Studiengang:	Computerlinguistik
Prüfer:	apl. Prof. Dr. Uwe Reyle HD Dr. Ulrich Heid
Betreuer:	apl. Prof. Dr. Uwe Reyle
begonnen am:	19. August 2005
beendet am:	22. Dezember 2005

Diplomarbeit Nr. 39

Analysing Names of Organic Chemical Compounds – From Morpho-Semantics to SMILES Strings and Classes

(Web Version)

Stefanie Anstein Gerhard Kremer

Universität Stuttgart

Institut für Maschinelle Sprachverarbeitung
Azenbergstraße 12
D–70174 Stuttgart

Erklärung

Wir versichern, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Alle Zitate und sinngemäßen Entlehnungen sind als solche unter genauer Angabe der Quelle gekennzeichnet.

(Ort, Datum) (Stefanie Anstein / Gerhard Kremer)

Aufteilung der Diplomarbeit

Diese Diplomarbeit wurde als Gruppenarbeit erstellt, wobei die Leistungen der einzelnen Kandidaten an Hand von inhaltlichen Kriterien zu unterscheiden sind. Der Kern des Systems entstand in Gemeinschaftsarbeit; die getrennt bearbeiteten Gegenstandsbereiche und Module sind in der folgenden Tabelle dargestellt. Die Aufteilung der schriftlichen Ausarbeitung ergibt sich aus den jeweils abgedeckten Themengebieten.

Themengebiet	Stefanie Anstein	Gerhard Kremer
Nomenklatur der organischen Verbindungen	●	
Nomenklatur der Kohlenhydrate		●
SMILES String Generierung		●
Klassifizierung	●	

Contents

Abstract

The linguistic analysis of chemical terminology is a key to biochemical text processing and semi-automatic database curation. The system described analyses systematic and semi-systematic names of chemical compounds, class terms, and also otherwise underspecified names by means of a morpho-semantic grammar developed according to IUPAC nomenclature. It yields an intermediate semantic representation which describes the information encoded in a name. Our tool provides SMILES strings for the mapping of names to their molecule structure and also classifies the analysed terms. It was implemented in Prolog as a prototype and a basis for further development to support research in the life sciences.

1 Introduction

The understanding of biochemical terminology is a key to many challenges in biochemistry, as unrecognised and undecoded terminology constitutes an essential bottleneck in processing the huge and growing amount of textual data available. The PubMed organisation[1] has collected around 13 million abstracts and publications until now and between 1,500 and 3,000 entries are added per day. An overview on "text mining in life science" is given by Fluck et al. (2005). According to Larsen (2005), about 42 % of biological information are still 'encoded' in journal files; only around 17 % are already structured in databases. To cope with this data in biochemical research, the fields of computational linguistics and applied information technology offer various methods of natural language processing (NLP). The development of adaptive tools for the semi-automatic extraction of knowledge, i. e. methods such as text mining or automatic summarisation, is essential to access and exploit the huge amount of textual data.

For all these interdisciplinary so-called BioNLP applications, terminology poses a serious challenge (see Krauthammer and Nenadić, 2004), because full sentence and text analysis is hampered by non-recognised names. To handle that, either huge, open-ended dictionaries of terms, a statistical approach, or methods of analysis and interpretation have to be used, the latter being our choice.

Grammar rules and lexica for the decomposition of biochemical terms can be implemented (an approach similar to the treatment of productive natural language phenomena) along the lines of organic compound[2] nomenclature systems. Names for organic compounds are too numerous to be explicitly enumerated but do have an internal structure that can be exploited with methods of computational linguistics. As they usually are the objects of interest in a text, their identification and classification is of great importance. Only with such a dynamic approach as opposed to the usage of static databases, the growing number of new terms can be coped with.

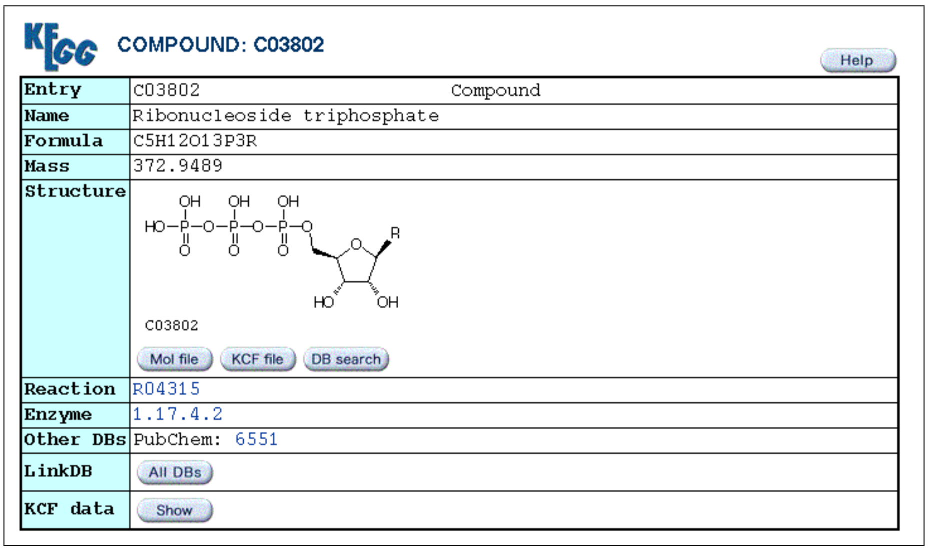
The challenge in dealing with terminology (e. g. for database curation), consists mainly in establishing links for term reference, i. e. to assign terms to their corresponding structure entry (see figure 1, an entry from the KEGG[3] chemical compound database). Another issue is to resolve coreferences in databases, i. e.

[1] PubMed is a service of the National Library of Medicine (NLM) and the National Institutes of Health (NIH).

[2] A compound in this context belongs to chemical terminology: an 'assembly' of atoms; it does not refer to a 'linguistic compound' (described in section 2.1).

[3] Kyoto Encyclopedia of Genes and Genomes (http://www.genome.ad.jp/kegg)

to detect different entries referring to one and the same structure (figures 1 and 2).

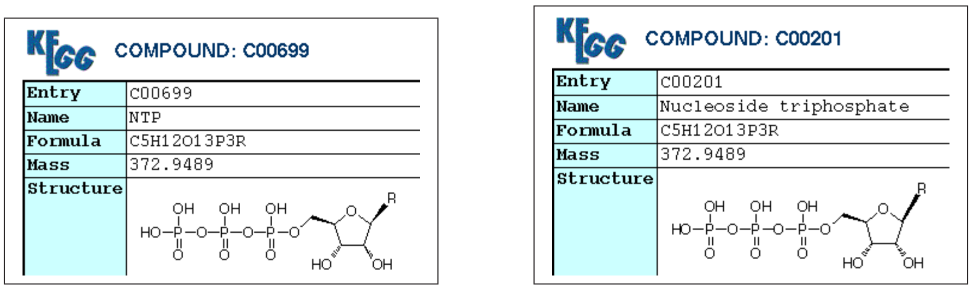


KEGG COMPOUND: C03802

Help

Entry	C03802 Compound
Name	Ribonucleoside triphosphate
Formula	C5H12O13P3R
Mass	372.9489
Structure	C03802 Mol file KCF file DB search
Reaction	R04315
Enzyme	1.17.4.2
Other DBs	PubChem: 6551
LinkDB	All DBs
KCF data	Show

Figure 1: Example entry from a chemical compound database



KEGG COMPOUND: C00699

Entry	C00699
Name	NTP
Formula	C5H12O13P3R
Mass	372.9489
Structure	

KEGG COMPOUND: C00201

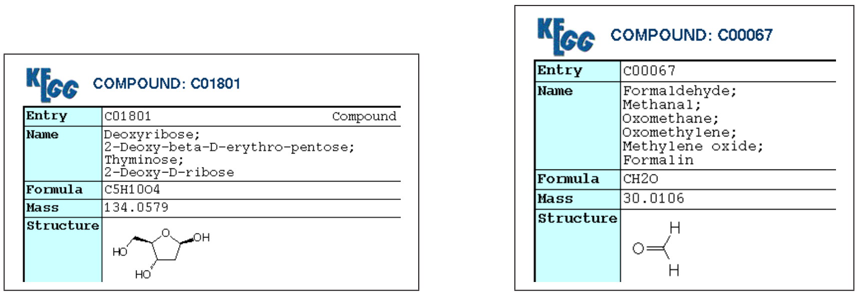
Entry	C00201
Name	Nucleoside triphosphate
Formula	C5H12O13P3R
Mass	372.9489
Structure	

(a) (b)

Figure 2: Multiple database entries for one compound

The variability of nomenclature principles, combined with the occurrence of alternative rules in each of them (as elaborated in section 2.2), leads to a number of synonymous compound names. Examples for such alternative names for a single structure are shown in figure 3.

Few tools for the linguistic processing of biochemical terms have been developed; none of these provide the representations of molecule structure or classify terms based on the name. Many systems of a similar kind allow for a


COMPOUND: C01801

Entry	C01801	Compound

Name	Deoxyribose;
	2-Deoxy-beta-D-erythro-pentose;
	Thyminose;
	2-Deoxy-D-ribose	

Formula	C5H10O4	
Mass	134.0579	
Structure		

COMPOUND: C00067

Entry	C00067

Name	Formaldehyde;
	Methanal;
	Oxomethane;
	Oxomethylene;
	Methylene oxide;
	Formalin

Formula	CH2O
Mass	30.0106
Structure	

(a) (b)

Figure 3: Database entries containing several synonymous compound names

search in static databases and lexica and can thus not deal with newly emerged names, not to mention underspecified ones, where the name not fully reflects the molecular structure.

Our work is based on Reyle (2005). This article contains a proposal for a deep semantic analysis of chemical terminology; formal properties of a system for the understanding of terms including underspecified ones are described there.

We implemented, on the basis of chemical nomenclature rules and domain knowledge such as chemical defaults[4], a parser in Sicstus 'Prolog'[5] which analyses names of organic chemical compounds. It conducts a morphological decomposition and semantics construction, the output of which is expressed in our semantic representation language. Based on this representation, our system reconstructs molecular structure as far as it is made explicit in such a name and assigns the corresponding SMILES string (a simple representation of the molecule structure). The system also classifies names according to functional and structural properties, which are expressed in the names' parts. Preliminarily and for testing the approach, SMILES strings are generated for carbohydrate[6] compounds and a classification is provided for other organic compounds.

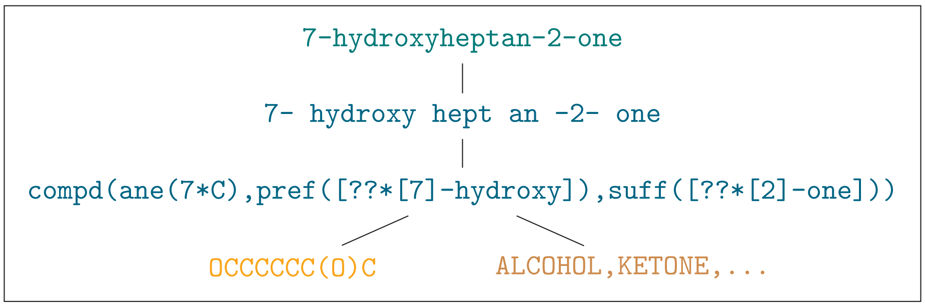
An example analysis to demonstrate the procedure of the system is shown in figure 4. The morphemes of the compound name in the top line are separated (second line) and a semantic analysis is conducted which yields the corresponding semantic representation as shown in the third line. From this representation,

[4]A default signifies an option that is selected automatically unless an alternative is specified, e. g. in this context, the filling of free valences with hydrogen atoms.

[5]Logic Programming Language (see http://www.sics.se/sicstus/docs/latest/html/sicstus.html)

[6]We will use the term 'sugar' for carbohydrate in the following; see also section 2.2.

a SMILES string as description of the molecular structure and a class list are generated (bottom line, from left to right).



```
7-hydroxyheptan-2-one
7- hydroxy hept an -2- one
compd(ane(7*C),pref([??*[7]-hydroxy]),suff([??*[2]-one]))
OCCCCCC(O)C        ALCOHOL,KETONE,...
```

Figure 4: Example analysis

We chose, as a starting point, the wide field of organic chemistry (containing e. g. alkanes, alkenes, alcohols, etc.) with a special focus, so to speak, on carbohydrates by going into depth there. Even though the nomenclature principles for general organic compounds and carbohydrates are quite distinct – carbohydrates as a part of organic chemistry have special principles of nomenclature – we developed a common grammar for the two parts because their names can contain each other mutually.[7]

Our linguistic approach as opposed to a molecule structure-based one has the advantage that underspecified names such as `butene`, for which no distinct structure can be depicted, can be analysed as well. In this example, the information on where the double bond (`-ene`) is located is missing. Such underspecified names, which are a frequent phenomenon in biochemical data, can be handled by generating partial structures. Their classification can help to retrieve a more detailed description for their further processing.

The system is able to deal with systematic, trivial and semi-systematic chemical terms of organic substances, chemical class names as well as semi-systematic class names. Examples for each name type are presented in table 1. For (semi-)systematic terms, both fully specified and underspecified forms occur. Underspecified trivial names do not exist; class names or semi-systematic class names are inherently underspecified.

The tool's ability to cope with underspecification and class names distinguishes it from existing systems as described in the section on related work (section 3).

[7]As the closely related topics heavily overlap, it seems reasonable to provide all information together instead of spreading it in different diploma theses.

	fully specified	underspecified
systematic names	`7-hydroxyheptan-2-one`	`butene`
trivial names	`benzene`	∅
semi-systematic	`benzene-1,3,5-triacetic acid`	`dihydrobenzene`
class names	∅	`sugar`
semi-systematic	∅	`2-deoxysugar`

Table 1: Examples for existing compound name types

With this tool, we created a valuable basis for term reference and coreference resolution. Nomenclature-based synonyms can be identified by either matching their semantic representation or their SMILES strings (`2-pentulose` and `pent-2-ulose` yield the same output). Ambiguities (possible readings of a term) are partially resolved by a consistency check of the information contained in the term.

The limitations of our system are the following: Our parser is not exhaustive for all organic chemistry but (in the context of this work) constitutes a fragment as a starting point, model and template. The rules are only formulated for the purpose of analysis; our system is not meant for name generation from structures even though that would be theoretically possible. The Prolog parsing algorithm chosen is not very efficient compared to other parser implementations, however, the formalism is ideal for intuitive and traceable morpho-semantic decomposition of linguistic entities.

To conduct large-scale qualitative evaluation of our tool was not possible because, on the one hand, the system is not meant to be exhaustive and, on the other hand, no data containing annotated term analyses exist to refer to for an assessment.

To summarise, the objectives of this project are to provide a semantic normalisation of biochemical terminology. Applications range from the support of text processing technologies for the growing amount of textual information to the automation of biochemical ontology-based[8] database population and curation. The challenge thereby consists in covering all existing variations of terminology and the resolution of underspecification and ambiguity.

After this general introduction, a description of the theoretical background to provide a common basis is presented in section 2. To range our work in the state of the art, information on related work follows in section 3. Section 4 presents

[8]An ontology is a systematically defined comprehensive system of concepts and objects which are linked by relations such as ‘is a’ or ‘part of’.

the system details, from a general overview to possible applications. In the outlook (section 5), perspectives for an extension of the system are enumerated. After a summary and conclusion, the program source files, a specification of our semantic representation language, and our testsuite are given in the appendix.

2 Theoretical Background

In order to provide a common basis for the explanation of our system's details, we introduce important notions and terms in the areas of linguistics and computational linguistics as well as some biochemical background in this section.

2.1 Computational Linguistics

Central concepts of linguistics with related terms will be introduced and complemented by the relevant notions in the special field of computational linguistics.

Morphology

The linguistic field of morphology deals with the study of the internal structure of words. It examines the elements which are used to form simplex words, complex words, or bigger grammatical units, so-called multi-word expressions. These elements are termed morphemes; they are defined as the smallest meaningful units of a language. Morphology is divided into the two principal components, inflection and word formation, where the latter is the field of main interest in the context of this work. An overview of the topic is presented by Spencer (1991).

There are various word formation processes, the most frequent and productive[9] ones being compounding and derivation. The former process combines several morphemes which can each stand alone as an independent word (free morphemes) such as in [[black][bird]] or [[apple] [tree]]. In the latter process, free and bound morphemes, the latter of which can only occur when attached to other morphemes, are combined (e. g. as in [[colour][ful]]). These bound morphemes are called affixes. They are divided into prefixes, infixes, and suffixes according to their position within the word they are part of.

Structurally complex words and multi-word expressions which are transparent, i. e. morphologically and semantically deconstructable, can be parsed. Morphological parsers analyse their structure and assign a 'parse tree', which shows the hierarchical relations of their parts.

In English, where linguistic compounds in most cases contain blanks, the distinction between morphology and syntax, the study of sentence structure, is not always clear. Multi-word expressions (which also occur especially in chemical terminology) have to be recognised as belonging together.

[9]The term productivity refers to the property of a word formation pattern to generate 'new' words.

Special terms of scientific domains, e. g. chemical compound names, are similar to complex and transparent non-terminological words of natural languages. Therefore, transferable methods of word analysis according to ‘word formation rules’ can be applied.

Semantics

The field of semantics studies the meaning of linguistic entities. In computational semantics, tools are developed to deal with the construction of meaning. Various theoretical devices exist, e. g. different levels of semantic or logic representation languages. An overview (in German) is given in Carstensen et al. (2004, chap. 2 and 3.5). Predicate logic uses predicates with a defined number of arguments (valency) and a corresponding truth value for the description of linguistic entities, such as ‘student(Tom)’ with the truth value ‘1’ for the sentence ‘Tom is a student.’. The Montague semantics is based on formal logic and applies, e. g., the lambda calculus (in the following: λ-calculus).[10] It is a means to compute the meaning of a complex entity, where its parts contribute their semantics to compositionally calculate the meaning of the whole.[11] The λ-expressions used abstract over properties of, e. g., objects (i. e. over predicate arguments) and express the application of functors to arguments (represented by the symbol @), which is then resolved with a so-termed beta reduction (β-reduction). An example expression is ‘λX.student(X)@tom’, which is β-reduced to ‘student(tom)’. In this work, we do shallow semantic processing as opposed to a deep interpretation or ‘understanding’ by computer programs.

Ambiguity A linguistic entity is ambiguous if it has two or more possible readings, usually with differing probabilities for an interpretation in a given context. The words ‘cold’ or ‘bank’ are standard examples for semantic ambiguities, the former having adjective or noun as parts of speech and the latter meaning the institution or the furniture. For structural ambiguities, several parse trees for complex entities can be generated. (1) is an example for a syntactically ambiguous sentence.

(1) She can see the student with the telescope.

In sophisticated NLP systems, methods of disambiguisation are being used. These either rely on the linguistic context or apply frequency statistics.

[10] See, for example, Lohnstein (1996) for a detailed introduction (in German).
[11] See Blackburn and Bos (2005) for a comprehensive introduction to compositional semantics.

Underspecification The term underspecification describes the fact that a certain feature of a linguistic entity to be definite and unambiguous is missing. The characteristics of the entity are thus not fully specified. Usually, the missing information can be deduced from the linguistic or other context (resolvable underspecification). In other cases, it is simply not relevant or aimed to be abstracted from in a certain application – the underspecification cannot be resolved. For example, for `ethene` (`C=C`[12]), the position of the double bond expressed by `-ene` is clear even though not indicated because there is only one possibility, whereas `butene` can be used to refer to either `C=CCC` or `CC=CC`. In order to deal with underspecified linguistic units, a semantic representation expressing the kind of underspecification as well as a resolution algorithm have to be designed. More information on underspecification can be found via the underspecification bibliography[13] of the Institute for Natural Language Processing (IMS), University of Stuttgart.

