $<i>$ = $<i>1;
  }
  ;

kleene_ringbond:
    /* empty */ { $<rings>$ = NULL; }
        | kleene_ringbond ringbond {
    std::list<int> *r = $<rings>1;
    r->push_back($<i>2);
    $<rings>$ = r;
    }

       | ringbond {
    std::list<int> *r = new std::list<int>;
    r->push_back($<i>1);
    $<rings>$ = r;
    }
```

```
  ;

kleene_branch:
    /* empty */ { $<groups>$ = NULL; }
   | kleene_branch branch {
    std::list<Bond> *t = $<groups>1;
    if ($<bond>2->order != 0) t->push_back(*$<bond>2);
    $<groups>$ = t;
    }
  | branch {
    std::list<Bond> *t = new std::list<Bond>;
    if ($<bond>1->order != 0) t->push_back(*$<bond>1);
    $<groups>$ = t;
    }
  ;

ringbond:
    RINGBOND { $<i>$ = lastBond; }
  | DIGIT { $<i>$ = lastNumber; }
  ;

branch:
    LPAREN chain RPAREN {
    $<i>$ = $<i>2;
    Bond *b = new Bond();
    b->order = atomList[$<i>2].aromatic ? -1 : 1;
    b->b = $<i>2;
    $<bond>$ = b;
    }
  | LPAREN bond chain RPAREN {
    Bond *b = new Bond();
    b->order = $<i>2;
    b->b = $<i>3;
    $<bond>$ = b;

  }
  | LPAREN DOT chain RPAREN {
    Bond *b = new Bond();
    b->order = 0;
    b->b = $<i>3;
    $<bond>$ = b;
  }
  ;

atom:
    bracket_atom { $<i>$ = $<i>1; }
  | aliphatic_organic { $<i>$ = $<i>1; }

  | aromatic_organic { $<i>$ = $<i>1; }
  | WILDCARD { $<i>$ = $<i>1; }
  ;

bracket_atom:
  LBRACKET isotope symbol chiral hcount charge class RBRACKET {
        Atom a;
        a.symbol = $<i>5 > 1 ? prevString : *$<str>3;
        a.aromatic = aromatic_organic(a.symbol);
        a.isotope = $<i>2;
        a.hcount = $<i>5;
        a.charge = $<i>6;
        a.aclass = $<i>7;
        atomList[atomCount] = a;
```

```
        $<i>$ = atomCount;
        atomCount++;
      }
  ;

aliphatic_organic:
   ELEMENT {
      Atom a; a.symbol = lastString.c_str();
      a.aromatic = aromatic_organic(lastString);

      atomList[atomCount] = a;
      $<i>$ = atomCount;
      atomCount++;
       }
  ;

aromatic_organic:
  AROMATIC {
    if (aromatic_organic(lastString)) {
      Atom a; a.symbol = lastString.c_str();
      a.aromatic = true;
      atomList[atomCount] = a;
      $<i>$ = atomCount;
      atomCount++;
    } else {
      yyerror("element not aromatic organic");
    }

  }
  ;

isotope:
    /* empty */ { $<i>$ = 0; }
  | number { $<i>$ = $<i>1; }
  ;

symbol:
    ELEMENT { std::string t = lastString; $<str>$ = &t; }
  | AROMATIC { std::string t = lastString; $<str>$ = &t; }
  | WILDCARD { $<str>$ = &lastString; }
  ;

chiral:
    /* empty */   { }
  | CHIRAL     { }
  | CHIRAL CHIRAL    { }
  | CHIRAL CHIRAL_CODE   { }
  ;

hcount:
     /* empty */ {$<i>$ = 0; }
  | ELEMENT { $<i>$ = 1;}
  | ELEMENT DIGIT { $<i>$ = lastNumber; }
  ;

charge:
    /* empty */ { $<i>$ = 0; }
  | CHARGE_MINUS { $<i>$ = -1; }
  | CHARGE_MINUS DIGIT { $<i>$ = -lastNumber; }
  | CHARGE_PLUS { $<i>$ = 1; }
  | CHARGE_PLUS DIGIT { $<i>$ = lastNumber; }
  | DEPR_MIN { $<i>$ = -2; }
```

```
  | DEPR_PLUS { $<i>$ = 2; }
  ;

class:
    /* empty */ { $<i>$ = 0; }
  | CLASS_COLON number { $<i>$ = $<i>2; }
  ;

number:
    number DIGIT { $<i>$ = $<i>1 * 10 + lastNumber; }
  | DIGIT { $<i>$ = lastNumber; }
  ;

%%


int main(int argc, char *argv[])
{
  ...
}
static void yyerror(const char *s)
{
  fprintf(stderr, "error: %s\n", s);
}

static void yywarning(const char *s)
{
  fprintf(stderr, "warning: %s\n", s);
}

int yywrap() { return(1); }
```

Atom.h

```
#ifndef ATOM_H
#define ATOM_H

class Atom {

    public:
        std::string symbol;
        int atomNumber;
        int isotope;
        int charge;
        std::string chiral;
        int hcount;
        int aclass;
        bool aromatic;

        bool processed;
        bool cyclic;
        bool visited;
        int ring;

        Atom()
        {
            symbol = "";
            isotope = 0;
            charge = 0;
            chiral = "";
            hcount = 0;
            aclass = 0;
```

```
            aromatic = false;

            processed = false;
            cyclic = false;
            visited = false;
            ring = -1;
        }

};

#endif
```

Bond.h

```
#ifndef BOND_H
#define BOND_H

class Bond
{

    public:
        int order;
        int a, b;

        bool processed;
        bool cyclic;
        bool visited;
        bool initial;
        int ring;

        Bond()
        {
            processed = false;
            cyclic = false;
            visited = false;
            initial = false;
            ring = -1;
        }

        void setSuggestion(int s)
        {
            if (s <= suggestion || suggestion == 0) suggestion = s;
        }

        int getSuggestion() { return suggestion; }

    private:
        int suggestion;

};

#endif
```

RingTable.h

```
#ifndef RING_H
#define RING_H

static const int MaxRingTableSize = 100;

class RingTable
{
```

```
    public:
        RingTable();

        bool isActive(int r);
        bool openRing(int atom, int ord, int r);
        int ringOrder(int r);
        int closeRing(int r);

    private:
        int table[MaxRingTableSize];
        int order[MaxRingTableSize];

};

#endif
```

RingTable.cpp

```
#include "RingTable.h"

/*
 ==========================================================
    'RingTable'
 ==========================================================
 */
RingTable::RingTable()
{
    for (int i = 0; i < 100; i++)
    {
        table[i] = -1;
        order[i] = 0;
    }
}


/*
 ==========================================================
    'isActive'
 ==========================================================
 */
bool RingTable::isActive(int r)
{
    return table[r] > -1;
}


/*
 ==========================================================
    'openRing'
 ==========================================================
 */
bool RingTable::openRing(int atom, int ord, int r)
{
    if (!isActive(r))
    {
        table[r] = atom;
        order[r] = ord;
        return true;
    }
    return false;
}
```

```
/*
 ==========================================================
     'ringOrder'
 ==========================================================
 */
int RingTable::ringOrder(int r)
{
    return order[r];
}


/*
 ==========================================================
     'closeRing'
 ==========================================================
 */
int RingTable::closeRing(int r)
{
    int ret = table[r];
    table[r] = -1;
    return ret;
}
```

Kekulize.h

```
#ifndef KEKULIZE_H
#define KEKULIZE_H

#include <map>
#include <vector>
#include <stack>

#include "../Atom.h"
#include "../Bond.h"

class Kekulize
{
    std::map<int,Atom> *atomList;
    std::vector<Bond> *bondList;

    std::stack<Bond*> kStack;
    int curRing;

public:
    Kekulize(std::map<int,Atom> &nAtomList,
             std::vector<Bond> &nBondList);
    void pushBond(Bond *b);
    void process();
    void preprocess();
    void setAdjacentBonds(int atomId, int ord, bool push);
    bool isAdjacentBond(Bond x, Bond y);
    bool stackNotEmpty();
    int newOrder(Bond *cur);
    int getNeighbourCount(int atomId);
    int getBondBudget(int atomId);
    bool aromaticBondsExist();
    bool isAromatic(Atom a);
    void aromaticToNormal();

    void findRings();
```

```
    void resetVisited();
    bool procAtom(int atom, int prevBond);
    bool procBond(int bond);
    int findCyclicUndefined();
};