Coreference From a general linguistic point of view, coreference means the 'pointing' of two distinct terms, usually common nouns or proper names, to one concrete single instance of an object or person. For example, the definite description 'the developer of the SMILES chemical language' and 'David Weininger' refer to one and the same person.

Chemical compound names are usually used in two kinds of readings depending on their context. On the one hand, in a sentence such as (2), the name `benzene` is used as a non-count or mass noun referring to the substance class. A mass noun is a type of common noun that represents a substance not easily quantified by a number, e. g. 'knowledge'.

(2) "Benzene is used in the manufacture of plastics, [. . .]"[14]

On the other hand, in a context such as (3), the concrete molecular structure is referred to.

(3) "Benzene is a six membered ring [. . .]"[15]

Usually, when `benzene` is used in scientific research publications, e. g. in reaction equations, the structure of a typical molecule is denoted.

[12]Details on such SMILES string representations can be found in subsection 2.2.
[13]http://www.ims.uni-stuttgart.de/projekte/sfb/b3/ul-bib.html
[14]http://benzene.lifetips.com/cat/61153/uses-of-benzene
[15]http://www.factbites.com/search.php?kp=Phenyl+ring

Presupposition An implicit or explicit assumption which underlies, e. g., a statement, is termed a presupposition. In the following example, (4b) is a presupposition of (4a).

(4) a. 'Why is Tom studying?'
    b. 'Tom is studying.'

Axiom An axiom is a well-established principle or rule. The term denotes a statement whose truth is taken to be obvious and self-evident without proof. An example from mathematics would be (5a); (5b) presents one related to chemistry.

(5) a. 'Two things equal to the same thing are equal to each other.'[16]
    b. 'Primary alcohols are alcohols.'

Grammars and Prolog

A grammar consists of a set of rules with which, on the one hand, a language is described and, on the other hand, a linguistic entity can be deconstructed into its parts. According to the Chomsky hierarchy, there are four kinds of grammars, increasingly constrictive (concerning the generated languages) from unrestricted grammars via context-sensitive grammars and context-free grammars to regular grammars.[17]

Depending on their configuration, rule-based (also called symbolic) grammar systems can be overanalysing, i. e. not rejecting incorrect input. If such grammars are used for generation, they in turn overgenerate, i. e. they produce incorrect output.

The logic programming language Prolog is ideal for dealing with the analysis of structurally complex linguistic entities.

> "Almost from its origin, the development of logic programming has been closely tied to the search for computational formalisms for expressing syntactic and semantic analyses of natural language sentences."[18]

The common framework to parse linguistic units in Prolog is the Definite Clause Grammar (DCG) formalism, a top-down, left-to-right symbolic parsing algorithm. A DCG rule consists of terminal and non-terminal symbols and

[16]http://www.answers.com/topic/axiom
[17]For a brief introduction, see e. g. http://en.wikipedia.org/wiki/Chomsky_hierarchy.
[18]http://www.let.uu.nl/~Willemijn.Vermaat/personal/courses/prolog

defines their possible substituents. Non-terminal symbols are expanded until they are replaced by a sequence of terminal symbols; terminal symbols are substituted by a corresponding lexicon entry. DCG rules in Prolog are written by use of the DCG operator '`-->`'. When coding a left-to-right parsing algorithm, left-recursive rules should be avoided in order not to risk endless loops. Rules are called left-recursive if a non-terminal symbol appears both as the symbol to be expanded and as the first symbol to be expanded to. This property can also be obtained indirectly by so-called chain productions. An example for the first possibility is a rule '`A` → `A a`'; the second case could look like '`A` → `B`' and '`B` → `A a`'. Symbols can have additional annotations as arguments (see (6)), which allow for syntax or semantics construction, for example.

(6) `organic_compound(O_C-Semantics)`

For the latter, the λ-calculus can be used within the DCG.

Basic Prolog features are the facts, which consist of functors with arguments, and rules (or clauses) with their corresponding head and body, presented in (7a) and (7b), respectively.

(7) a. Prolog fact:
`functor(arg1,arg2,...).`
b. Prolog clause:
`head_of_rule :- body_of_rule.`

Prolog is a programming language which attempts to satisfy goals by processing conditions in the form of clauses and facts; the unification mechanism, which tries to 'match' variables and atoms, serves this purpose. An atom denominates the simplest entity possible, e. g. 'tom'; variables begin with a capital letter or an underscore; in the latter case the name of the variable is disregarded for unification. For more details, see e. g. Clocksin and Mellish (1987).

2.2 Biochemistry

The basis for our work are naming conventions for chemical compounds, especially the general ones for organic compounds and those for carbohydrates. An overview together with some term definitions are presented in this subsection. Additionally, we describe here the SMILES notation, which we used to assign a molecular structure to each name, and give a brief introduction to the classification of organic chemical compound names.

Nomenclature Systems

A chemical nomenclature system defines a standard for the naming procedure of molecules. Its purpose is to provide a common basis for chemists to assign a name to a molecule structure, so that it identifies the chemical species and its structural properties and can be correctly interpreted by others knowing the same nomenclature rules. That is, for being useful in scientific communication in biochemistry, common nomenclatures had to be established.

Historical Background The first attempt to organise international organic chemical nomenclature was at the Geneva Conference in 1892,[19] which established a basis for a chemical nomenclature to evolve – one of the main interests was to bring up an international standard.[20] As a consequence, in 1919 the International Union of Pure and Applied Chemistry (IUPAC) was formed, one of its aims being the standardisation of names in chemistry.

Other initiatives exist to promote the use of nomenclature systems, e. g. the Joint Commission on Biochemical Nomenclature of IUPAC and IUBMB[21] (JCBN) and the Nomenclature Committee of the International Union of Biochemistry and Molecular Biology (NC-IUBMB).

> "The Mission of the IUBMB is to foster and support the growth and advancement of biochemistry and molecular biology as the foundation from which the biomolecular sciences derive their basic ideas and techniques in the service of mankind. This it does throughout the world [. . .] by promoting international cooperation [and high standards in research, discussion, application and publication, and] through international standardization of methods, nomenclature and symbols [. . .] The IUBMB promotes the norms, values, standards and ethics of science and the free and unhampered movement of scientists [. . .]"[22]

In the past decades, computers are increasingly involved in dealing with chemical structures and names, and processing and storing chemical data in its digital form is a common interest. Consequently, new institutions have emerged, e. g. the Chemical Nomenclature and Structure Representation Division was established by IUPAC in 2002. It is "responsible for maintaining

[19] http://www.iupac.org/general/about.html
[20] http://www.chemheritage.org/explore/timeline/NOMEN.HTM
[21] IUBMB: International Union of Biochemistry and Molecular Biology
[22] http://www.iubmb.unibe.ch/Standing_Orders/Mission.htm

and developing standard systems for designating chemical structures, including both conventional nomenclature and computer-based systems" (cited from http://www.iupac.org/divisions/VIII/index.html). There also exists an online naming service of ACD/Labs[23] in collaboration with the IUPAC Committee on Chemical Identity and Nomenclature Systems to facilitate and promote the use of the nomenclature.[24] This service predicts a name according to IUPAC nomenclature rules for each a drawn chemical structure used as input.

Naming Variability As international nomenclature systems had not been established before people realised their need, chemical compound names not corresponding to such rules are used. They had been employed in literature prior to the existence of international nomenclature systems, either created systematically by a regional nomenclature system or coined as trivial names. Furthermore, other naming conventions (and IUPAC nomenclature 'dialects') which are used in addition to IUPAC's international nomenclature system exist and yield still other name variants (see e. g. Chemical Abstracts Services, 2002; Beilstein, 1997)[25].

Another point to consider is that the rules defined in IUPAC's nomenclature system are only recommendations, i. e. there is and can be no enforcement to use them. For that reason, chemical compound names mentioned in texts do not always correspond to nomenclature rules. Names that are "used but not recommended" occur, caused, e. g., by misinterpretation of rules or motivated as mentioned in IUPAC Commission on Nomenclature of Organic Chemistry (1993):

> "Occasionally, an author may wish to convey a particular emphasis by assigning a name which departs from standard priority considerations. So long as this is adequately explained and appropriate consequences (e. g., for numbering) accepted and logically treated so as to preserve freedom from error and ambiguity, this set of procedures may still be applicable. However, names so generated are not recommended for general use."

The other way round, there are recommended names which do not occur. Diachrony is also to be taken into account, as nomenclature rules change over time and yield contradicting recommendations – not to mention the number of morpho-syntactic variations (with hyphens, brackets, space etc.).

[23]Advanced Chemistry Development, a chemistry software company
[24]http://www.iupac.org/nomenclature and http://www.acdlabs.com/products/name_lab
[25]"Beilstein's Handbook of Organic Chemistry"

Systematic names proposed by nomenclatures are not the only form of terms used in literature. Especially, class names, established trivial names, and semi-systematic names which are a combination of trivial or class names and systematic names do appear.

IUPAC Nomenclature Rules

We chose for this work – as a starting point – the IUPAC nomenclature for organic compounds (IUPAC Commission on Nomenclature of Organic Chemistry, 1993) and for carbohydrates (IUPAC-IUBMB Joint Commission on Biochemical Nomenclature, 1996). There also exist more recent, however provisional, nomenclature recommendations, e g. to meet the changing requirements for chemical nomenclatures depending on latest research.[26]

> "Provisional Recommendations are drafts of IUPAC recommendations on terminology, nomenclature, and symbols made widely available to allow interested parties to comment before the recommendations are finally revised and published in Pure and Applied Chemistry."[27]

Although carbohydrates (we will use the term 'sugars'[28] in this work) can also be named according to the general nomenclature system for organic compounds, the existing separate, special nomenclature for sugar names is more useful as it provides specific abbreviations to facilitate their naming.

In the field of biochemistry nomenclatures define a language for assigning a name to a molecule structure of a chemical compound. They prescribe the morphemes as well as the grammar rules for combining these morphemes. To form a valid name, morphemes and rules have to be used according to certain aspects of the molecular structure, which are described in these nomenclatures, too. In the following, some of these rules are presented.[29]

The morphemes used in compound naming refer to certain molecules (e. g. `phospha-`), operations (`deoxy-`), locants (`3`), or multipliers (`hexa-`). They are

[26] http://www.iupac.org/reports/provisional/archives.html

[27] http://www.iupac.org/reports/provisional

[28] The two terms are not truly equivalent: "The term 'carbohydrate' comprises monosaccharides, oligosaccharides and polysaccharides as well as their derivatives [. . .] The term 'sugar' is frequently applied to monosaccharides and lower oligosaccharides." (see IUPAC-IUBMB Joint Commission on Biochemical Nomenclature, 1996). We used this term correctly in the respect that the current system exclusively analyses monosaccharides.

[29] The sample rule specifications given here are literally cited from IUPAC Commission on Nomenclature of Organic Chemistry (1993) and IUPAC-IUBMB Joint Commission on Biochemical Nomenclature (1996).

attached to the parent name either as prefixes or suffixes. Some of them have a prescribed order defined in the nomenclature rules. For example, so-called detachable prefixes modify parent structures and precede nondetachable prefixes, which have to be attached directly to the parent name.

(8) R-0. 1.8.3
Detachable prefixes describing substituents are cited preceding nondetachable prefixes (see R-0. 1.8.1 and R-0. 1.8.2), if any, [...]

Several operation types are defined in the organic compound nomenclature, in particular substitutive, replacement, additive, conjunctive, subtractive, ring formation and cleavage, rearrangement, and multiplicative operation (R-1.2). Each is expressed by specific morphemes and rules, e. g.:

(9) R-1. 2.1 Substitutive Operation
The substitutive operation involves the exchange of one or more hydrogen atoms for another atom or group. This process is expressed by a prefix or suffix denoting the atom or group being introduced (see R-3.2 and R-4 for lists of prefixes and suffixes).

A configurational prefix (especially needed in sugar naming) describes the configuration[30] of up to four carbon atoms with their hydrogen (`H`) and an hydroxygen group (`OH`) attached.

(10) 2-Carb-8.3. Multiple configurational prefixes
An aldose containing more than four chiral centres is named by adding two or more configurational prefixes to the stem name. Prefixes are assigned in order to the chiral centres in groups of four, beginning with the group proximal to C-1. The prefix relating to the group of carbon atom(s) farthest from C-1 (which may contain less than four atoms) is cited first.

Some nomenclature rules also prescribe the usage of lower- or uppercase characters, super- or sub-scripts, different fonts, etc., for certain morphemes.

(11) R-0. 1.6.2
Italicized element symbols, such as *O*-, *N*-, *P*-, *S*-, are locants indicating attachment to these heteroatoms.

Although IUPAC nomenclature rules describe the naming of a compound, i. e. what is allowed and what should be avoided, it is intended to produce unambiguous, not unique names for a given chemical compound – synonymous names for one structure are possible to emerge (`2-pentene` and `pent-2-ene`).

[30] Configurations designate the stereochemical properties of a structure. For a list of stereochemistry terms and their explanation see http://www.chem.qmul.ac.uk/iupac/stereo.

The other way round, it cannot be directly inferred from differing names that they relate to differing structures. They could be synonyms, which has to be checked. To summarise, between names and molecular structures, relations of the following kind exist: one name can refer to different structures (it is an ambiguous, underspecified or class name, e.g. `hexulose`) and one structure can have several names (synonymous names) according to, e. g., various nomenclature rules.

In the nomenclatures chosen as a basis for our system, we encountered several unclear cases which partly made it difficult to implement the naming rules. These cases include, e. g., complex example names[31] for simple rules, amongst others.[32] Some examples for the unclear rules are listed and described in (12).

(12) a. R-5. 7.2.1, where the explanation is about `sulfo-` and `sulfino-`, whereas the example uses the prefix `sulfono-`,
b. R-6, where the systematic description with (a) to (c) does not correspond to exactly such a numbering in the examples,
c. R-0. 1.7.3
Addition of the vowel “o”. For euphonic reasons, the vowel “o” is sometimes inserted between consonants.

General Naming Principles

For chemical structures named after the general nomenclature for organic compounds, the following principles are applied (see also R-4): First, the operation type is determined. The principal characteristic/functional group[33] to be named as the suffix is then chosen. After that, the parent structure and the non-detachable prefixes have to be determined and all parts are named (with an alphabetical order for prefixes) and numbered.

The naming procedure for sugars[34] is as follows: First, the parent monosaccharide, being the longest chain of carbon atoms, is determined. The corresponding aldose ($\mathtt{H\text{-}[CHOH]_n\text{-}CHO}$) or ketose ($\mathtt{H\text{-}[CHOH]_n\text{-}CO\text{-}[CHOH_n\text{-}H}$) in its original (acyclic and unmodified) form is the basis for naming. After numbering the carbon atoms, the parent name is chosen. The configurations of the `CHOH` groups are also specified in the parent name by means of configurational

[31]However, for chemists, such rules requiring more chemical knowledge may not pose problems.

[32]By the way, this could be one reason for preventing people from using the nomenclature.

[33]A functional group is defined as an atom or group of atoms in an organic compound that gives the compound some of its characteristic properties, such as the `C=O` functional group in aldehydes and ketones.

[34]For larger molecules where more than one monosaccharide structure is embedded, see IUPAC-IUBMB Joint Commission on Biochemical Nomenclature (1996).

symbols and prefixes. After that, the side branches of the parent are determined and named, and the corresponding affixes are attached to the parent name.

We used the nomenclature rules as a basis to determine and construct the corresponding molecular structure of a name instead of naming a structure, i. e. the intrinsic use of the nomenclatures is reversed. The rules are not easy to be rewritten as a grammar (to build a name analyser) as they comprise linking morphemes and the deletion of single characters as well as replacement and deletion of morpheme parts.

SMILES Strings

The Simplified Molecular Input Line Entry System (SMILES), provided and supported by Daylight Chemical Information Systems[35] and initially developed by Weininger (1988), is a 'nomenclature system' used to represent molecules in a line notation. This allows to process them with computer programs. It is also possible to reconstruct the two- or three-dimensional model from the so-called SMILES string. For example, `L-threo-Tetrodialdose` can be represented in SMILES notation as in (13).

(13) `C(=O)[C@]([H])(O)[C@@]([H])(O)C(=O)`

The corresponding molecular structure is depicted in figure 5(a). An example including a ring connection is shown in figure 5(b) (α-`D-threo-Hexo-2,4-diulo-2,5-furanose`); a SMILES string representing this structure can be written as in (14).

(14) `C([H])(O)[C@@]2(O)[C@@]([H])(O)C(=O)C(O2)C([H])(O)`

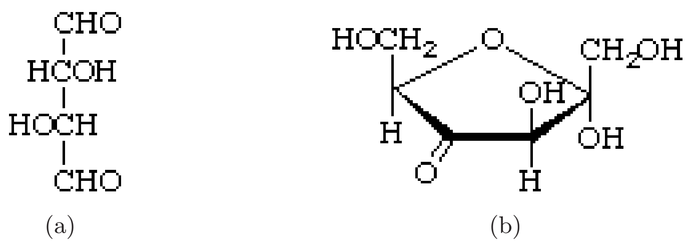


Figure 5: Example structures for (a) `L-threo-Tetrodialdose` (in Fischer projection) and (b) α-`D-threo-Hexo-2,4-diulo-2,5-furanose` (in Haworth representation). Both pictures are from IUPAC-IUBMB Joint Commission on Biochemical Nomenclature (1996).

[35]http://www.daylight.com/smiles

The Depict tool[36] is a program for converting a SMILES string into a two-dimensional structure displayed using the Haworth representation[37]

SMILES is one of a family of languages. Related languages are, e. g., USMILES, ASMILES, SMARTS, or CHUCKLES as described at http://msdlocal.ebi.ac.uk/docs/daylight/smiles/smiles-relatives.html. A molecule can be described by a SMILES string in various ways (the naming process 'walks through' a structure and stores the encountered atoms and bonds), whereas, e. g. USMILES is a language to provide a unique string for each molecular structure. An algorithm to convert a SMILES string into a unique SMILES string representation is described in Weininger et al. (1989).

SMILES Rules The following list contains simplified SMILES rules for representing a molecular structure. The complete SMILES nomenclature and a tutorial can be found at http://www.daylight.com/smiles.

(15) a. Atoms are represented by their standard atomic symbol, e. g. `C` for a carbon atom. Hydrogen atoms can be omitted; otherwise, they appear in brackets (`[H]`).
b. Single bonds can be omitted or are represented by '`-`', double bonds by '`=`', and triple bonds by '`#`'.
c. Branches are enclosed by parentheses (`(...)`); they may be nested.
d. Ring closure bonds are represented by matching digits as indices behind the specifications of the joined atoms (e. g. `C1CCCC1` for a cyclic form of a `pentose`).
e. Tetrahedral chirality is specified as an atomic property right behind the respective atom (anti-clockwise order of atoms: '`@`', clockwise order: '`@@`'); the whole expression comprising atom and its properties is enclosed in brackets (see figure 6).

All characters listed are arranged without spaces anywhere between. Generally, each atom is described separately with its properties specified, if necessary. Its potential branches are appended thereafter.

Classification

Chemical compounds can be subsumed by generic categories according to structural and functional properties such as characteristic/functional groups. One

[36]This tool is provided by Daylight Chemical Information Systems; an online version is available at http://www.daylight.com/daycgi/depict.