#endif
```

Kekulize.cpp

```
#include <iostream>

#include "Kekulize.h"

/*
 ==========================================================
    'Kekulize' Constructor, stores references to the atom
    and bond lists
 ==========================================================
 */
Kekulize::Kekulize(std::map<int,Atom> &nAtomList,
                   std::vector<Bond> &nBondList)
{
    atomList = &nAtomList;
    bondList = &nBondList;

    curRing = 0;
}

/*
 ==========================================================
    'pushBond' Pushes a given bond b onto the stack
 ==========================================================
 */
void Kekulize::pushBond(Bond *b)
{
     kStack.push(b);
}

/*
 ==========================================================
    'process' Processes the stack
 ==========================================================
 */
void Kekulize::process()
{
     //determine for all atoms and bonds if they are in a ring
     findRings();

     //if the atoms at the end of a bond are in different rings
     //according to the findRings() algorithm, that bond cannot
     //be aromatic, so it is set to order 1, else the bond is
     //in a cycle.
     std::vector<Bond>::iterator bit;
     for (bit = bondList->begin(); bit != bondList->end(); bit++)
     {
         if ((*bit).order == -1 && (*bit).ring != -1)
         {
             if ((*atomList)[(*bit).a].ring !=
                 (*atomList)[(*bit).b].ring)
             {
                 (*bit).order = 1;
```

```
            }
            else
            {
                (*bit).cyclic = true;
            }
        }
    }

    //replace tricky situations with the solution
    preprocess();

    while (aromaticBondsExist())
    {
        //if the stack is empty place the first bond of order
        //-1 in the bondlist on the stack
        if (!stackNotEmpty())
        {
            std::vector<Bond>::iterator iit;
            int i = 0;
            for (iit = bondList->begin(); iit != bondList->end();
                 iit++)
            {
                if ((*iit).order == -1)
                {
                    (*iit).setSuggestion(2);
                    pushBond(&(*iit));
                    break;
                }
                i++;
            }
        }

        while (stackNotEmpty())
        {
            Bond *cur = kStack.top();
            kStack.pop();

            if (cur->order == -1)
            {
                //newOrder computes the possible order of the bond
              cur->order = newOrder(cur);

              //set the suggestion for the next bonds
                int suggest = cur->order == 2 ? 1 : 2;

                //push all adjacent bonds on the stack with the
                //suggestion
              std::vector<Bond>::iterator it;
                for (it = bondList->begin(); it != bondList->end();
                     it++)
                {
                  if (isAdjacentBond((*it), *cur))
                    {
                        if ((*it).order == -1)
                        {
                            (*it).setSuggestion(suggest);
                            pushBond(&(*it));
                        }
                    }
                }
             }
        }
```

```
    }
}

/*
 ==========================================================
    'preprocess' Replace situations where there is only one
    possibility with the solution, and place the borders of
    the parts on the stack
 ==========================================================
 */
void Kekulize::preprocess()
{
    std::map<int,Atom>::iterator it;
    for (it = atomList->begin(); it != atomList->end(); it++)
    {
        //if an aromatic oxygen atom appears, it has two single
        //bonds
        if ((*it).second.symbol.compare("o") == 0)
        {
            setAdjacentBonds((*it).first, 1, true);
        }

        //if an aromatic sulphur atom appears and it has two
        //neighbours it has two single bonds
        if ((*it).second.symbol.compare("s") == 0)
        {
            if (getNeighbourCount((*it).first) == 2)
            {
              setAdjacentBonds((*it).first, 1, true);
            }
        }

        if (isAromatic((*it).second))
        {
            std::vector<Bond>::iterator bit;
            for (bit = bondList->begin(); bit != bondList->end();
                 bit++)
            {
              if ((*bit).order == 2 &&
                  ((*bit).a == (*it).first ||
                   (*bit).b == (*it).first))
                {
                  setAdjacentBonds((*it).first, 1, true);
                }
             }
        }
    }
}

/*
 ==========================================================
    'setAdjacentBonds' Sets the bonds adjacent to atom with
    id = atomId to order ord. If push is true then the
    affected bonds will be pushed on the stack.
 ==========================================================
 */
void Kekulize::setAdjacentBonds(int atomId, int ord, bool push)
{
    std::vector<Bond>::iterator bit;
    for (bit = bondList->begin(); bit != bondList->end(); bit++)
    {
        if ((*bit).order == -1 && ((*bit).a == atomId ||
```

```
            (*bit).b == atomId))
        {
            (*bit).order = ord;

            if (push)
            {
              std::vector<Bond>::iterator pbit;
                for (pbit = bondList->begin();
                     pbit != bondList->end(); pbit++)
                {
                    if (isAdjacentBond(*pbit, *bit) &&
                        (*pbit).order == -1)
                    {
                      (*pbit).setSuggestion(2);
                        pushBond(&(*pbit));
                    }
                }
            }
        }
    }
}

/*
 ==========================================================
    'isAdjacentBond' Returns true if two bonds share a
    common connected atom
 ==========================================================
 */
bool Kekulize::isAdjacentBond(Bond x, Bond y)
{
    return x.a == y.a || x.a == y.b || x.b == y.a || x.b == y.b;
}

/*
 ==========================================================
    'stackNotEmpty' Checks if the stack is not empty
 ==========================================================
 */
bool Kekulize::stackNotEmpty()
{
    return !kStack.empty();
}

/*
 ==========================================================
    'newOrder' Computes the possible order of an aromatic
    bond. If both atoms can handle another bond, return
    the minimum of the suggested value and 2. Otherwise
    return 1.
 ==========================================================
 */
int Kekulize::newOrder(Bond *cur)
{
    int max;
    int aBudget = getBondBudget(cur->a);
    int bBudget = getBondBudget(cur->b);

    if (aBudget >= 1 && bBudget >= 1)
    {
        return cur->getSuggestion() < 2 ? cur->getSuggestion() : 2;
    }
    else
```

```
    {
        return 1;
    }
}

/*
 ==========================================================
    'getNeighbourCount' Get the number of neighbouring
    atoms of an atom
 ==========================================================
 */
int Kekulize::getNeighbourCount(int atomId)
{
    int count = 0;
    std::vector<Bond>::iterator it;
    for (it = bondList->begin(); it != bondList->end(); it++)
    {
        if ((*it).a == atomId || (*it).b == atomId) count++;
    }
    return count;
}

/*
 ==========================================================
    'getBondBudget' Returns the number of spots left to
    form new bonds
 ==========================================================
 */
int Kekulize::getBondBudget(int atomId)
{
    Atom a = (*atomList)[atomId];
    int nbs = getNeighbourCount(atomId);
    int val = 0;

    if (a.symbol.compare("c") == 0) val = 4;

    if (a.symbol.compare("n") == 0)
    {
        if (nbs + a.charge > 3) val = 5;
        else val = 3;
    }

    if (a.symbol.compare("o") == 0) val = 2;

    if (a.symbol.compare("p") == 0)
    {
        if (nbs + a.charge > 3) val = 5;
        else val = 3;
    }

    if (a.symbol.compare("s") == 0)
    {
        if (nbs + a.charge > 4) val = 6;
        else if (nbs + a.charge > 2) val = 4;
        else val = 2;
    }

    return val - nbs - a.charge - a.hcount;
}

/*
 ==========================================================
```

```
    'aromaticBondsExist' Checks if there are aromatic bonds
    in the molecule
 ==========================================================
 */
bool Kekulize::aromaticBondsExist()
{
    std::vector<Bond>::iterator it;
    for (it = bondList->begin(); it != bondList->end(); it++)
    {
        if ((*it).order == -1) return true;
    }
    return false;
}

/*
 ==========================================================
    'isAromatic' Checks if a certain atom is aromatic
    (symbol)
 ==========================================================
 */
bool Kekulize::isAromatic(Atom a)
{
    std::string s = a.symbol;
    if (s.compare("c") == 0) return true;
    if (s.compare("n") == 0) return true;
    if (s.compare("o") == 0) return true;
    if (s.compare("p") == 0) return true;
    if (s.compare("s") == 0) return true;
    return false;
}