[37]We used this tool in the development of the SMILES string generator for checking the correctness of the generated SMILES string.

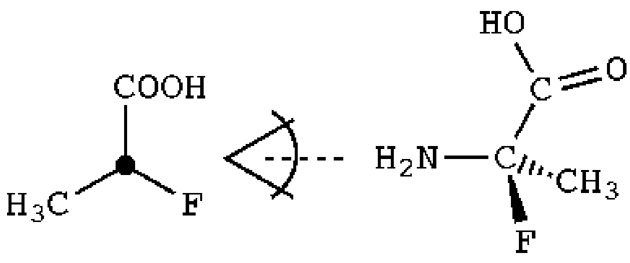


Figure 6: Tetrahedral chirality in `SMILES` notation. For `N[C@](C)(F)C(=O)O`, looking at the chiral center `[C@]` from the `N` atom, the atoms are anticlockwise in order: `C`, `F`, `C(=O)` (picture is from http://www.daylight.com/smiles/smiles-isomers.html#TETRA).

compound can belong to several (not necessarily same-level) classes. By determining super- and sub-classes of compound names, a hierarchy of categories can be established. For a glossary of classes of organic compounds defined by IUPAC see http://www.chem.qmul.ac.uk/iupac/class.

3 Related Work

To present the state of the art as to the processing of chemical compound names, we give a survey of previous work in this section. Relevant existing tools are listed and briefly described, followed by a summarising subsection.

3.1 Existing Systems

The following paragraphs present several systems similar to ours. Tools along the lines of our modules are described first, supplemented by the presentation of several additional resources.

Parsers

The series of papers Cooke-Fox et al. (1989a,b,c, 1990a,b) and Kirby et al. (1991) describe an early approach to systematic chemical compound name analysis. We could, however, not evaluate this system because it is not available online.

The tool 'Chemfinder'[38] provides the molecular structure of a queried name as an image as well as more detailed information, including a list of synonyms. It is based on a linguistic analysis of the name and also accepts deviations from official rules (IUPAC, IUBMB and CAS). Underspecified compounds are not covered. The system underlying Chemfinder is the parser Name=Struct as described in Brecher (1999) and Cambridgesoft (2005).

The system for a "semantic analysis of names of organic compounds" developed by Gerstenberger (2001) in a student research project analyses names of organic chemical compounds both syntactically and semantically. It does not deal with underspecification; besides, the lexicon was not intended to be comprehensive, which is one reason why the system is currently not used in any applications.

The 'ACD/Name'[39] tools provided by ScienceServe/Scientific Software Solutions offer the generation of structures from systematic names and, vice versa, the naming of given structures. As it is a commercial tool and no detailed documentation is available, we were not able to find out what kind of approach it is based on, i. e. if it conducts an online systematical name analysis or relies on a static database, nor if it handles underspecification.

[38] http://chemfinder.cambridgesoft.com
[39] http://www.scienceserve.com/Software/ACD/ACD_Name.htm

SMILES String Generators

All the spotted systems providing SMILES strings, such as 'Accelrys' (http://www.accelrys.com) or the 'ChemAxon' products (http://www.chemaxon.com), take a graphical compound structure as input; none of them processes compound names linguistically.

Classifiers

We did not locate any compound classifier which directly computes classes on the basis of compound names. One example providing non-dynamic classification by means of a fix hierarchy of classes assigned to specified compound names is the 'PAREO' system (http://genome.jouy.inra.fr/pareo). Wittig et al. (2004) developed a classification tool for compound names on the basis of SMILES strings by conducting a subgraph search for functional groups. As SMILES strings cannot be determined for underspecified compound names, this system does not deal with underspecification.

Additional Resources

On the PubChem website[40] of the National Center for Biotechnology Information (NCBI), the 'PubChem Compound' tool presents a search for unique chemical structures using, e. g., names. It offers to answer questions such as "which compounds have `tylenol` as (a part of) their molecule name". The tool seems to use simple regular expressions over the queried compound names for its search in the database. It does not conduct an analysis, which is why compositionally generated new names cannot be found there; underspecified terms are also not included.

A similar repository is 'ChEBI'[41], a "freely available dictionary of small molecular entities". It offers a wildcard search in its database, which provides structures, SMILES strings, etc. for a static set of specified compounds.

Other resources are 'Klotho'[42], a "Biochemical Compounds Declarative Database" offering a search using regular expressions of compound names, or 'Whatizit'[43], a tool providing links to different database entries containing corresponding SMILES strings, for example. Neither system contains underspeci-

[40]http://pubchem.ncbi.nlm.nih.gov
[41]http://www.ebi.ac.uk/chebi
[42]http://www.biocheminfo.org/klotho
[43]http://www.ebi.ac.uk/Rebholz-srv/whatizit/form.jsp

fied entries. More such resources can be found on the site of the “Information centre for chemistry, biology, pharmacy and related fields”[44].

3.2 Conclusion

The majority of the systems described above do not totally correspond to the functionality of our tool – either they take a different kind of input (e. g. graphical), or they produce their output according to static databases. We are only aware of two tools which process names of chemical compounds linguistically, one of which being not developed far enough to be used practically (Gerstenberger, 2001) and both not covering underspecification (Gerstenberger, 2001; Brecher, 1999). All the others do not conduct an analysis but are based on fix databases, which is why they do not cover systematically and productively formed new compound names. None of the systems deals with cases of underspecification, which is a serious drawback as such names appear frequently in literature; they are, e. g., used in publications and reaction equations to generalise information. To the best of our knowledge, no comparable tool for the linguistic processing of biochemical terms exists and our system clearly extends the functionalities, even though not the quantitative coverage of similar ones.

[44] http://www.infochembio.ethz.ch/links/index.html (in German)

4 The System

In this section, we describe the system developed in this project in more detail – after the general background and a detailed description of the three modules, a specification of the results is provided and several possible applications of the tool are given.

4.1 Overview

This subsection will present a general introduction to the system. Starting with the basics, we then depict the architecture, specify the input and output, and in the end show a detailed example analysis.

General Information

The aim of our work was to build a flexible analysing system for names of organic chemical compounds according to IUPAC nomenclature which is able to deal with (semi-)systematic, trivial, and class names including underspecification.[45] The semantic representation yielded by the parser is transformed to a SMILES string specifying a name's molecular structure. Additionally, the semantic representation is used to classify the chemical compound. Such results are collected in our testsuite as can be seen in appendix B.2.1.

We decided in favour of a theoretical basis and thus integrated rules systematically starting with one selected nomenclature system. An alternative approach is to investigate biochemical text sources and cover the most frequent phenomena first. Our procedure is meant to start with a systematic approach whose coverage can later be adapted according to frequency data.

After testing and evaluating how well the approach works, semi-automatic extending will be necessary. For the time being, only prevalent nomenclature rules from IUPAC Commission on Nomenclature of Organic Chemistry (1993) are implemented – later, also names which were not formed according to this special nomenclature or, e. g., morpho-syntactic variations should be analysed. Totals formula, e. g. `H2O` or even $\mathtt{H_2O}$, will not be covered.

We conduct a linguistic analysis because the entities and properties of organic compound names are expressed by their morphemes. The following are some of these features which we adopt also in our representation language (with a different syntactic structure, e. g. `cyclo([...],ane(5*C))`) for the description of compounds (cf. Gerstenberger, 2001, chap. 4):

(16) a. (groups of) atoms, e. g. `C`, `OH`

[45]The tool is a prototype; it does not claim to be exhaustive.

b. bonding types, e. g. single bonds (`ane([...])`)
c. structural properties of the chain, e. g. cyclic (`cyclo([...])`)

As opposed to other approaches, e. g. which go from SMILES strings over a graph structure to a classification (viz e. g. Wittig et al., 2004), we also chose this linguistic approach because a SMILES string (as opposed to a name) is always bound to a certain structure. To catch ambiguous cases as well, we thus consider the name directly and conduct a compositional analysis (see section 2.1).

For systematic names, we provide a morpho-semantic grammar and lexicon with currently about 80 rules and 450 lexicon entries; trivial and class names are treated by lexicon lookup.

We chose Prolog as programming language because the DCG formalism is ideal for the symbolic analysis of linguistic entities (see section 2.1). Our semantic representation with its predicate/functor-argument terms is also suited for further processing with Prolog. We implemented our version in 'Sicstus Prolog' (see Swedish Institute of Computer Science, 2001) and derived another one to work with 'SWI Prolog' (see Free Software Foundation, 1987).

The source code of the system is kept in separate files as listed in table 2.

file	contents
inout.pl	input and output routine
compd_lex.pl	lexicon entries
compd.pl	common root rules
compd_common.pl	common grammar rules and additional common predicates
parent_nonsugar.pl	non-sugar parent rules
parent_sugar.pl	sugar parent rules
smiles.pl	predicates generating the SMILES strings
classes.pl	predicates generating the class lists

Table 2: Prolog files with their corresponding contents description

Parsing Approach

For parsing the chemical compound names we used a symbolic as opposed to a statistical approach (see, e. g. Appelt and Israel, 1999). A statistical procedure depends on training data for the parser's learning process, i. e. on a collection of names and their correct and annotated analyses. Such data is not available in sufficient amount for a machine learning approach. Therefore, to use a statistical parser for our work, it would have been necessary to choose a representative set of names and annotate them with information on their structure. Even then, the disadvantage of a statistical parser remains: It works best with data which is similar to that of the training corpus; new names and also rarely occuring phenomena cannot be analysed reliably. Using a symbolic approach, the parser can be more easily and intuitively extended by adding and modifying rules. However, the possible drawbacks of a symbolic approach have to be considered, namely regarding ambiguities and robustness. First, some ambiguities are intentionally contained in the names themselves (semantic ambiguities, as in `pentadecene` (figure 10 on page 30)) and hence must not be resolved. Furthermore, as we use the semantic outcome (not the syntactic analysis) as a result to be evaluated, most structurally ambiguous name structures pose no difficulties for our system. Thus, syntactic ambiguities are partly resolved in our system by semantics construction yielding identical semantic representations, In addition, a successful SMILES string generation and classification are additional constraints to be fullfilled, which can prevent syntactic ambiguities from emerging. Also without considering semantics or the additional modules, there did not occur many structurally ambiguous analyses until now, probably because the rules are simply not exhaustive. Second, concerning robustness, our system currently fails to analyse incorrect names and names including unknown morphemes, i. e. which are not in the system's lexicon. This drawback is to be remedied with, on the one hand, systematic lexicon enrichment and, on the other hand, with more elaborate methods as described in section 5.

Generalisation

It is important to note that the grammar rules of our system are not meant to be used for generating names, but only for analysing them. The grammar rejects names with an 'incorrect' structure (i. e. not corresponding to the implemented nomenclature principles), but not all of them. So, if it was used for generating instead of parsing, that would result in overgeneration: Not only the valid names of the modeled language would be generated, but also void ones. Because our system is built for analysing names, the term overanalysis applies here, which describes the succeeding analysis of incorrect or non-existent names. The semantics construction and the SMILES string generation can be seen as additional constraints to prevent void names from being analysed. By focusing on the correct parsing results and not considering generation problems, the nomenclature principles can be better generalised (with less restrictive rules covering more alternative cases)[46], so that the number of grammar rules is small. However, this means that we deliberately do not consider a few names that might also match a rule and occur in literature. As for all symbolic (vs. statistical) parsing approaches, where rules are hand-coded, it is an ambitious task to cover all phenomena occurring without risking overanalysis; therefore, a compromise had to be accepted in this respect.

Architecture

The general structure of our system illustrating its modules can be seen in figure 7. The process sequence is as follows: A chemical compound name is morpho-semantically deconstructed by the parser module, resulting in a semantic representation term. This term is, on the one hand, processed by the SMILES string generator yielding a SMILES string. On the other hand, our classifier calculates the superclasses to which the name belongs.

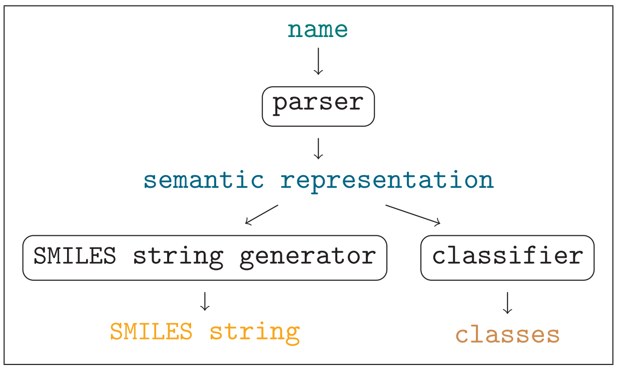


Figure 7: System architecture

Program Flow

There are two ways of using the system, with each input-output variant having its own call predicate. The call `preprocess` is for the first case, where the input can be a single name. This query yields a complete call for a given name to commit it again to Prolog (an example is shown in (17)). The analysis output is displayed at the terminal.

(17)
```
org_compd(Syn,Sem,
          [7,-,h,y,d,r,o,x,y,h,e,p,t,a,n,-,2,-,o,n,e],[]),
```

[46]Generally, the purpose of rules is to reach a generalisation as opposed to writing a separate rule for each case.

```
    pp(Syn), beta_reduce(Sem,RedSem),
    smile(RedSem), classes(RedSem,ClassList).
```

As another possibility, a set of names can be stored in a file (named `testsuite.txt`, one name per line) to serve as input to the system. This file may include comment lines beginning with a `%` sign and empty lines, which are both ignored. In this case (called with `process`), which we used for processing our testsuite, the output is both displayed at the terminal and stored in a result file (named `testsuite.out`).

Each input name is converted into a character list in Prolog format, e. g. `hexose` becomes `[h,e,x,o,s,e]`. This conversion is done using a 'Perl' script[47](see Perl Foundation, 2004) which is called from the current Prolog process and which writes the converted name into a temporary file. This file, in turn, is read by Prolog to be processed further.

The Perl script not only splits the input name into single characters, but also ensures the proper treatment of special characters, i. e. parentheses, commas, hyphens, apostrophes and blanks. These get surrounded by single quotes, so that they can be treated as Prolog atoms with no special meaning for the program. Capital letters are converted to lower case for the same reason, which also means that it does not matter if (parts of) names are given in lower or upper case letters. A period at the end of an input name, which is usually required for Prolog queries, may be omitted; if given it is ignored by the preprocessor. Several successive space characters are ignored as well if they occur at the end of, not within, a name. Special cases that have to be considered when giving a name as input to the system are, for example, the symbol α, which has to be replaced by the text string `alpha`.

After preprocessing the input name, Prolog is directed to try and find all possible parsing solutions. The semantics of the name are generated simultaneously in the parsing process by means of the λ-calculus, which yields a complex semantic representation. It is β-reduced in the next step, i. e. the λ-expressions are evaluated, to obtain a simplified semantic representation. Duplicates of semantic representations in the set of alternative parsing solutions are sorted out because they would result in identical further processing. The program then attempts to generate the SMILES string from the semantic representation of the input name. This is possible if the representation term is valid and the name is neither a non-sugar term nor a class name, as these cases are not covered. The semantic representation then also serves as basis for the generation of the list of classes for non-sugar names, which are calculated in the next step.

[47]We used Perl for the conversion because it was more intuitive to implement and less code was necessary.

The results, i. e. a syntax and a semantic analysis as well as a SMILES string and a class list, if available, are then printed as output.[48]

Analysis Examples

A slightly simplified analysis for a non-sugar name (`7-hydroxyheptan-2-one`, depicted in figure 8) is shown in figure 9. Simplified relevant grammar rules for this non-sugar example with annotated semantics can be seen in (18) to (21).

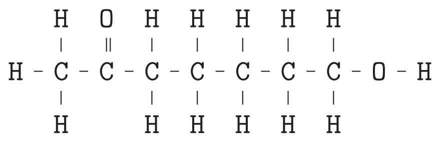


Figure 8: Molecular structure of `7-hydroxyheptan-2-one`

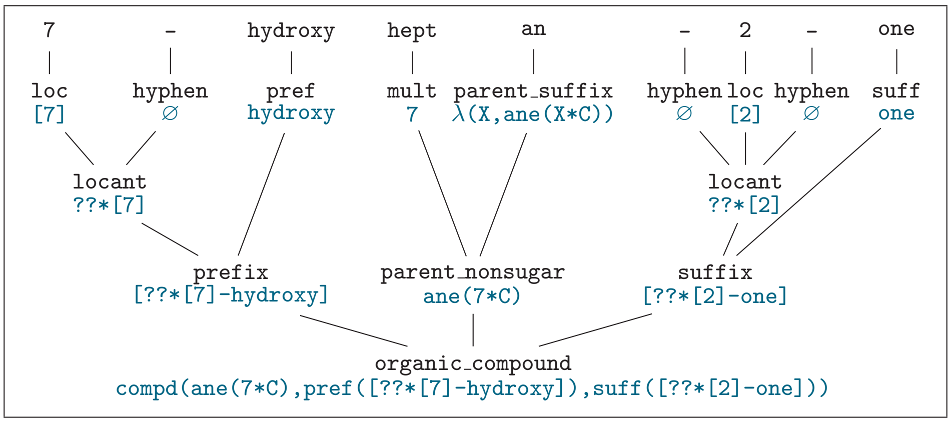


Figure 9: Non-sugar example analysis

[48]The testsuite processing yields a reduced output omitting the syntactic analysis.

(18) excerpt from `compd.pl`:

a.
```
organic_compound(compd(Sem_parent_nonsugar,
                       pref(Sem_prefix),suff(Sem_suffix)))
  -->
     prefix(Sem_prefix),
     parent_nonsugar(Sem_parent_nonsugar),
     suffix(Sem_suffix).
```
b.
```
prefix(Sem_locant-Sem_pref)
  -->
     locant(Sem_locant),
     pref(Sem_pref).
```
c.
```
suffix(Sem_locant-Sem_suff)
  -->
     locant(Sem_locant),
     suff(Sem_suff).
```

(19) excerpt from `parent_nonsugar.pl`:
```
parent_nonsugarSem_parent_suffix@Sem_mult)
-->
   mult(Sem_mult),
   parent_suffix(Sem_parent_suffix).
```

(20) excerpt from `compd_common.pl`:

a.
```
locant(??*Sem_loc)
  -->
     loc(Sem_loc),
     hyphen(_Sem_hyphen).
```
b.
```
locant(??*Sem_loc)
  -->
     hyphen(_Sem_hyphen),
     loc(Sem_loc),
     hyphen(_Sem_hyphen).
```

(21) excerpt from `compd_lex.pl`:

a. `lex([7],loc,7).`
b. `lex([2],loc,2).`
c. `hyph(hyph(-),affix_sep) --> [-].`
d. `lex([h,y,d,r,o,x,y],pref,hydroxy).`
e. `lex([h,e,p,t],mult,7).`
f. `lex([a,n],parent_suffix,`λ`(X,ane(X*C))).`
g. `lex([o,n,e],suff,one).`

For ambiguous compound names, the parser yields several analyses as shown in figure 10 on the next page. For `pentadecene`, we get on the one hand an analysis where the main molecular structure has five C atoms and on the other hand an analysis where it has ten C atoms, because locants specifications for the clarification of their number are missing. The impossible analysis (the bottom one) has to be eliminated either by a consistency check as to how many skeleton atoms are available to be referred to by prefix locants or during the SMILES string generation.