/*
 ==========================================================
    'aromaticToNormal' Converts aromatic symbols to normal
    symbols
 ==========================================================
 */
void Kekulize::aromaticToNormal()
{
    std::map<int,Atom>::iterator it;
    for (it = atomList.begin(); it != atomList.end(); it++)
    {
        std::string s = (*it).second.symbol;
        if (s.compare("c") (*it).second.symbol = "C";
        if (s.compare("n") (*it).second.symbol = "N";
        if (s.compare("o") (*it).second.symbol = "O";
        if (s.compare("p") (*it).second.symbol = "P";
        if (s.compare("s") (*it).second.symbol = "S";
    }
}

/*
 ==========================================================
    'findRings' Looks for cycles in the molecular graph
 ==========================================================
 */
void Kekulize::findRings() {
    int b;
    b = findCyclicUndefined(); //find the undefined bond
    while (b != -1)
    {
        (*bondList)[b].initial = true; //first bond of the cycle
```

```
        procBond(b); //process the bond (recursive process)
        resetVisited(); //reset information for the next iteration
        b = findCyclicUndefined();
        curRing++;
    }
}

/*
 ==========================================================
    'resetVisited' Set the visited flag of all bonds and
    atoms to false
 ==========================================================
 */
void Kekulize::resetVisited()
{
    std::map<int,Atom>::iterator ait;
    for (ait = atomList->begin(); ait != atomList->end(); ait++)
    {
        (*ait).second.visited = false;
    }

    std::vector<Bond>::iterator bit;
    for (bit = bondList->begin(); bit != bondList->end(); bit++)
    {
        (*bit).visited = false;
        (*bit).initial = false;
    }
}

/*
 ==========================================================
    'procAtom' Visit an atom in the cycle-finding algorithm
 ==========================================================
 */
bool Kekulize::procAtom(int atom, int prevBond)
{
    Atom *a = &(*atomList)[atom];
    if (a->visited)
    {
        return false;
    }
    bool res = false;
    a->visited = true;
    if (getNeighbourCount(atom) > 0)
    {
        std::vector<Bond>::iterator bit;
        int i = 0;
        for (bit = bondList->begin(); bit != bondList->end(); bit++)
        {
            if ((*bit).a == atom || (*bit).b == atom)
            {
              if (i != prevBond)
                {
                    res = res || procBond(i);
                }
            }
            i++;
        }

        if (res)
        {
            a->processed = true;
```

```
            a->cyclic = true;
            a->ring = curRing;
            return true;
        }
        else
        {
            a->processed = true;
            a->cyclic = false;
            return false;
        }
    }
    else
    {
        a->processed = true;
        a->cyclic = false;
        return false;
    }
}

/*
 ==========================================================================
    'procBond' Visit a bond in the cycle-finding algorithm
 ==========================================================================
 */
bool Kekulize::procBond(int bond)
{
    Bond *b = &(*bondList)[bond];
    if (b->visited)
    {
        b->processed = true;
        b->cyclic = true;
        b->ring = curRing;
        return true;
    }
    b->visited = true;
    bool res = false;
    if ((*atomList)[b->a].visited)
    {
        res = procAtom(b->b, bond);
    }
    else
    {
        res = procAtom(b->a, bond);
    }

    if (res)
    {
        b->processed = true;
        b->cyclic = true;
        b->ring = curRing;
        return true;
    }
    else
    {
        b->processed = true;
        b->cyclic = false;
        return false;
    }
}

/*
 ==========================================================================
```

```
    'findCyclicUndefined' Returns the first bond where no
    cyclic definition is defined
 ==========================================================
 */
int Kekulize::findCyclicUndefined()
{
    std::vector<Bond>::iterator bit;
    int i = 0;
    for (bit = bondList->begin(); bit != bondList->end(); bit++)
    {
        if (!(*bit).processed && (*bit).order == -1 &&
            !((*atomList)[(*bit).a].processed &&
             (*atomList)[(*bit).b].processed))
        {
            return i;
        }
        i++;
    }
    return -1;
}
```