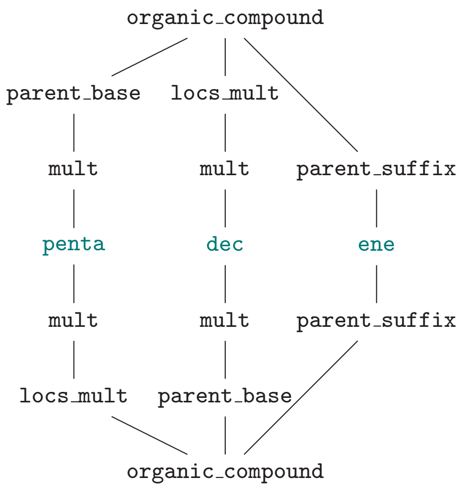


Figure 10: Ambiguous name with two analyses

4.2 Parser

The parser of our system depends on morpho-syntactic DCG rules written in the Prolog programming language. They make it possible to analyse organic compound names and to generate their corresponding semantic representations, which is described below.

The analysis is obtained from the names exactly as given to the system. This is in contrast to a system as developed by Gerstenberger (2001) which assumes a preprocessing module that splits the names into appropriate morphemes before processing them by means of grammar rules.

The Grammar

Here we describe how the grammar of our system is structured. Most DCG rules for a specific syntactic category have alternative rules, which appear in the grammar in groups. The comment lines just above each rule contain information about the nomenclature principle implemented and/or give sample names to be processed with that rule. The topmost syntactic category (the root of the grammar) is `org_compd` which relates to a complete organic compound name. It may also be embedded, which requires a recursive rule.

DCG Rule Example To describe the DCG notation format, a sample rule of our grammar is shown in (22). It declares that `ring_stem` can be substituted by `locs_mult` and `ring_triv`, in the given order from left to right. The symbols `ring_stem` and `locs_mult` are non-terminal symbols, whereas `ring_triv` is a terminal symbol which will be substituted by a lexicon entry in a next step. In the name `hexopyranose` the part `pyranose` is parseable by means of this rule.

(22)
```
ring_stem(ring_stem(Syn_locs_mult,Syn_ring_triv),
          Sem_ring_triv@Sem_locs_mult)
-->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    ring_triv(Syn_ring_triv,Sem_ring_triv).
```

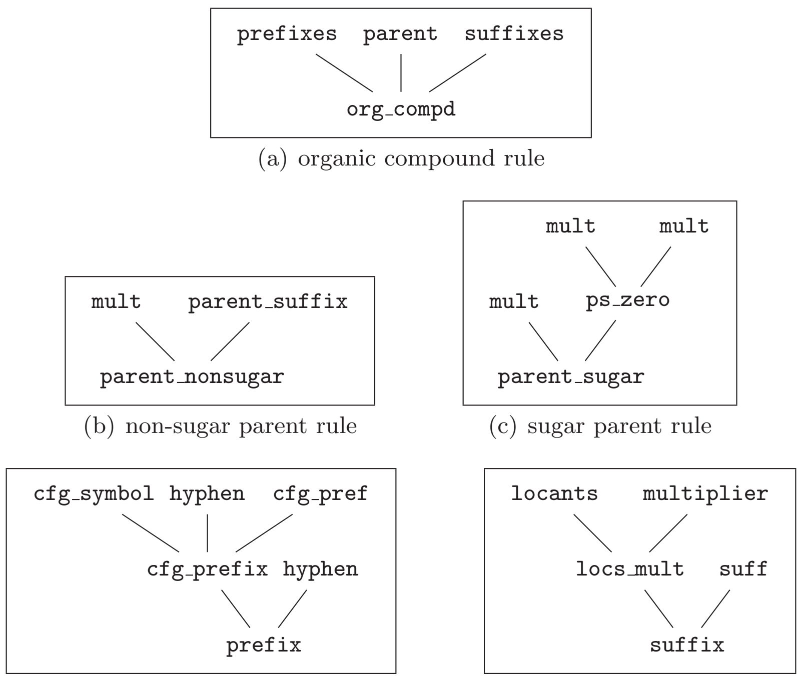
Each symbol's arguments are its syntactic and semantic annotation. The first argument describes the syntactic structure which is for `ring_stem` constructed from the syntactic category `ring_stem`, comprising the syntactic substructures of `locs_mult` and `ring_triv`. They are kept in the variables `Syn_locs_mult` and `Syn_ring_triv`, respectively. Each symbol's second argument constitutes its semantic representation. The rule shown here determines that the semantic representation of `ring_stem` is an application of the semantic expression of `ring_triv` to the semantics of `locs_mult`.

Grammar Rules The root rule with the head `org_compd` determines what constitutes an organic compound name, viz at least a parent name, possibly supplemented by one or more prefixes, or suffixes, or both (see figure 11(a)). A parent name can either be a sugar or a non-sugar name.

Non-sugar parents roughly are composed of a parent base (e. g. an element morpheme or a multiplier) and either a saturated or an unsaturated parent suffix. For unsaturated parents, locants and a multiplier are allowed to specify the location of the unsaturation. See figure 11(b) for an example.

Sugar parents can consist of locants and multipliers combined with ring stems or one or two stem suffixes with possibly more multipliers and locants attached (see figure 11(c) for a simple example). Configurational symbols and prefixes, which in the strict chemical sense also belong to the parent structure, are dealt with in the prefix rules, as our categories are rather linguistically motivated.

The rule for prefixes prescribes their constituents to be one or more prefixes with or without a succeeding hyphen. A prefix can be either a configurational



(a) organic compound rule

(b) non-sugar parent rule

(c) sugar parent rule

(d) prefix rule

(e) suffix rule

Figure 11: Grammar excerpts

one, a simple one (`pref_zero`[49]) with or without locants and/or a multiplier, or a double prefix consisting of locants and/or a multiplier, an element (e. g. `C` or `O`) and a simple prefix (see figure 11(d)). In addition, as noted above, this is the rule for nested compounds (possibly with locants and/or a multiplier), with or without parentheses enclosing the nested compound, for example as shown in figure 12 on page 39. Suffixes can be added either with or without a preceding hyphen, one or more of them, with or without preciding locants and/or a multiplier A simplified example can be seen in figure 11(e). Resulting from nomenclature rules such as in (23), multi-word expressions such as '`hexanoic acid`' occur.

(23) R-5. 7.1.1
Carboxylic acid groups [. . .] are denoted by adding the suffix "-oic acid" or "-dioic acid" [. . .] to the name of the acyclic hydrocarbon [. . .].

They have an adjective-noun structure, and also other combinations with such emerged adjectives can be encountered in texts (e. g. `hexanoic extract`[50]). For automatic term recognition in textual data, an analysis for the single parts of such multi-word expressions (on a syntactic level) is therefore necessary. Our current solution allows `-oic` as an adjective suffix together with a blank and a class name (instead of allowing a fix suffix `-oic_acid`). This complex suffix is treated as a 'functional groups (adjective) suffix' which can attach to a parent name in a kind of syntax rule. Although a separate analysis of `hexanoic` is not provided thereby (this would have gone beyond the scope of this work), it creates a basis for doing that.

The complete set of rules can be seen in appendix A.2.

Syntactic Structure In our grammar, the symbols' names acting as syntactic categories are motivated by linguistic, syntactic-semantic interests. Nevertheless, terms from the biochemistry vocabulary are also used as a basis for naming the categories. However, a category with a name reminding of a biochemical term does not necessarily refer to exactly the same part of a molecule expressed by that biochemical term. We chose the syntactic category names of our grammar for the practical reason to easily and uniformly construct the semantics of the names. The syntactic structure of an analysed name was useful when developing the grammar rules to check the parser's performance, but is currently not used for anything else.

[49]The term `pref_zero` is motivated by syntactic categories such as N^0 for a basic level noun.
[50]http://www.swsbm.com/Abstracts/Crataegus-AB.txt

The syntactic structure is represented in a functor-arguments format in which the functor is named after the head symbol's syntactic category and comprises the syntactic categories of the rule body's symbols. Thereby, the order of the symbols in the grammar rule is retained. The term constructed in this manner is the representation of the currently processed grammar rule and thus is used as the syntactic annotation of the rule head. For the DCG example rule (22) on page 31 the corresponding syntactic structure format is shown in (24).

The value of the variable `Syn_locs_mult` will be itself a syntactic structure because it relates to the non-terminal symbol `locs_mult`. The value of `Syn_-ring_triv` will be a character list of the respective morpheme which the related terminal symbol `ring_triv` yields the lexicon entry.

(24) `ring_stem(locs_mult(Syn_locs_mult),ring_triv(Syn_ring_triv))`

Semantic Representation The semantic representations are constructed in a different, more complex manner than the syntactic structures. Working with the λ-calculus with the aim to get a consistent semantic representation, the morphemes' semantic annotations have to be combined in the correct order. This is achieved in the grammar rules by means of the operator '`@`' put between a λ-expression and each of the appropriate arguments to be used in the λ-operation (as can be seen in (22) on page 31). It represents the application of a λ-expression to the expression's argument and combines all the semantic representations in the parsing process of a name. The resulting complex semantic representation has then to be β-reduced, i. e. the λ-expressions have to be resolved by applying the λ-operation. The resulting semantic representation describes the molecules of a compound name – for our needs – in a kind of operator-arguments logic. Note that the grammar rules as well as the morphemes' lexicon entries may contribute a λ-expression. A simplified example demonstrating the beta-reduction of the semantic representation for `pent-2-ulose` is shown in (25).

(25) a. complex λ-expression:

```
lam(X,ulose(??*[2],X))@(5*C)
```

b. β-reduced semantic representation:

```
ulose(??*[2],5*C)
```

We defined the language of the semantic representation of organic compound names in appendix B.1.1. It is described there in form of an Extended Backus-Naur Form[51] (EBNF). Generally, it is a term composed of functors and their

[51]We defined it according to the standard ISO/IEC 14977:1996(E)

arguments. Each functor is an operator acting on a structure contained in one of its arguments. Detailed specifications needed for the operation are in the functor's other arguments. The semantic representation of `hex-2,3-diene` (shown in (26)) describes a skeleton structure consisting of six carbon atoms (`6*C`), on which the functor `ane` (which determines that single bonds are to be constructed) operates on. This semantic term is embedded as argument to the functor `ene` which represents the creation of double bonds. This functor has a supplementary argument to specify the locants where the double bonds should be created (and how many): `2*[2,3]` means that two locants are specified in the corresponding name: `2` and `3`.

(26) `compd(ene(2*[2,3],ane(6*C)),pref([]),suff([]))`

The outermost functor of our semantic representation (`compd`) is used to apply prefix and suffix operations on the structure represented in its first argument (the representation of the parent name). This functor and its arguments represent the whole structure denoted by the name that was given as input to the parser. Terms with the functor `compd` can also be nested to describe embedded, complete organic compound names, e. g. occurring as prefix to a parent structure.

The format of a term consisting of the functor `compd` and its arguments is shown in (27).

(27) a. `compd(Parent, pref(Prefixes_List), suff(Suffixes_List))`
b. `Prefixes_List:`
`[Prefix_Spec_1, Prefix_Spec_2, Prefix_Spec_3,...]`
c. `Prefix_Spec_`*n*`:`
`Mult*Locant_List-Prefix` (e. g. `??*[2,3]-deoxy`)
`or`
`Mult*Locant_List-Element-Prefix` (e. g. `1*[3]-C-compd(...)`)

The functors `pref` and `suff` are special; their respective argument is a list of operators that are to be applied to `Parent`, i. e. they are functors without arguments and the structure they operate on is not directly associated. Each of these operators is preceded by a multiplier, a list of locants, and a hyphen, which denote the details for the operation, e. g. `2*[3,4]-deoxy`. Note that `Prefix` can be as well an embedded `compd` term.

Table 3 describes the functions of some of the operators contained in `Parent` and in `Prefixes_List` and `Suffixes_List`.

operator	function
ose	creates a carbonyl group (C=O) at first and/or last locant
ulose	creates a carbonyl group at the specified locants
anose	creates a ring connection between pairwise specified locants

Table 3: Examples of operational functors occurring in the semantic representations and their associated functions

The Lexicon

Our lexicon consists of entries for valid morphemes to form a name with, separated into sets by their corresponding syntactic categories. A lexicon entry comprises a morpheme represented as a Prolog list of characters, a syntactic category and a semantic representation. These function as arguments to the functor `lex`, altogether being a so-called Prolog fact. The format is shown in (28a). Example (28b) shows the entry for the multiplier `penta-`.

(28) a. `lex(Char_List,Syn_Cat,Sem).`
b. `lex([p,e,n,t,a],mult,5).`

For some morphemes we created more than one lexicon entry, since they are used with differing syntactic categories (e. g. `erythro-` is used as configurational prefix and sugar parent trivial prefix). Moreover, as we do not deal with linking morphemes (which bear no semantic meaning) and deletion of vowels, we wrote double entries for morphemes that can also appear followed by a linking morpheme. For example, we provide one entry for `erythro-` and one for `erythr-`, both having exactly the same arguments except for the list of characters (representing the morpheme string).

(29) a. `lex([e,r,y,t,h,r,o],cfg_pref,erythro).`
b. `lex([e,r,y,t,h,r,o],ps_triv_pref,erythrose).`
c. `lex([e,r,y,t,h,r],ps_triv_pref,erythrose).`

The semantic annotation in a lexicon entry is either the morpheme itself, the appropriate chemical symbol, or the appropriate number (in case of locants and multipliers). A few morphemes require a complex λ-expression, e. g. as shown in (30).

(30) `lex([u,l,o,s,e],stem_suff,lam(X,lam(Y,ulose(X,Y)))).`

The λ-expression for the morpheme `ulose` defines the operator `ulose` as a functor taking two arguments (the λ-operator is represented by the functor `lam` in our Prolog implementation).

The special character morphemes (comma, hyphen, etc.) also have annotations which are at the argument position of the semantic annotations, but only for consistency reasons – they are not used in the grammar. A special lexicon entry clause is not necessary here, as these are single characters to be processed, thus we used the usual lexicon rule. Note that special characters have to be quoted in Prolog (to be processed as atoms) as is shown in the lexicon rule for the left parenthesis in example (31b).

(31) a. `hyph(hyph(-),affix_sep) --> [-].`
b. `leftparenthesis(leftparenthesis('('),'(') --> ['('].`

Table 4 shows some of the syntactic categories we used in our lexicon; their corresponding lexicon entries can be seen in appendix A.2.1.

syntactic category	description
ps_class	class names for sugar parent structures
pns_triv	trivial names for nonsugar parent structures
stem_suff	parent structure stem suffixes
loc	locants
pref_elem	atomic element prefixes
cfg_symb_anom	anomeric configurational symbols
hyph	hyphen

Table 4: Sample syntactic lexicon categories and their desription

Efficiency Considerations

The system we present here is a prototype – other implementations, including more sophisticated parsing methods, other programming languages, etc., may ameliorate its performance. Nevertheless, we took care of efficient performance, since it is not only a crucial aspect for the eventual application, but it was already helpful during the system development.[52] We checked the improved efficiency by use of the built-in predicate `statistics` which showed great impact of the methods used and described in this subsection.

[52]For example, looking for all alternative parsing possibilities for a name (e. g. for checking ambiguous analyses) may prolongs the parsing process essentially, as opposed to taking the first parsing solution found.

Cut Symbol In Prolog, the cut symbol '`!`' is used in a clause to prevent backtracking, i. e. the search for alternative clauses in case a clause failed. This applies analogously for DCG rules (as the Prolog compiler translates them into clauses). Our grammar contains several non-terminal symbols for which there are alternative rules. The cut symbol can be used there like any other (non-)terminal symbol. It is inserted at a point in the rule where no other solutions for its head symbol should be found (i. e. further possible solutions are 'cut'). Thus, unnecessary expanding of fruitless rules is prevented, which results in faster processing. Cut symbols have to be inserted carefully not to accidentally exclude wanted (correct) solutions. In addition, the order of alternative rules is important when using cut symbols.

Lexicon Entry Clauses The DCG operator '`-->`' is usually not only used in grammar rules, but also for lexicon entries (i. e. for expanding non-terminal symbols). However, we preferred writing a special Prolog clause for terminal symbols because a lexicon rule using the DCG operator '`-->`' would process the lexicon entry character by character, as it constitutes a list of characters in Prolog format. As opposed to this method, the clause we wrote for terminal symbols to be substituted by their corresponding lexicon entries results in a faster process: It checks if the complete lexicon entry matches the remaining part of the name currently being processed.

Covington (1989) also emphasises the efficiency improvement possible by using the "first argument indexing of Prolog". This is achieved by giving Prolog facts (in our case the lexicon entries) the most diverse argument first. We followed his recommendations by choosing the character list of the respective morpheme as first argument to the lexicon entry functor `lex`.

Example (32a) shows a sample lexicon rule usually used. The corresponding lexicon entry clause we used instead is depicted in (32b), and (32c) is an appropriate lexicon entry of our system.

(32) a.
```
cfg_pref(cfg_pref(Entry),Sem)
-->
    {lex(Entry,cfg_pref,Sem)},
    Entry.
```
b.
```
cfg_pref(cfg_pref(Lex),Sem,All,Rest)
:-
    lex(Lex,cfg_pref,Sem),
    append(Lex,Rest,All).
```
c.
```
lex([r,i,b,o],cfg_pref,ribo).
```

Embedded Compound Names Our parser is able to analyse certain types of embedded compound names, i. e. names which represent complete compounds themselves, but that are part of other compound names. There are several kinds of embeddings, some of which are listed in (33).

(33) a. `1,2-di`*`methyl`*`-hexane`
b. `2-C-(`*`Hydroxymethyl`*`)-D-ribose`
c. *`Methyl`* `3-deoxy-D-threo-pentonate`

The compound name `methyl` in (33a) is embedded as a prefix of `hexane`, preceded by locants and a multiplier. Example (33b) shows the embedding of the non-sugar name `Hydroxymethyl` in a sugar name (`D-ribose`). According to the nomenclature rules, names can also be combined as shown in (33c), where we regard `Methyl` as the embedded name.

Figure 12 shows a nested syntactic structure illustrating our grammar rules, where an organic compound structure can again be integrated in the prefix of an organic compound.

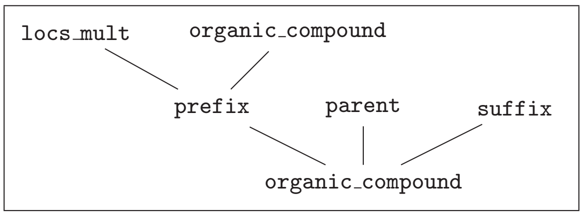


Figure 12: Grammar fragment including a nested organic compound

The disadvantage of embedding names is that it takes the parser longer to decide if an input name is valid or not, corresponding to the rules. This is because it multiplies the number of rules which the parser has to try when processing a rule including an embedded compound. For that reason, we put the rules for embedded compounds at the end of the respective set of alternative grammar rules. But, as we want the parser to look for all possible solutions, it will still try to process these rules – for names both with and without an embedded compound name – even if other solutions have already been found. On the other hand, when processing a name containing an embedded compound name, all the alternative rules have to be tried first before finding at last the correct rule. Here, working with cuts, put carefully at the correct positions in the rule, improves the performance essentially.

Left-Recursiveness Embedded compounds require left-recursive grammar rules (for a definition see section 2.1). As we work with a top-down, left-to-right parser, grammar rules are processed by expanding their left symbols first, beginning at the rule for the top symbol. Additionally, alternative rules for a non-terminal symbol to be expanded are processed in Prolog in the given order as written in the files. Processing left-recursive rules may thus lead to endless loops if the parser does not succeed in expanding the first, non-terminal symbol of a left-recursive rule but by use of the same rule again. As no terminal symbol is expanded and hence the input string is not reduced, the parser will continue expanding the left-recursive rule by itself.

If the left-recursive rule is not the last of all alternative rules, the parser will never reach the following rules. Instead, it will continuously try to expand the left-recursive rule by itself.

Although correct input may lead to such endless loops, primarily, non-parseable expressions (i. e. incorrect names, which are not intended to be parsed) will do so when using left-recursive rules.

To circumvent the difficulty with left-recursiveness in general, a different parsing algorithm can be used – one that is also more efficient in processing embedded compounds. For example, Voss (2004) describes an Earley chart parsing algorithm implemented in Prolog.

4.3 SMILES String Generator

A SMILES string is a structural notation of a molecule which sequentially lists the main chain elements with their properties and branches. In our system, the basis for the generation of the SMILES string is the semantic representation of the compound name, which describes the operations to be applied to nested semantic structures. Thus, we decided to generate the SMILES string via a pre-representation which transforms the operational description of the semantic representation notation into a description of chain elements with their respective properties and branches. Its format is basically a functor-arguments expression containing lists, which can be easily and efficiently processed in Prolog. From the pre-representation, the SMILES string can be generated more intuitively. In case of underspecified names, it is constructed as completely as possible and serves as an argument to an underspecification expression, i. e. a mixed representation instead of a valid SMILES string is the output.

General Procedure

The construction of the pre-representation from the semantic representation is an inside-out process: The SMILES string generator traverses the semantic representation by matching a functor-argument-structure pattern, in which the argument itself is another functor-argument-structure. This is done recursively until there is a simple structure embedded as an argument, i. e. the innermost structure is found.

This structure serves for constructing the main chain of the molecule's parent structure. In case the innermost structure is a trivial name, a look-up in the lexicon is made. After that, the superordinate functor which operates on this structure is processed. The consistency of its first argument is checked to see if the multiplier-locant combination is valid. Only in case it is valid, the still incomplete pre-representation is modified, depending on the respective operation associated with this functor. As for sugars, the functors `ose` and `ulose` each create a carbonyl group (i. e. a carbon-oxygen double bond: `C=O`); `anose` builds a ring connection.

The respective superordinate functors are used repeatedly to build a more and more complete parent structure[53] pre-representation, until the outermost functor is processed. At first, default operationsare then carried out on the parent structure pre-representation.

After having constructed the pre-representation of the compound main chain, the prefixes (including configurational prefixes) of the compound name are processed.[54] They are bundled in a list of Prolog expressions, as described in section 4.2.

Embedded compounds are processed analogously when they are encountered, e. g. in the prefixes processing step, and will be embedded in the pre-representation like any other element (or molecule group).

Innermost Semantic Structure

The innermost structure of the semantic representation is either a trivial name or an expression of the form `Length*Element`, e. g. '`5*C`' representing a skeleton structure consisting of five carbon atoms.

Trivial Name If the innermost structure is a trivial name representation (e. g. `triv_name(ribose)` for `ribose`), the semantic representation (in the form produced by the system) for the corresponding systematic name is looked up in the lexicon for trivial names and the SMILES string is generated from it.[55]

[53]To be exact, it is not the pre-representation of the biochemical term parent structure, but of the structure which is comprised by the syntactic category `parent` in our grammar.

[54]Currently, no sugar name is analysed as having suffixes. Therefore, these are not processed in the SMILES string generator.

[55]This method can also be used for abbreviations of names.

Example (34) shows a lexicon entry, which contains the trivial name, the corresponding systematic name, and the corresponding semantic representation.

(34)
```
lex_triv(ribose,
    'D-ribo-pentose',
    compd(ose(??*[??],5*C),pref([cfg([D-ribo])]),suff([]))).
```

The general usage of trivial names is to be considered when associating their systematic names in this lexicon. For example, the trivial name `ribose` is often used for the cyclic form, β-`D-ribofuranose`. We dealt with this phenomenon of ambiguous trivial names by adding a second lexicon entry for the cyclic form. The order of the alternative lexicon entries is important as it decides which systematic name is chosen first (which is crucial if alternatives are 'cut' in the process of generating the SMILES string).

Skeleton Structure In the second case, the innermost structure is a specification of the form `Length*Element` representing a chain of elements. For example, the parent structure of `pentose` consists of five carbon atoms. This skeleton is represented as `5*C`, being the innermost structure of its semantic representation shown in (35).

(35)
```
compd(ose(??*[??],5*C),pref([]),suff([]))
```

The skeleton structure of the `SMILES` pre-representation is constructed by creating a list in Prolog format, which consists of list elements. Each of these elements describes a main chain element of the molecular structure and has the form as generally expressed in (36). The number of such list elements is determined by the value of `Length`, which would be `5` in our example.

(36)
```
chain_el(Element,Index,Branches_List,Features_List)
```

The description of a main chain element includes its properties and side branches, being arguments to the functor `chain_el`. The argument at the position of `Element` determines the chemical element which is inherited from the semantic representation; in our example this would be `C`. The argument `Index` is the locant number of the respective main chain element. As the number of chain elements is specified by `Length`, this may be a number from 1 to 5, in our example. For the arguments `Branches_List` and `Features_List`, empty lists (noted as '`[]`') are created initially, because only the skeleton structure are constructed in this step. They will be modified in a later processing step.

After creating the skeleton of the main chain representation, a supplementary list element is added at the beginning of the list, which we use to represent a possible set of underspecifications. This additional list element is composed of the functor `uspecs` and a preliminarily empty list as its argument, which can be filled up later.

The complete pre-representation skeleton is illustrated in (37) in a simplified form, with index numbers from 1 to the specified length of the main chain.

(37)
```
[uspecs([]),
 chain_el(Element,1,[],[]),
 chain_el(Element,2,[],[]),
 ⋮
 chain_el(Element,Length,[],[])]
```

Consistency Check

On each expression of the form `Multiplier*Locants_List` a consistency check is carried out. Such expressions appear as arguments of functors (e. g. as in `ene(2*[2,3],...)`), and they are prefixed to operators (`2*[1,3]-deoxy`) in the list of prefixes or suffixes in the semantic representation. If either `Multiplier` or `Locants_List` is specified, or if neither of the two values is specified, the consistency check succeeds.[56] This means that no inconsistency could be detected, although maybe resolvable or unresolvable underspecification is on hand.

Otherwise, the success of the consistency check depends on the operator that is associated with the specification `Multiplier*Locants_List` or on the functor comprising this specification – thus, these are considered in the consistency check, too. In general, the number of locants in the locant list has to match the number the multiplier presupposes. Any other case shows an inconsistency in the naming of the respective compound.

However, if the operator comprising the specification is `anose` (as in shown in (38)), the consistency check is special, as `anose` describes a ring connection. In this case there have to be twice as much locants as the number given by the multiplier number, or if no multiplier is specified, the number of locants has to be divisible by 2. This presupposition has to be justified since each ring connection – unless underspecified – has to be specified by a pair of locants.

(38) `compd(anose(??*[2,5],5,...)`
(e. g. `in hexos-2-ulo-`*`2,5-furanose`*)

The principles mentioned cover the following cases for non-ring specifications:

[56] Note that a specification which is required by our semantic representation language but which is missing in a compound name, is depicted as '`??`'.

(i) `3*[1,2,5]`
(multiplier and locants are specified;
the multiplier matches the number of locants)

(ii) `3*[??]`
(only the multiplier is specified[57])

(iii) `??*[1,2,5,6]`
(only the list of locants is specified)

For ring specifications (appearing as an argument to the functor `anose`), the following cases are covered:

(iv) `2*[1,5,3,6]`
(multiplier and locants are specified; the locants' quantity,
divided by 2, matches the multiplier)

(v) `2*[??]`
(only the multiplier is specified)

(vi) `??*[1,5,3,6]`
(only locants are specified; their number is divisible by 2)

Fully specified examples as in (i) and (iv) are the simplest and most definite kinds of specifications to be processed further. The other specifications shown are also likely to appear and are therefore also considered. Examples as in (ii) and (iii) may occur due to deliberate underspecification of a compound name or insufficient knowledge of the nomenclature rules. Examples as in (v) and (vi) are allowed specifications according to the nomenclature rules.

Note that in the case of full underspecification as shown in (vii), the consistency check also succeeds, for both ring and non-ring specifications.

(vii) `??*[??]`
(neither multiplier nor locants are specified)

Other cases are not allowed, i. e. when the number of locants does not match the multiplier number (there are more locants or less locants than presupposed by the multiplier) in non-ring specifications, or when the number of locants is not divisible by 2 or the multiplier number is not equal to the number of locants divided by 2 in ring specifications. These cases will cause the SMILES string generation to fail.

[57]The locants are probably forgotten or underspecified.

Ring Construction

A ring construction is determined in the semantic representation by the `anose` functor, which comprises three arguments: `Multiplier*Locants_List`, `Ring_-Size`, and the structure representation to operate on (i. e. to construct the ring). The multiplier determines the number of (same-sized) rings to be constructed, and the pairs of locants determine which chain elements are to be connected to form a ring. If neither multiplier nor locants are specified, the default supposition is '1' for the multiplier. That is, we preliminary opted for only one ring connection (more exactly, for the first to be possible) in that case of underspecification, although other underspecified expressions (e. g. prefixes) are processed by looking for all possibilities. However, the implementation of this feature is complex and we did not consider it essential for the current system.

The examples shown in (39) represent possible specifications and their corresponding methods for building the appropriate ring connections.

(39) a. `2*[1,4,3,6]`
  ⇾ build ring connection using locant pairs

b. `??*[1,4,3,6]`
  ⇾ build ring connection using locant pairs

c. `2*[??]`
  ⇾ try to build 2 ring connections using ring size

d. `??*[??]`
  ⇾ try to build 1 ring connection using ring size

All other cases are rejected by the consistency check. In the examples (39a) and (39b), the multiplier is not necessary for further processing, because the locants required for building ring connections are specified. In the examples (39c) and (39d), the ring size is needed additionally for building ring connections, because no locants are specified.

Using Locants If locants are specified as in (39a) and (39b), rings will be constructed directly by changing the pre-representation at the appropriate chain elements as described below.

Before that, an additional consistency check of each pair of locants in relation to the ring size is performed to ensure that the locants' distance is matching the ring size[58].Furthermore, the order of each pair of locants is checked and reversed if necessary. For the subsequent, sequential processing of the Prolog pre-representation list, the lower locant should be the first to be processed.

[58]Note that the oxygen atom which is used as connection element has to be taken into account, too, as it is also counted when specifying the ring size.

Besides, a supplementary check is performed to ensure that a carbonyl group is existent at exactly one of the chain elements specified by the pair of locants. This is because the oxygen atom (`O`) of the carbonyl group (`C=O`) is a precondition for the ring connection in a sugar compound. Such a chain element is displayed in our pre-representation as shown in example (40). Only the chain element of the pre-representation bearing the carbonyl group is depicted.

(40) `[..., chain_el(C,Loc1,[[=,O]],Features_List),...]`

Using Ring Size When the multiplier is given but no locants (as seen in (39c)), the number of rings specified by the multiplier will be each constructed only from the specified ring size. This construction method is also used in case neither locants nor multiplier are given (see (39d)).

Again, the precondition for building a ring connection is the existence of a carbonyl group at either of the two chain elements to be connected. The locant of the other ring connection element can be calculated from the locant of the carbonyl group and the specified ring size. In principle, the ring connection element for which the locant is calculated may be either at a higher-numbered locant or at a lower-numbered one.

We decided to build the first[59] possible ring connection of the following attempts: Starting from locant number 1, try and find a carbonyl group in the main chain and calculate the appropriate locant for the other ring connection element. If a ring connection is not possible there (e. g. if the chain is shorter than the calculated locant number or if the corresponding chain element already has a side branch), try with the next carbonyl groups in the chain until the whole chain is processed. If no ring connection could be built, repeat trying to find carbonyl groups starting from locant number 1. This time, calculate lower-numbered locants for elements for a tentative ring connection. If all attempts fail, the output is '`NO SMILES`' instead of a SMILES string.

Modifying the Pre-representation For ring connections that can be constructed at the two ring elements specified or found, appropriate changes of the pre-representation must be made. The ring connection is expressed in SMILES by indexing the two connection elements with the same number and noting this connection index just after the respective element in the SMILES string. In our pre-representation, we insert the appropriate functor `ring_el1` or `ring_el2`. Each functor's only argument is the index number of the connection, which we

[59]An enhancement of the system could be storing all possible ring connections and trying to resolve the resulting ambiguity during further processing.

chose to be the lower-numbered locant of the two ring connection elements, for ease of implementation and unambiguousness.[60]

Additionally, the feature `carbgrp` is added to the feature list of the element where the carbonyl group was attached before building the ring connection. This information will be needed later for processing the configuration specifications.

At the ring connection element with the higher-numbered locant, the expression `ring-O` is added to its previously empty Branches-List.[61] In turn, the oxygen atom of the carbonyl group needed to build a ring connection, is deleted from the branches list of the respective chain element – together with the double binding specification in the same branch list.

Example (41) illustrates the pre-representation specifications for ring connection elements before and after the ring connection is constructed. A pre-representation of a ring connection is always specified as shown in this example, no matter if constructed from locants or from ring size, forward or backward.

(41) ring connection elements
- a. . . . before connection is made:
  - (i) `chain_el(C,1,[[=,O]],[...])`
  - (ii) `chain_el(C,5,[],[...])`
- b. . . . after connection is been made:
  - (i) `chain_el(C,1,[],[ring-el1(1),carbgrp,...])`
  - (ii) `chain_el(C,5,[[ring-O]],[ring-el2(1),...])`

Defaults

Before applying the prefix functors, default operations are used to adapt the skeleton structure of the pre-representation. This is useful at this point, because there are prefixes which need to substitute default atoms of the side branches, which have not been specified in our pre-representation, yet. Depending on which (i. e. sugar or general organic compound) nomenclature is used, different defaults are applied.

As for sugars, each chain element has one oxygen atom attached to it by a single bond, unless it was specified otherwise. Additionally, the remaining valences of each chain element (and oxygen atom attached) are filled up with hydrogen atoms. These are implicit in the SMILES notation and thus need not to be inserted. However, we need to note one hydrogen atom (as a side

[60] However, this can lead to complications if one carbon atom is involved in two ring connections. We did not prepare the system to deal with such a case.

[61] This is to keep the information that this is no usual branch of the carbon atom in the chain, but an oxygen atom involved in a ring connection.

branch) explicitly at each chain element so that it can be used for the consistent specification of the chain element's chirality[62] and for a possible hydrogen substitution operation. The remaining default hydrogen atoms (which are not needed) are omitted. These defaults are applied in our pre-representation as depicted in example (42).

(42) chain element of skeleton structure

a. ... before applying defaults:

`chain_el(C,Index,[],[])`

b. ... after defaults applied:

`chain_el(C,Index,[[[H]],[O]],[])`

Every empty branches list of a chain element is replaced by a branches list representing a hydrogen branch and an oxygen branch attached to the carbon chain element. For representing two side branches, the two elements are themselves noted as lists (enclosed in brackets). The atoms are noted by their element symbol – the hydrogen atom has to be surrounded by brackets as this is prescribed by the SMILES notation principles (see section 2.2).

When applying the defaults, chain elements bearing the feature `ring_el1` or `ring_el2` are ignored. The chain element specification containing `ring_el1` has an empty branches list, which should be kept empty as this is a ring connection element bearing no side branches. As for `ring_el2`, the branches list is not empty (it comprises branch with the ring oxygen atom `ring-O`), and would thus be ignored anyway.

Configuration Specifications

Configurational properties of a compound are determined in its name by prefixing an expression to the parent name. Formally, this expression is part of the description for the parent structure, consisting of an optional anomeric configurational symbol, a configurational symbol, and a configurational prefix. These may appear in the combinations shown in example (43), always in this prescribed order. Several of those expressions may appear successively. They have to be always attached directly in front of the parent name, which is why they are are called nondetachable prefixes.

(43) a. `D`
(in front of trivial names, e. g.
`D-Ribose`)

[62]If the order of the branches is the same for all chain elements, a chirality symbol assigned will always result in the same predictable chirality (see section 2.2).

b. `alpha-D`
   (in front of trivial names with ring connection, e. g.
   `alpha-D-Ribose`)
c. `D-threo`
   (in systematic names, e. g.
   `D-threo-tetrodialdose`)
d. `alpha-D-threo`
   (in systematic names with ring connection, e. g.
   `alpha-D-threo-Hexo-2,4-diulo-2,5-furanose`)

Configurational prefixes are usually stated together with a preceding configurational symbol (e. g. `D`, `L`), which specifies the configuration at the highest-numbered centre of chirality, the so-called configurational atom.

Anomeric configurational symbols (α, β) determine the stereochemistry of a compound containing a ring connection. They are attached in front of a configurational prefix (in the name as well as in the semantic representation), namely that one describing the part of the structure which comprises the highest-numbered carbon atom involved in the ring connection.

In the semantic representation, the list of (possibly several) configuration specifications (each one being of a form shown exemplarily in (43)) is the argument of the functor `cfg`. This expression is put into the list of prefixes being an argument of `compd`. Example (44) shows a partial semantic representation for `D-glycero-L-gulo-heptose`.

(44) `compd(ose(??*[??],7*C),pref([cfg([D-glycero,L-gulo])]),suff([]))`

Chirality Sign Assignment To generate a SMILES string of a sugar compound name containing configuration specifications, we process the configurational symbol first, after that the configurational prefix, then the anomeric configurational symbol, always in this order, skipping the configuration specifications that are not given. In each step, chirality signs (`@` and `@@`) are assigned and changed respectively. Each sign is attached to the feature list of the corresponding chain element, designated by the functor `chir` (see example (45)).

(45) `chain_el(C,Index,[[[H]],[O]],[chir(@)])`

Traversing the list of chain elements one by one, each element's properties are checked to decide if a chirality sign is to be added or if it is possibly to be changed, if already existent. Prolog facts exist associating each of the configurational prefixes its list of chirality signs in the appropriate order (as shown in example (46)). They are used like a lexicon, for looking up the list of chirality signs needed in this processing step. Likewise we have also provided

Prolog facts that associate combinations of an anomeric configurational symbol and a configurational symbol with the corresponding chirality sign (see (47)). Moreover, we provided two Prolog facts which have the purpose to deliver the respective complementary chirality sign.

(46) `lex_smile(threo,['@@','@']).`

(47) `lex_smile(alpha-'D','@').`

Considering Ring Elements In the processing step of assigning chirality signs, each lower-numbered element of a pair of elements involved in a ring connection is considered. If it bears the feature `carbgrp` marking that a carbonyl group once existed there, no chirality sign is assigned. If no carbonyl group feature is encountered, one chirality sign is removed from the list of chirality signs to be assigned. Each higher-numbered ring connection element will be ignored in assigning a chirality sign for another reason: It does not have the required branches list consisting of a hydrogen and a hydroxy group. We will assign the appropriate chirality signs for ring connection elements later in the process, when processing the anomeric configurational symbol. That is because the two chiralities to be assigned to the ring connection elements are different from each other, and they depend on the combination of anomeric configurational symbol and configurational symbol.

Processing Order The order of processing different detachable and configurational prefixes is crucial for the correct specification of chiralities in the pre-representation of the SMILES string. In fact, the assignment of chirality signs still poses some difficulties for the current implementation.

If a `deoxy-` prefix (deleting an oxygen atom) is applied before the configuration prefixes, at those locants a chirality cannot be assigned because our application method for the configuration prefixes requires a hydroxy (and a hydrogen) branch to be existent there; it would not make sense to assign a chirality as there is no centre of chirality any more (the carbon atom is not asymmetric anymore: it has two branches of the same groups, namely hydrogen groups). This mechanism is desirable only in cases where the oxygen atom is not to be replaced.

But, if the `deoxy-` prefix is applied and an additional prefix is specified which adds a molecule at the same locant and branch (i. e. it replaces the oxygen), e. g. `amino-`, the chirality sign should be assigned despite the specified `deoxy-` prefix.

The difficulty is basically that our application method for the configuration specifications currently requires the branches list consisting of a hydrogen and hydroxy group. Only in this case chirality sings will be assigned. This is to ensure that they are not assigned spuriously to elements without asymmetric branches.

When appearing in front of a trivial name, the `deoxy-` prefix deletes an oxygen atom from a chain element where the chirality has already been assigned. If the hydroxy group is only deleted and not replaced, the chirality sign is not necessary any more. In case it is replaced, this molecule takes the position of the hydroxy group, also in respect to its chirality, and so the chirality sign has to be kept.

In the current implementation, we first process the configuration specifications, then the `deoxy-` prefixes, and after that the other prefixes. An enhancement of this method could be to assign chiralities not until all other prefixes have been processed. In that case, a check for asymmetry of each carbon atom would be necessary. However, a simple equality comparison of branches representations is not possible because of special branches (e. g. `deoxy-[H]` and `ring-O`).

Detachable Prefixes

Prefixes which need not to be attached directly to the parent name, are called detachable prefixes. Each prefix bears its own operation purpose which is to be used for altering the parent structure. Possible operations are replacing, adding, or removing chemical elements or molecules. The respective prefix operation is applied to the pre-representation of the molecular structure built so far.

Single Prefixes Processing simple prefixes, the information needed for the respective operation is provided in form of a lexicon entry as shown in (48) for the prefix `thio-`. As the operation types mentioned can all be modeled by substitutions of branches, currently only this kind of operation is implemented. Thus, each entry includes a prefix name, the operation type `substitute`, the substituted element, and the substitution element.

(48) a. format:
```
lex_smile(PrefixName,OpType,[Substituted,Substitute]).
```
b. example:
```
lex_smile(thio,substitute,[O,S]).
```

The lexicon look-up is only meant to be working for predefined prefixes, of course. In case the lexicon look-up fails, we suppose that an embedded compound is given as a prefix. If possible, the pre-representation for this compound is generated and used as the substitute for the chain element's side branch at the specified locant. To be attached there to a carbon atom, we determined the precondition that an oxygen atom has to be missing at its side branch. This is true, if a `deoxy-` prefix was applied before; thus the compound is only allowed to substitute a side branch represented by `[deoxy-[H]]`.

Double Prefixes Double prefixes of the semantic representation type as shown in (49) can be processed similarly to single prefixes.

(49) `Multiplier*[Locs_List]-Element-PrefixName`

For sugar compounds, `Element` may be e. g. `C`, `O`, or `N`, of which the two latter elements are not implemented, yet. Double prefixes including the element `C` are used to specify a substitution of a hydrogen atom (as a branch of a carbon atom) for `PrefixName`, where `PrefixName` may be a compound itself. We implemented this prefix type to work with embedded compounds, especially to integrate a simple non-sugar compound like `methyl` into our sugar SMILES string generator.

Dependent Prefixes Prefixes can depend on each other, e. g. the prefix `amino-` requires the appropriate `deoxy-` prefix to be present in the name (see IUPAC-IUBMB Joint Commission on Biochemical Nomenclature, 1996, R.2-Carb-14.1)). For implementing the dependency of the prefixes `deoxy-` and `amino-` (as an example), we opted for an approach which changes the branches list: By applying the `deoxy-` prefix on a chain element, the oxygen branch in the branches list is substituted by `[deoxy-[H]]` instead of just being deleted from the list. When applying the `amino-` prefix then, the amino group is tried to be substituted for this branch to ensure that a `deoxy-` operation was conducted before.

However, the `amino-` prefix might be as well used without specifiying also `deoxy-` in some names, although exactly the same operation is meant. For that case, we implemented an alternative (although not according to the IUPAC nomenclature), which tries to substitute the amino group for an oxygen branch. In case `deoxy-[H]` has not been substituted by any prefix at the end of the pre-representation construction, there is a rule in the output routine to treat it correctly, i. e. to ignore '`deoxy-`' (see the output routine subsection).

Underspecifications

Chemical compound names can be underspecified in various ways, namely by using class names or including prefixes, suffixes, or parent structure morphemes where not all information is provided, i. e. where the locants are missing. Ambiguous names can be seen as another kind of underspecification. Furthermore, some underspecified names used in the literature are meant to describe a certain, fully specified structure, e. g. `ribose`[63], where `D-ribose` would be correct (or one of its cyclic forms, depending on the context: α-`D-ribose` or β-`D-ribose`).

Some seemingly underspecified terms in our semantic representations ('`??`' represents a missing but expected value, mostly used for locants and multipliers) are only the consequence of consistently assigning syntactic structures to names. For example, a semantic representation for multiplier and locants is created also for the parts of a name consisting only of one of them (to lower the number of rules, always the same grammar rule is used, covering all their combinations). An example is `hexose` (locants are never specified for aldoses), which is semantically represented in our system as `ose(??*[??],6*C)`. Such underspecifications can be resolved by applying defaults.

We treat all the underspecification types described above by trying to resolve the underspecification or, if this is impossible, by stating the parts of the name which are underspecified. Their output is in the form of packed representations determining the underspecified operations and all possibilities where they could be applied.

Parent Names In the semantic representation of parent skeletons, there are terms with expressions of the form `Multiplier*[Locs_List]` which seem to be underspecified, e. g. as in `ose(??*[??],6*C)`. This term represents `hexose` where neither multiplier nor locants were specified in the name. Only the number of carbon atoms is specified. Aldoses named this way are meant to have one carbonyl group at locant number 1. Thus, the term defaults should be 1 for the multiplier and 1 for the list of locants. These default values should then be used to replace the underspecified markers '`??`' in the semantic representation for the underspecification to be resolved.[64] There are more examples like this, where multiplier and locants should get default values unless they are specified.

For examples bearing underspecification concerning the ring connection elements (e. g. `hexopyranose`), it is not clear where to construct the ring connection, and there are no defaults that could be applied. However, there is only one possibility for a ring connection. As the ring size is specified (a `pyranose` has six ring elements, including one oxygen atom) and by knowing that one chain

[63]Note that this is not only an underspecified name, but also a trivial name.

[64]In our implementation, however, we directly apply the appropriate operations for the term to the pre-representation, because changing the values in the semantic representation would be an unnecessary intermediate step in this case.

element with a carbonyl group will be one connection element, both locants are inferred.

Yet other examples are underspecified and not resolvable, e. g. for `hexos-2,3-ulooxirose` one cannot decide where to construct the ring connection, as there are three carbonyl groups, each possibly serving as one of the two ring connection elements. Hence, there are three possible ring connections and it cannot be inferred where one should be constructed. In the current implementation, the first possible ring connection is constructed.

Names without configuration specifications could be treated as another kind of underspecification, as there are several combinations of chirality signs possible to assign to a parent structure. We have not treated that as an underspecification in the scope of this work.

Prefixes The semantic representation of a prefix occuring without a list of locants is underspecified. For processing such a prefix, we use the Prolog clause for fully specified prefixes, but call it with a variable substituting a locant, which results in finding an appropriate one. In our implementation, Prolog is instructed to find all solutions to the clause by calling it repeatedly as described above. This procedure yields all the possible locants for applying the current prefix to the pre-representation built so far. The acquired list of locants is attached to the prefix in the same format as in the semantic representation: `[Locants]-Prefix`. The resulting term is then inserted into the list of underspecifications, which is the only argument of the functor `uspecs`. As stated before, this functor is the first element in the list of the pre-representation. The pre-representation remains unchanged in other respects. Example (50) represents an compound with a fully underspecified specification for the `deoxy-` prefix which can be applied to locants 1–5.

(50)
```
[uspecs([??*[1,2,3,4,5]-deoxy]),
 chain_el(C,1,...),
 chain_el(C,2,...),...]
```

In a next step, one could try to infer which prefixes at which locants can be applied to the pre-presentation created finally, i. e. which underspecifications are resolvable, thereby considering coinciding locants and prefixes depending on each other as described above.

When trying to generate the list of locants for the prefixes `deoxy-` and `amino-`, we encountered the following difficulty: If the `deoxy-` prefix is underspecified and thus does not change the pre-representation, the `amino-` prefix cannot be applied to any chain element, because the dependency on the `deoxy-` prefix is modeled as described earlier in this subsection. This is another reason for allowing the alternative proccessing of the `amino-` prefix, namely to be applied (in a second try) independently from the `deoxy-` prefix.

Prefixes which cannot be applied to any chain element are inserted into the list of underspecifications, too, but with their underspecified markers '`??`' instead of a list of locants. – to express that they are probably not applicable in the way described, but they could as well just miss in the lexicon, be spelled incorrectly, or a naming error.

Class Names Pure class names as well as class names mixed with systematic nomenclature rules forming a semi-systematic class name occur. The SMILES string generator does not deal with names of both of these types; these cases are not covered in the frame of this work. If enhanced, it should express the underspecification (especially for the latter case) by printing the morphemes that are involved so that more detailed information is available.

Output Routine

The SMILES string generation requires a more sophisticated output routine than the remaining output data. This is due to the fact that it is based on an intermediate pre-representation, which is necessary for the flexible and frequent insertion and deletion of elements. Furthermore, ways to deal with underspecifications had to be included. The SMILES string output is generated stepwise, as each chain element and each branch have to be checked for certain features before being printed accordingly. That is to say, the SMILES string generated is not accessible via a Prolog variable, but displayed directly.

Underspecifications At the top level of the output, the pre-representation given as list of chain elements is processed element by element. The first element being the list of underspecifications is treated specially. In case it is not empty, a functor-argument representation will be created, which combines the SMILES string and the underspecified prefixes as shown in (51a). For `deoxy-tetrose`, the corresponding output is shown in (51b).

```
(51) a. underspecified(SMILES_String,Underspecified_Prefixes_List)
     b. underspecified(C(=O)C([H])(O)C([H])(O)C([H])(O),
                       [??*2,3,4-deoxy])
```

The prefixes in the list of underspecifications are displayed in the same form as they occur in the pre-representation, differing only in that the list of locants is not embraced by brackets (`[`,`]`), but by braces ({,}) to indicate a set of possibilities. In case the list of underspecifications is empty, there is no such output and the next element in the pre-representation (which is the chain element with locant number 1) is processed.

Chain Elements Each of the chain elements is checked for a chirality symbol; if specified, the element is printed together with the element symbol, surrounded by brackets. After that, it is determined if the chain element is involved in a ring connection, and in that case the locant number serving as connection index is printed. Next, the side branches are output one by one, embracing each one by parentheses (‘`(`’,‘`)`’). Branches can themselves be embedded pre-representations of a complex compound, which will be treated as described above, before the next step.

Any other branch will first be checked for special ring representations such as `ring-O` (note that the `O` could be also substituted by a different element). If existent, the respective element will be printed together with the appropriate connection index which is to be found in the feature `ring_el2(Index)` of the feature list. Other special expressions prefixed with a hyphen to an element (as in `deoxy-[H]`) are omitted and only the element is output. Furthermore, a branch specification in the pre-representation can consist of a list of elements and binding types, which are just output altogether.

Our pre-representation of SMILES strings is an easily extendible language; the output routine has to be adapted accordingly. For example results of the SMILES string generator see the testsuite in appendix B.2.2.

4.4 Classifier

The module for the structural classification of chemical compound names is based on the morpho-semantic analysis yielded by the parser. Our ‘chemical class calculus’ takes a semantic representation term (as described in section 4.2) as input and yields a list of classes for each non-sugar name.

Compound classes are defined according to the combination of a name’s morphemes, which partly represent its functional groups. The morphemes are coded in our semantic representation language in the form of predicates with determined arities (predicate-argument terms). We use rules (Prolog clauses) formulated on the basis of domain knowledge, i. e. nomenclatures and chemical defaults, to determine the corresponding classes. Along with the classes for a name, (if applicable) the respective functional group on the basis of which a class is defined is provided in parentheses, e. g. ‘`ALCOHOL (-OH)`’. Fully specified names are processed first, followed by the underspecified cases.

Procedure

The three arguments of the semantic representation predicate `compd` are adjusted to lists of an analogue format in order to compare them in a next step. For example, the semantic representation term for `2-oxahexan-1-ol` in (52) yields a parent predicate list `[6*C-ane]`, an unchanged prefix predicate list `[??*[2]-oxa]`, and an also unmodified suffix predicate list `[??*[1]-ol]`.

(52) `compd(ane(6*C),pref([??*[2]-oxa]),suff([??*[1]-ol]))`

The general format of the output with the three predicate lists is illustrated in example (53).

(53) `compd(ParentPredList,PrefixPredList,SuffixPredList)`

For nested parent predicates such as `ene(??*[2],ane(5*C))`, the order of items in the parent list (`[??*[2]-ene,5*C-ane]`) corresponds to their nesting, i. e. from the outermost to the innermost predicate.

The core part of the class calculation consists of a complex set of rules with conditions for particular cases of predicate combinations. Most classes are calculated directly from the morphemes (for parent and suffix predicates). For example, according to the list of parent predicates, the combinations `(cyclo,ene,ane)`, `(cyclo,ane)`, and `(yl,ane)` are assigned the classes `CYCLOALKENE`, `CYCLOALKANE`, and `ALKYL`, respectively. Table 5 shows several of the morpheme-class mappings implemented.

morpheme	class (defining property)
-ane	ALKANE (single bond)
-ene	ALKENE (double bond)
hydroxy-, -ol	ALCOHOL (-OH)
amino-, -amine	AMINE (-NH2)

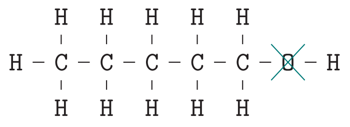
Table 5: Morpheme-class mapping examples

Occasionally, prefix predicates do not simply add more classes, but ‘interact’ with parent or suffix predicates in such a way that the classes deduced from the latter are either substituted by other classes (class-changing process) or deleted (destructive process). As an example for such a prefix effect, the prefix `2,3-dihydro` in `2,3-dihydropent-2-ene` ‘neutralises’ the `-ene` (double bond) desaturation: the latter is ‘no longer’ of the class `ALKENE` (see example (55) for an image of the molecule structure).

The evaluation of the predicate lists to generate a class list proceeds as follows. First, very specific combinations of predicates are treated. They are processed first with a special rule before general rules are applied. With the help of Prolog cuts (described in section 4.2), it is made sure that more general rules are prevented from being applied later if such specific rules succeed. The following examples show the treatment of increasingly general cases of predicate combinations.

In example (54) with the subtractive nomenclature operation prefix `deoxy-`, which removes the oxygen atom and thereby the functional group for alcohol, the rule in (54c) describes the following: The predicate list combination (`ParPredList`, `PrefPredList`, `SuffPredList`) is turned into a class list by 'collecting' certain predicates being members[65] of the three lists. These three predicates are stored in the variables `ParPred`, `PredPred`, and `SuffPred`. The combination of these three variables is then checked in the lexicon-like Prolog fact (54d) which yields a corresponding class. The lexicon is hand-coded according to chemical domain knowledge.

(54) a. name:
`5-deoxypentan-5-ol`



b. semantic representation:
`compd(ane(5*C),pref([??*[5]-deoxy]),suff([??*[5]-ol]))`

c. Prolog clause:
```
predlistscomb2classes(ParPredList,PrefPredList,
                      SuffPredList,Class)
:-
      member(_Mult*_Elem-ParPred,ParPredList),
      member(_Mult*[Loc|_Rest]-PrefPred,PrefPredList),
      member(_Mult*[Loc|_Rest]-SuffPred,SuffPredList),
      prefchangeclass(ParPred,PrefPred,SuffPred,Class),
      !.
```
d. Prolog fact:
`prefchangeclass(ane,deoxy,ol,[ALKANE]).`

After that, cases with only two predicates in combination are dealt with, e. g. for `2,3-dihydropent-2-ene` with the additive nomenclature operation prefix `dihydro-`. The rule and lexicon entry are shown in (55).

[65]The built-in Prolog predicate `member` succeeds if its first argument is contained in the list in its second argument.

(55) a. name:
`2,3-dihydropent-2-ene`

```
    H   H   H   H   H
    |   |   |   |   |
H - C - C = C - C - C - H
    |   |   |   |   |
    H   H   H   H   H
```

b. semantic representation:
`compd(ene(??*[2],ane(5*C)),pref([2*[2,3]-hydro]),suff([]))`

c. Prolog clause:

```
predlistscomb2classes(ParPredList,PrefPredList,[],Class)
:-
      member(_Mult*[CommonLoc|_Rest]-ParPred,ParPredList),
      member(2*[CommonLoc,FollowingLoc|_Rest]-PrefPred,
                                             PrefPredList),
      prefchangeclass(ParPred,PrefPred,none,Class),
      FollowingLoc is CommonLoc+1.
```

d. Prolog fact:
`prefchangeclass(ene,hydro,none,[ALKANE]).`

Then, cases where one predicate does not have influence on the classification are processed, e. g. for `2-phosphapentan-5-ol` with the replacement nomenclature operation prefix `phospha-`. As this prefix does not influence the classification process, it gets the property `no_change` assigned. More such prefixes can thus be subsumed in a 'class of unpersuasive prefixes'.

(56) a. name:
`2-phosphapentan-5-ol`

```
    H   H   H   H   H
    |   |   |   |   |
H - C - P - C - C - C - O - H
    |   |   |   |   |
    H   H   H   H   H
```

b. semantic representation:
`compd(ane(5*C),pref([??*[2]-phospha]),suff([??*[5]-ol]))`

c. Prolog clauses:

```
predlistscomb2classes(ParPredList,PrefPredList,
                                       SuffPredList,Class)
:-
      member(Loc*_Elem-ParPred,ParPredList),
      member(_Mult*_LocList-PrefPred,PrefPredList),
      member(_Mult*[Loc|_Rest]-SuffPred,SuffPredList),
      prefchangeclass(ParPred,PrefPred,SuffPred,Class),
      !.
prefchangeclass(ane,AnyPref,ol,
                  [ALKANE,PRIMARY ALCOHOL (-OH)])
```

```
:-
      no_change(AnyPref).
```

d. Prolog fact:

```
no_change(phospha).
```

Cases where only one (i. e. the parent) predicate is taken into consideration, such as for `pentane` (without affixes), are then dealt with. Only a lexicon entry is needed in this case.

(57) a. name:
`pentane`
```
    H   H   H   H   H
    |   |   |   |   |
H - C - C - C - C - C - H
    |   |   |   |   |
    H   H   H   H   H
```
b. semantic representation:
`compd(ane(6*C),pref([]),suff([]))`
c. Prolog fact:
`predlistscomb2classes([_Mult*_Elem-ane],[],[],[ALKANE]).`

Additional classes are yielded by taking superclasses out of a (manually defined) hierarchy, e. g. with the axiom “Primary alcohols are alcohols.”. The rule looks as shown in (58) and can be paraphrased as follows: Add the more general class `ALCOHOL (-OH)` to the class list generated so far if `PRIMARY ALCOHOL (-OH)` is a member of this class list.

(58) Prolog clause:
```
addsuperclasses(ClassList,['ALCOHOL (-OH)'|ClassList])
:-
     member('PRIMARY ALCOHOL (-OH)',ClassList).
```

Class names such as `alkene` and modified ones (semi-systematic class names) such as `2-alkene` are treated as shown in (59) and (60), respectively. The clauses express that the class names (‘labeled’ by `class_name` in the semantic representation) are directly written to the class list variable.

(59) a. name:
`alkene`
b. semantic representation:
`compd(class_name(alkene),pref([]),suff([]))`
c. Prolog clause:
```
classes(compd(class_name(ClassNameforList),
                pref([]),suff([])),[ClassNameforList]) :- !.
```

(60) a. name:
`2-alkene`
b. semantic representation:
`compd((??*[2],class_name(alkene)),pref([]),suff([]))`
c. Prolog clause:
```
classes(compd(_Mult*_Locs,class_name(ClassNameforList),
                pref([]),suff([])),[ClassNameforList]) :- !.
```

A direct systematic classification of trivial names is not possible, therefore the corresponding classes have been entered manually. See (61) for the `benzene` example.

(61) Prolog clause:
```
classes(compd(triv_name(benzene),pref([]),suff([])),
        ['AROMATIC']) :- !.
```

For a methodic extension of this ‘lookup’ data, classes for trivial names could be acquired, e. g., (by a corresponding tool) from SMILES strings, if available. This procedure could also be enhanced by replacing the trivial names with their corresponding systematic names (by means of a lexicon) and classifying these. However, such systematic names are often rather complex (which is probably why they were substituted by trivial names); their classification would not have been covered in our prototype.

More examples as to special rules and lexicon entries can be found in the file `classes.pl` in appendix A.

Perspectives

As far as the completeness of calculated classes is concerned, such a purely linguistic approach has a potential drawback in comparison to a graph-of-molecules processor. If new functional groups emerge in a compound structure from the combination of original functional groups, a graph handles that simply by searching for subgraphs (see Wittig et al., 2004). As such new functional groups may not be expressed by a new morpheme in the compound name, a classification relying exclusively on the name itself could be supplemented by a classification via SMILES. As an alternative, a sophisticated deep semantic interpretation of complex morpheme combinations has to be developed.

For the classifier module to become exhaustive, classification has to be handled by directly starting with the name as the most specific kind of class. By detaching the affixes incrementally and thereby ‘abstracting’ step by step, partial names corresponding to intermediate class names will emerge. Additionally, more superclasses have to be added by adding further axioms. After these analyses, predicate interaction must be considered and potential class changes must be made. Our system provides a basis for the ultimate aim to systematically derive all intermediate classes and superordinate classes. Like that, a comprehensive hierarchy can be built to be used as an ontology of chemical compounds and their classes such as the one in figure 13.

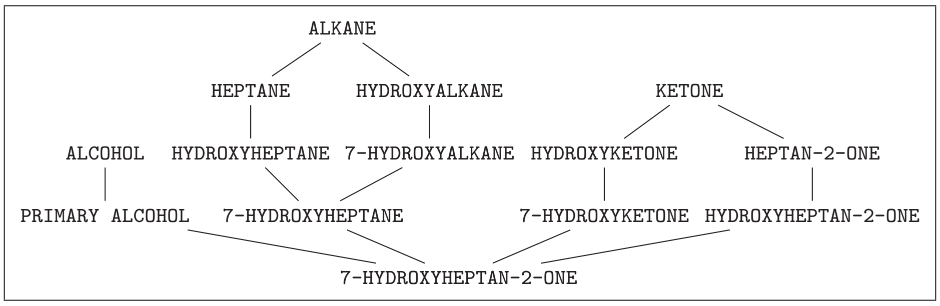


Figure 13: Class hierarchy for `7-hydroxyheptan-2-one`

4.5 Results

The main output of our tool consists of a semantic representation of a compound name together with a SMILES string for sugar compounds and a list of classes for non-sugar compounds.[66]

The concrete output format for a Prolog call with the name `hexose` looks as shown in (62). The `org_compd` call is followed by a syntactic tree, the SMILES string, and the variables with their values of the complex semantics, the syntax, the β-reduced semantics corresponding to our semantic representation, and the class list.

(62)
```
| ?- org_compd(Syn,Sem,[h,e,x,o,s,e],[]),pp(Syn),
               beta_reduce(Sem,RedSem),
               smile(RedSem),classes(RedSem,ClassList).
=org_compd
|=par_sugar
| |=mult
| | |=[ h e x ]
| |=ps_zero
| | |=stem_suff
| | | |=[ o s e ]

C(=O)C(O)C(O)C(O)C(O)C(O)

Sem = compd(lam(??*[??],lam(6*C,ose(??*[??],6*C)))@
                          ??*[??]@6*C,pref([]),suff([])),

Syn = org_compd(par_sugar(mult([h,e,x]),
               ps_zero(stem_suff([o,s,e])))),
```

[66]For the time being, the complementary parts, i. e. SMILES for non-sugar and classes for sugar names, are not provided (see section 1).

```
RedSem = compd(ose(?? *[??],6*C),pref([]),suff([])),

ClassList = [NO CLASSES]
```

In the testsuite, this output is simplified; only the name, its list representation, the reduced semantics, the SMILES string, and the class (list) are printed, if available. The compact analyses are followed by a triple dash (`---`); an empty line separates different names from each other. If several analyses exist for one name, these are separated only by a double dash as displayed in (63).

(63)
```
hexane
[h,e,x,a,n,e]
compd(ane(6*C),pref([]),suff([]))
NO SMILES
ALKANE
---

cyclohexatriene
[[c,y,c,l,o,h,e,x,a,t,r,i,e,n,e]]
compd(cyclo(??*[??],ene(6*[??],ane(3*C))),pref([]),suff([]))
NO SMILES
CYCLOALKENE
---
compd(cyclo(??*[??],ene(3*[??],ane(6*C))),pref([]),suff([]))
NO SMILES
CYCLOALKENE
---
```

A sample of analyses with classes for non-sugar and SMILES strings for sugar compound names can be seen in table 6. For the former case, the non-sugars, `7-hydroxyheptan-2-one` is a complex example, `alkene` a class term and `dipentene` an underspecified name. For the latter, we show `L-threo-tetrodialdose` as an example with configurational prefixes and `D-fructose` as a trivial name; `2-pentulose` and `pent-2-ulose` are synonyms due to nomenclature variations, which can be matched by their identical semantic representations and SMILES strings.

We cover nomenclature operations such as substitutive (exchange of `H`), replacement (exchange of `C`), additive, subtractive operation, ring formation, as well as stereochemical features. The complete testsuite for our regression testing with examples for all the phenomena covered is attached in appendix B.2.1.

A general assessment of the quality and the degree of coverage of this output is difficult, because our lexicon is not filled with sufficient morphological infor-

name	results
7-hydroxyheptan-2-one	compd(ane(7*C),pref([??*[7]-hydroxy]),suff([??*[2]-one])) NO SMILES KETONE (>(C)=O)
alkene	compd(class_name(alkene),pref([]),suff([])) NO SMILES ALKENE
dipentene	compd(ene(2*[??],ane(5*C)),pref([]),suff([])) NO SMILES ALKENE (double bond)

L-threo-tetrodialdose	compd(ose(2*[??],4*C),pref([cfg([L-threo])]),suff([]))
	C(=O)[C@]([H])(O)[C@@]([H])(O)C(=O)
	NO CLASSES

D-fructose	compd(triv_name(fructose),pref([cfg([D])]),suff([]))
	C(O)C(=O)[C@@]([H])(O)[C@]([H])(O)[C@]([H])(O)C(O)
	NO CLASSES

2-pentulose	compd(ulose(??*[2],5*C),pref([]),suff([]))
	C([H])(O)C([H])(O)C(=O)C([H])(O)C([H])(O)
	NO CLASSES

pent-2-ulose	compd(ulose(??*[2],5*C),pref([]),suff([]))
	C([H])(O)C(=O)C([H])(O)C([H])(O)C([H])(O)
	NO CLASSES

Table 6: Example analyses: the results contain our semantic representation, the corresponding SMILES string, if available, and the value of the class list variable

mation. Additionally, there is no data to be referred to for name analysis (terms with correctly annotated structures), especially not for underspecification.

A rough estimation yields the supposition that missing rules are rarely the reason for wrong or no results, but that mostly missing lexicon entries, especially for trivial names, cause such failures. The situation is comparable to the one with natural language systems, where the rules can be coded quite completely as opposed to the lexicon, which is practically open-ended.

Another issue to be considered are the ‘impure’ results for large-scale automatically generated compound lists. From one such list extracted from the KEGG database by the Scientific Database and Visualisation (SDBV) group[67] of the European Media Laboratory (EML), Heidelberg, we show the examples in (64) for entries where our tool failed because of its current limitations.

(64) a. (partial) sum formula:
`H2O`; `OH-`
b. comments at the end of lines:
`(oxidized); with phosphoric propanoic acid`
c. miscellaneous:
`2-Propanone 1-hydroxy-3-(phosphonooxy)-;`
`(R)-(-)-Epirenamine; L-Tyrosyl-tRNA()`

Nevertheless, we provide with this system a sophisticated basis for biochemical name analysis, which functions well for recommended names corresponding to the major nomenclature rules. This is crucial since semi-automatic NLP methods for the tool’s extension can now be brought forward in order to essentially enhance biochemical research.

4.6 Applications

In this subsection, we illustrate several contexts in which our tool can be used to support research in the biochemical field.

The analysis of terminology is an important basis for computational text processing, as unrecognised or unanalysed terminology poses a serious problem there. If a semantic description can be provided for special terms, their identification supports the processing of the huge amount of data available, especially for the life sciences (see Krauthammer and Nenadić, 2004).

Intermediate classes as shown in figure 13 on page 63 become crucial for automatic intelligent text processing, e. g. if an article which treats specific compounds only mentions one of its superclasses in the title. An example for a publication containing such an intermediate class (viz `Hydroxyketone`) is

[67]Thanks to all for their support of our work.

Ishmuratov et al. (2001): “Ozonolysis of Alkenes and Study of Reactions of Polyfunctional Compounds: LXIII. A New Procedure for Direct Reduction of 1-Methylcycloalkene Ozonolysis Products to Hydroxyketones”.

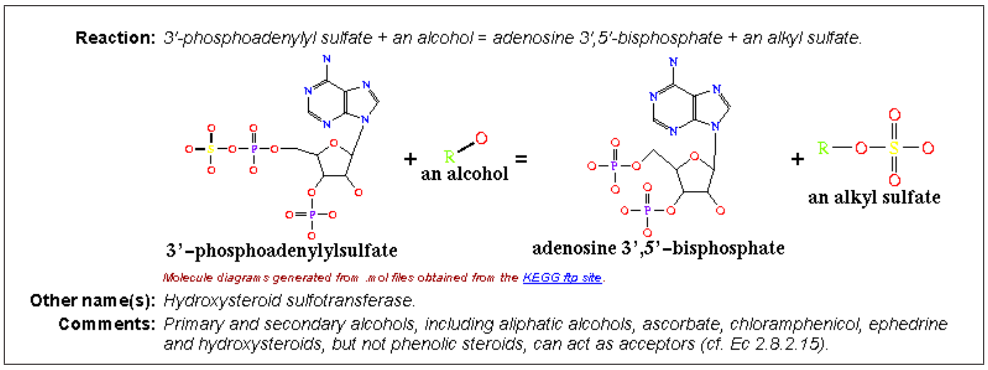
The support of biochemical database curation as, e. g., described in Rojas et al. (2002) is another kind of application.[68] The need for high-quality databases requires the manual work of experts, which is time-consuming and expensive. Therefore, semi-automatic database population and integration by NLP for efficient, correct, consistent and non-redundant as well as non-overlapping databases is necessary. Until now, most biochemical databases consist of plain text and inserted images of molecules, i. e. they are ‘flat’ (without a deep structure) and do not offer the possibility to deduce further information. For such a purpose, automatised deeply-structured databases are needed and must be populated with the help of semantic processing or ontologies. In this population task, our resulting SMILES strings can be used for term reference, i. e. the assignment of a name to a structure, and for resolving coreferences. Multiple entries for one and the same molecule (as seen in figure 2 on page 2) can be detected and identified as such by the comparison of their corresponding SMILES strings. Their elimination will contribute to database reliability; besides, the population and curation of databases can be accelerated, which is very important because of the huge amount of data available.

The classification of chemical compound names serves for database curation as well. Compounds can be subsumed according to their classes; like that an application can predict their common features. Thus, relations between compounds and their classes are established in a hierarchical way, which helps to automatise database handling.

Classification is also crucial for the processing of abstract reaction equations. If a general class name or an otherwise underspecified name occurs in a reaction equation such as in the example in figure 14 (taken from the Enzyme Structures Database of the EBI[69]), possible instances of this equation can be generated automatically. The comment lines have to be considered if they contain additional constraints (e. g. ‘but not . . . ’), or they can be searched for relations between compounds and their classes such as ‘is a’, ‘part of’, etc. Biochemical reactions at different levels of abstraction can thus be compared to each other. In general, data which is represented in different ways can be matched, e. g. in distinct databases for their integration.

[68]We conducted a preliminary implementation of the system (called from a ‘Java’ application) at EML for the System for the Analysis of Biochemical Pathways (SABIO) in the BioBrowser environment.

[69]EBI: European Bioinformatics Institute (http://www.ebi.ac.uk/thornton-srv/databases/enzymes)



Reaction: *3'-phosphoadenylyl sulfate + an alcohol = adenosine 3',5'-bisphosphate + an alkyl sulfate.*

Molecule diagrams generated from .mol files obtained from the KEGG ftp site.

Other name(s): *Hydroxysteroid sulfotransferase.*

Comments: *Primary and secondary alcohols, including aliphatic alcohols, ascorbate, chloramphenicol, ephedrine and hydroxysteroids, but not phenolic steroids, can act as acceptors (cf. Ec 2.8.2.15).*

Figure 14: Example entry for a reaction equation

All the above is associated to the field of bioinformatics, where the development of computational methods to model, simulate and analyse biochemical processes, e. g. also for the research on medication, is a central point of interest.

5 Outlook

In this section, we will first describe concrete proximate steps to extend our system and second take a look at further enhancement possibilities.

5.1 System Enrichment

Some proposals for a tangible extension of our work arelisted in this subsection. For testing a potential database curation support and, in the course of that, for the incremental semi-automatic perfection of the tool via user feedback, the system should be integrated into a corresponding environment.

Parser

For better coverage of data, the system's functionality has to be extended by completing the linguistic analysis components. On the one hand, the grammar has to be expanded according to nomenclature rules[70]and in respect of rather syntax-based rules as mentioned in section 4.2. On the other hand, especially the morphological lexicon[71] must be enriched, e. g. also with features such as the rule-based combination of multipliers (`pentadec` → `[[penta][dec]]`). This should be done semi-automatically with the help of computational methods. A sample of features not yet covered are complex class names such as `3-hydroxy acid`, the rearrangement nomenclature of compounds (`abeo-`), organometallic compounds (containing `vinyl`, `gold`, etc.), mononuclear hydrides with an explicit bonding number (λ^5`-phosphane`) or isotopically modified compounds (`Dichloro[`$^2H_2$`]methane`). Morpho-syntactic variations as well as exceptions to rules are other features to be implemented, and as the most important objects in a text are often abbreviated, the analysis of all kinds of abbreviations is to be added (cf. also the Biomedical Abbreviation Server[72] for a collection of abbreviations and their corresponding long forms). More intricate cases are, e. g., prefixes originated from compounds such as `methylimine`, which then end in a different vowel (`methylimino-`). As we do not want to list compounds as possible prefixes, a vowel change has to be allowed without creating too much overanalysis.

As a more elaborate system will yield more ambiguities, sophisticated solutions for disambiguation will have to be found. Weighted rules for the pref-

[70]Trying to implement CAS nomenclature rules in addition to IUPAC rules caused high overanalysis – a comprehensive coverage will thus present quite a challenge.

[71]A systematical lexicon enhancement will be done in a student research project at the IMS, University of Stuttgart.

[72]http://bionlp.stanford.edu/abbreviation

erence of frequent combinations may be introduced, as well as more complex disambiguation methods according to syntactic, semantic, discourse context, or ontologies. Implausible analyses can thus be marked or even suppressed.

As the number of rules increases and the lexicon grows, the efficiency of the tool will no longer be satisfactory with the current DCG parsing algorithm. The analyser will have to be converted into a more efficient parser with a more favourable complexity. For example, Voss (2004) describes an Earley chart parsing algorithm implemented in Prolog, with which also the difficulty concerning left-recursiveness (as mentioned in section 4.2) could be solved.

To further increase the efficiency of the program, a database containing all the names that have already been processed, together with their analyses, SMILES string and classes, should be automatically generated during the system execution. Frequent compound names will thus not be calculated repeatedly, but simply looked up in a database, which is a faster solution. As another extension, the stored result for a trivial name should also be re-used if this trivial name is part of a semi-systematic name, such as `benzene` in `benzene-1,3,5-triacetic acid`. Lists with the systematic equivalents to trivial names may be useful there; examples can be found, e. g., on http://www.chem.qmul.ac.uk/iupac/ions/app.html (for non-sugar names) or on http://www.chem.qmul.ac.uk/iupac/2carb/app.html (for sugar names).

The robustness of the system is to be enhanced, e. g., by allowing partial parses for incomplete input or input which is (to some extent) not compliant to rules. This can be done along the lines of an underspecification handling, where partial analyses and predictions about missing parts are needed as well.

A systematic error analysis with a corresponding output notice for the user, e. g. '`morpheme <...> is not in lexicon`', is also useful as a further development. Along with that, warning messages such as '`no longer recommended according to IUPAC`' could be provided.

SMILES String Generator

The SMILES module has to be elaborated and extended to comprehensive non-sugar name processing including also functional group suffixes, for example.

For class names (possibly mixed with systematic morphemes), a resulting SMILES string should express the underspecification, as opposed to underspecified prefixes, where the missing information can mostly be resolved by considering coinciding locants, for example. As currently the first possible ring connection is constructed for underspecified carbohydrate rings (see section 4.3), this can be improved by yielding all possibilities, e. g. in a packed representation.

Generally, the tool should be able to deal with resolvable and unresolvable underspecification by means of domain knowledge. For a comprehensive treatment of underspecified compound names, the idea of partial SMILES strings (with variables for unknown substrings) has to be further worked out and implemented in the system. Packed representations such as `{1,2}-ene` for `butene` according to a valence calculus or a detailed predicate logic description with operators on SMILES strings are possible solutions. 'Extended SMILES' providing explicit numbering (as already provided in the SMILES pre-representation), e. g. for certain trivial names with non-systematic, prescribed indices such as for `tryptophane`, are also to be developed. For such trivial names, special lexica with exactly this numbering information have to be used.

As an extension of the current consistency check as described in section 4.3, locant-multiplier combinations such as `2-di-` must be tested for their potential validity – the special case of two operations applied to one single atom without specifying its locant twice (`2,2-di-` would be correct) could be on hand.

Nomenclature-based synonyms which do not directly yield the same SMILES string should be identified with further thorough processing, for example, according to detailed saturation information.

Classifier

The class calculus has to be elaborated for sugar compound names and, e. g., also for a complete classification of semi-systematic names (combined with trivial or class names). Additionally, our name-based classification should be systematically compared to a functional group classification, e. g. with a 'loop-way' over SMILES strings, to assess and enhance it. The automatic acquisition of a 'natural' comprehensive ontology with a detailed classification has to be developed and implemented for results such as in the `hydroxyketone` example in figure 13 on page 63. Such a knowledge base for scientists and also a basis for advanced automatic knowledge management systems is crucial for further biochemical research.

5.2 Further Research

In the following paragraphs, we present several perspectives for possible research projects on the basis of a comprehensive tool as described in the last subsection.

As one continuative step, the component for the analysis of chemical terminology can be advanced to cover all types of (bio-)chemical terms[73], e. g. the enzyme names of BRENDA[74]. More rules of the 'still outstanding' IUPAC nomenclatures taken from IUPAC Commission on Nomenclature of Organic Chemistry (2005) could be integrated. Some of them are listed in table 7; note that several specific ones overlap with more general ones.

Branched nucleic acids	EC 5 Isomerases	Polymerized amino acids
Carbohydrates	EC 6 Ligases	Polypeptide conformation
Carotenoids	Folic acid	Polynucleotide conformation
Corrinoids (vitamin B12)	Glycolipids	Polysaccharide conformation
Cyclitols	Glycoproteins	Prenol nomenclature
Electron transport proteins	myo-Inositol numbering	Pyridoxal (vitamin B6)
Enzyme kinetics	Lignan Nomenclature	Retinoids
Enzyme nomenclature	Lipid Nomenclature	Steroids
EC 1 Oxidoreductases	Multienzymes	Tetrapyrroles
EC 2 Transferases	Nucleic acid sequence	Tocopherols (vitamin E)
EC 3 Hydrolases	Organic Chemistry	Translation Factors
EC 4 Lyases	Peptide hormones	Vitamin D

Table 7: A sample of IUPAC nomenclatures

Because of the huge amount of this nomenclature information, frequency data from texts and databases might have to be considered during the extension of the system in order to prevent needless inflation of rules and lexica.

While in the field of organic compounds there are many systematic names and a classification functions quite regularly, other biochemical terms such as protein complexes and enzyme names consist of more trivial and underspecified names and the classification is more difficult. The complexity can be indicated with the example of proteins depending on their environment and state for their classification. Additionally, the productive phenomenon of verb formation from biochemical terminology such as in `dephosphorylate` can be tackled.

A comprehensive valence and numbering model will have to be developed, e. g., for the proper treatment of fusion nomenclature, certain trivial names, or underspecified names for which the underspecified part can be resolved.

As our semantic representation language comprises all kinds of information on a compound in a standardised form, it can also serve as basis for other

[73] As a special challenge, we found the following statement on http://fun.drno.de/incoming/20020501/longestword.txt: "The longest word in the English language is 1,913 letters long and it refers to a distinct part of DNA".

[74] The Comprehensive Enzyme Information System (http://www.brenda.uni-koeln.de)

bioinformatics processing beyond the generation of SMILES strings and the classification of compounds, e. g. biochemical reaction modelling and simulation.

For a deeper understanding of texts, domain-specific inferencing and presupposition resolution are necessary, and semantic, discourse, and extralinguistic context properties must be exploited. In Cimiano (2002) and Cimiano et al. (2004), systems towards this end are described.

In general, interrelations between form and contents, e. g. between language and encoded knowledge (thus morphology and semantics), are also needed for more comprehensive ontologies than classification hierarchies or as well for semantic web applications.

An extension concerning human-machine interaction could be the repair of users' lexicographic errors, e. g., caused also by non-mother tongue speakers, to additionally enhance the system's robustness. This can be tackled with string matching algorithms and the calculation of similarity, for example. Luque Ruiz et al. (1996a,b) describe a system for the "Error Detection, Recovery and Repair in the Translation of Inorganic Nomenclatures" as to this issue.[75]

A morpho-semantic interface for biochemical terminology such as in our approach can be both transferred to other term languages than English and to other scientific domains with their respective terminology. The latter extensions could resemble DeriF (Namer and Zweigenbaum, 2004), a morpho-semantics parser to acquire a definition for French medical terminology.[76] Moreover, complex words of natural languages could be analysed similarly for a deep semantics calculation; one approach providing a basis for that can be found in Schmid et al. (2001).

An extended chemical compound name parser is a key to full term identification, i. e. recognition, classification and mapping (see Krauthammer and Nenadić, 2004). This, in turn, serves as a central component for information extraction (see Šarić, 2005), for text mining (see Tanabe et al., 1999), or for full text understanding of scientific publications in order to 'structure' the knowledge coded.

Coreference resolution in texts can as well be supported, e. g. for Discourse Representation Theory (see Kamp and Reyle, 1993; Kamp et al., 2004) according to sophisticated lexical semantics and detailed ontology modelling.

[75]From another viewpoint, Kirby and Polton (1993) propose the tackling of the difficulties in automatic chemical compound processing as a motivation to systematically change current nomenclature methods. This could, e. g., be supported by determining frequent error sources.

[76]One application of such a tool could be biomedical term extraction by entity recognition, e. g. described in Finkel et al. (2004).

Another valuable application of an exhaustive term understanding system can be a further automatised development of biochemical database systems. By means of a linguistic analysis of terms and, particularly, by their semantic normalisation, fully specified terminology can be covered by a fully automatic tool; for underspecified terms a semi-automatic, interactive dialogue with the expert to resolve remaining ambiguities is to be developed.

6 Summary

In this work we describe a natural language processing system developed in Prolog which analyses names of organic chemical compounds. We used a rule-based approach yielding an intermediate representation of semantic features contained in a name. This semantic representation is, on the one hand, further processed to generate a SMILES string describing the molecular structure, and, on the other hand, it is used for the classification of a compound name. The IUPAC nomenclature rules are the basis for our tool, which is able to analyse several types of names, including systematic, trivial, and underspecified ones such as class names. It could not be made exhaustive in the given framework, but it is a valuable prototype as a template which can be extended with a reasonable effort.

Existing similar systems either are not capable of assigning a molecular structure to chemical names directly or classifying them by a linguistic analysis, or they depend on databases and thus are not able to analyse new names not encountered before. None of the system spotted covers underspecified names, which appear frequently in biochemical literature.

An adequate representation of terminology presents a crucial basis for term identification in BioNLP applications. The necessary treatment of synonymous names is only possible by an analysis and direct understanding of terms. The system's possible applications in the field of computational text processing and bioinformatics include, e. g., supporting term extraction and database creation and curation. From further enhancements, which were beyond the objectives of this work, BioNLP can greatly benefit.

With this system we present a sophisticated basis for the semantic analysis of recommended names of an excerpt of nomenclature rules. Terminology decoding is essential for NLP methods in, e. g., biochemical research, and our work is a key to large-scale data processing in this field.

To conclude, the linguistic approach we present is well applicable for handling the existing complex biochemical data. By further extensive transfer between experts from the NLP and biochemistry disciplines, more methods can be developed to essentially support research in the life sciences by automating the processing of the huge and growing amount of biochemical data.

Bibliography

Appelt, D. E. and Israel, D. J. (1999). Introduction to information extraction technology. IJCAI-99 Tutorial. Retrieved November 2005, from http://www.ai.sri.com/~appelt/ie-tutorial/IJCAI99.pdf. (Cited on page 24.)

Beilstein, F. K. (1918–1997). *Beilsteins Handbuch der Organischen Chemie*. Springer, Berlin, fourth edition. Multi-volume. (Cited on page 13.)

Blackburn, P. and Bos, J. (2005). *Representation and Inference for Natural Language: A First Course in Computational Semantics*. CSLI Press. (Cited on page 8.)

Brecher, J. (1999). Name=Struct: A Practical Approach to the Sorry State of Real-Life Chemical Nomenclature. *J. Chem. Inf. Comput. Sci.*, 39:943–950. (Cited on pages 20 and 22.)

Cambridgesoft (2005). Converting Chemical Names to Structures with Name=Struct. Retrieved November 2005, from http://www.cambridgesoft.com/products/pdf/whitepapers/NameStruct.pdf. (Cited on page 20.)

Carstensen, K.-U., Ebert, C., Endriss, C., Jekat, S., Klabunde, R., and Langer, H., editors (2004). *Computerlinguistik und Sprachtechnologie – Eine Einführung*. Spektrum Akademischer Verlag, Heidelberg, second edition. (Cited on page 8.)

Chemical Abstracts Services (2002). Appendix IV. Chemical Substance Index Names. In *Chemical Abstracts Index Guide*, pages 175–328. Retrieved November 2005, from www.cas.org/ONLINE/UG/indexguideapp.pdf. (Cited on page 13.)

Cimiano, P. (2002). On the Resolution of Bridging References Within Information Extraction Systems. Master's thesis, University of Stuttgart. (Cited on page 73.)

Cimiano, P., Reyle, U., and Šarić, J. (2004). Ontology Driven Discourse Analysis for Information Extraction. *Data and Knowledge Engineering Journal*. (Cited on page 73.)

Clocksin, W. F. and Mellish, C. S. (1987). *Programming in Prolog*. Springer, Berlin, Heidelberg, third edition. (Cited on page 11.)

Cooke-Fox, D. I., Kirby, G. H., Lord, M. R., and Rayner, J. D. (1990a). Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 4. Concise Connection Tables to Structure Diagrams. *J. Chem. Inf. Comput. Sci.*, 30(2):122–127. (Cited on page 20.)

Cooke-Fox, D. I., Kirby, G. H., Lord, M. R., and Rayner, J. D. (1990b). Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 5. Steroid Nomenclature. *J. Chem. Inf. Comput. Sci.*, 30(2):128–132. (Cited on page 20.)

Cooke-Fox, D. I., Kirby, G. H., and Rayner, J. D. (1989a). Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 1. Introduction and Background to a Grammar-based Approach. *J. Chem. Inf. Comput. Sci.*, 29(2):101–105. (Cited on page 20.)

Cooke-Fox, D. I., Kirby, G. H., and Rayner, J. D. (1989b). Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 2. Development of a Formal Grammar. *J. Chem. Inf. Comput. Sci.*, 29(2):106–112. (Cited on page 20.)

Cooke-Fox, D. I., Kirby, G. H., and Rayner, J. D. (1989c). Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 3. Syntax Analysis and Semantic Processing. *J. Chem. Inf. Comput. Sci.*, 29(2):112–118. (Cited on page 20.)

Covington, M. A. (1989). Efficient Prolog: A Practical Guide. Research report, University of Georgia. Retrieved November 2005, from http://www.ai.uga.edu/~mc/ai198908.pdf. (Cited on page 38.)

Finkel, J., Dingare, S., Nguyen, H., Nissim, M., Manning, C., and Sinclair, G. (2004). Exploiting Context for Biomedical Entity Recognition: From Syntax to the Web. In *Proceedings of the Joint Workshop on Natural Language Processing in Biomedicine and its Applications at Coling*. (Cited on page 73.)

Fluck, J., Hofmann, M., Hahn, U., Wermter, J., and Schulz, S. (2005). Text Mining in den Life Sciences. *Deutsche Zeitschrift für Klinische Forschung*, 5/6:20–26. (Cited on page 1.)

Free Software Foundation (1987). What is swi-prolog? http://www.swi-prolog.org. (Cited on page 24.)

Gerstenberger, C. V. (2001). Semantische Analyse von Namen Organischer Verbindungen oder Was Bedeutet 3,3'-Ureylen-dibenzamidin? Master's thesis, University of Stuttgart. (Cited on pages 20, 22, 23 and 30.)

Ishmuratov, G. Y., Kharisov, R. Y., Yakovleva, M. P., Botsman, O. V., Muslukhov, R. R., and Tolstikov, G. A. (2001). Ozonolysis of Alkenes and Study of Reactions of Polyfunctional Compounds: LXIII. A New Procedure for Direct Reduction of 1-Methylcycloalkene Ozonolysis Products to Hydroxyketones. *Russian Journal of Organic Chemistry*, 37(1):37–39. publ: MAIK Nauka Interperiodica. (Cited on page 67.)

IUPAC Commission on Nomenclature of Organic Chemistry (1993). *A Guide to IUPAC Nomenclature of Organic Compounds (Recommendations 1993)*. Blackwell Scientific publications. Web version retrieved November 2005, from http://www.acdlabs.com/iupac/nomenclature. (Cited on pages 13, 14 and 23.)

IUPAC Commission on Nomenclature of Organic Chemistry (2005). Retrieved November 2005, from http://www.chem.qmul.ac.uk/iupac. (Cited on page 72.)

IUPAC-IUBMB Joint Commission on Biochemical Nomenclature (1996). Nomenclature of Carbohydrates (Recommendations 1996). *J. Pure Appl. Chem.*, 68:1919–2008. Web version retrieved November 2005, from http://www.chem.qmul.ac.uk/iupac/2carb/; also available as PDF-File from http://www.iupac.org/publications/pac/1996/pdf/6810x1919.pdf. (Cited on pages 14, 16, 17 and 52.)

Kamp, H. and Reyle, U. (1993). *From Discourse to Logic*. Kluwer, Dordrecht. (Cited on page 73.)

Kamp, H., van Genabith, J., and Reyle, U. (2004). Discourse Representation Theory. In Gabbay, D. and Günthner, F., editors, *Handbook of Philosophical Logic*. Kluwer, Dordrecht. (Cited on page 73.)

Kirby, G. H., Lord, M. R., and Rayner, J. D. (1991). Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. 6. (Semi)automatic Name Correction. *J. Chem. Inf. Comput. Sci.*, 31(1):153–160. (Cited on page 20.)

Kirby, G. H. and Polton, D. J. (1993). Systematic Chemical Nomenclatures in the Computer Age. *J. Chem. Inf. Comput. Sci.*, 33:560–563. (Cited on page 73.)

Krauthammer, M. and Nenadić, G. (2004). Term Identification in the Biomedical Literature. *Journal of Biomedical Informatics (Special Issue on Named*

Entity Recognition in Biomedicine), 37(6):512–526. (Cited on pages 1, 66 and 73.)

Larsen, C. (2005). Knowledge Management Strategies for Biologics Research. Retrieved November 2005, from http://www.biopharminternational.com/biopharm/article/articleDetail.jsp?id=162436. (Cited on page 1.)

Lohnstein, H. (1996). *Formale Semantik und natürliche Sprache*. Westdeutscher Verlag, Opladen. (Cited on page 8.)

Luque Ruiz, I., Cruz Soto, J. L., and Gómez-Nieto, M. A. (1996a). Error Detection, Recovery, and Repair in the Translation of Inorganic Nomenclatures. 1. A Study of the Problem. *J. Chem. Inf. Comput. Sci.*, 36:7–15. (Cited on page 73.)

Luque Ruiz, I., Cruz Soto, J. L., and Gómez-Nieto, M. A. (1996b). Error Detection, Recovery, and Repair in the Translation of Inorganic Nomenclatures. 2. A Proposed Strategy. *J. Chem. Inf. Comput. Sci.*, 36:16–24. (Cited on page 73.)

Namer, F. and Zweigenbaum, P. (2004). Acquiring Meaning for French Medical Terminology: Contribution of Morphosemantics. In AMIA, editor, *Proceedings of the Eleventh MEDINFO International Conference*, pages 535–539, San Francisco, CA. (Cited on page 73.)

Perl Foundation (2004). The Perl Programming Language. http://www.perl.org. (Cited on page 27.)

Reyle, U. (2005). Understanding Chemical Terminology. *Terminology*. Retrieved November 2005, from ftp://ftp.ims.uni-stuttgart.de/pub/papers/reyle/terminology.pdf. (Cited on page 3.)

Rojas, I., Bernardi, L., Ratsch, E., Kania, R., Wittig, U., and Šarić, J. (2002). A Database System for the Analysis of Biochemical Pathways. *J. In Silico Biol.*, 2,0007(2):75–86. (Cited on page 67.)

Šarić, J. (2005). *Information Extraction for Biology*. PhD thesis, University of Stuttgart. (Cited on page 73.)

Schmid, T., Lüdeling, A., Säuberlich, B., Heid, U., and Möbius, B. (2001). DeKo: Ein System zur Analyse komplexer Wörter. In *GLDV - Jahrestagung 2001*, pages 49–57. (Cited on page 73.)

Spencer, A. (1991). *Morphological Theory. An Introduction to Word Structure in Generative Grammar*. Basil Blackwell, Oxford. (Cited on page 7.)

Swedish Institute of Computer Science (2001). Sicstus prolog. http://www.sics.se/sicstus. (Cited on page 24.)

Tanabe, L., Scherf, U., Smith, L. H., Lee, J. K., Hunter, L., and Weinstein, J. N. (1999). MedMiner: An Internet Text-mining Tool for Biomedical Information, with Application to Gene Expression Profiling. *BioTechniques*, 27:1210–1217. (Cited on page 73.)

Voss, M. (2004). Improving Upon Earley's Parsing Algorithm in Prolog. Retrieved November 2005, from http://www.ai.uga.edu/mc/ProNTo/Voss.pdf. (Cited on pages 40 and 70.)

Weininger, D. (1988). SMILES, a Chemical Language and Information System. 1. Introduction to Methodology and Encoding Rules. *J. Chem. Inf. Comput. Sci.*, 28:31–36. (Cited on page 17.)

Weininger, D., Weininger, A., and Weininger, J. L. (1989). SMILES. 2. Algorithm for Generation of Unique smiles Notation. *J. Chem. Inf. Comput. Sci.*, 29:97–101. (Cited on page 18.)

Wittig, U., Weidemann, A., Kania, R., Peiss, C., and Rojas, I. (2004). Classification of Chemical Compounds to Support Complex Queries in a Pathway Database. *J. Comp. Funct. Genom.*, 5(2):156–162. (Cited on pages 21, 24 and 62.)

Additional Links

Biochemistry Background

- The Organic Substance Classes, in German:
  http://www.chemieseite.de/organisch
- Virtual Library of Biochemistry and Cell Biology:
  http://www.biochemweb.org/databases.shtml
- CHEMINFO Chemical Information Sources:
  http://www.indiana.edu/~cheminfo

Resources for Life Science Terminology

- Bibliography of IUPAC and IUBMB Nomenclature Recommendations:
  http://www.indiana.edu/~cheminfo/14-03.html
- The Beilstein Institute and Database:
  http://www.beilstein-institut.de
- Compound Synonym Lists:
  http://www.chemicalland21.com,
  http://lb.chemie.uni-hamburg.de
- Unified Medical Language Systems:
  http://www.nlm.nih.gov/research/umls/about_umls.html
- NCBI Organism Taxonomy:
  http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=Taxonomy
- InterPro Database of Protein Families, Domains, etc.:
  http://www.ebi.ac.uk/interpro
- KDBI: Kinetic Data of Bio-molecular Interactions:
  http://xin.cz3.nus.edu.sg/group/kdbi/kdbi.asp
- EBI/BCBI Taxonomy/Synonyms Databases:
  http://www.ebi.ac.uk/msd-srv/docs/dbdoc/ref_taxonomy.html
- Gene Ontology:
  http://www.geneontology.org

Life Science Text Processing Applications

- MedMiner:
  http://discover.nci.nih.gov/textmining
- GenIE (= Genome Information Extraction):
  http://www.ims.uni-stuttgart.de/projekte/GenIE
- GENIA information extraction project:
  http://www-tsujii.is.s.u-tokyo.ac.jp/~genia

A Source Code

A.1 Input and Output

A.1.1 File inout.pl

```
% ------------------------------------------- %
% File:      inout.pl                         %
% Author:    Gerhard Kremer                   %
% Purpose:   input and output processing      %
% ------------------------------------------- %


%% preprocess/0:
% produces query from single typed-in name
preprocess :-
    format("Give␣me␣a␣name:␣",[]),
    flush_output,
    system('perl␣preprocess.perl').
             % s/(\(\))/"$1"/g;\


/*
processes nameslist 'testsuite.txt' (output in 'testsuite.out')
and prints name, semantics, (classes, SMILES strings)
perl-actions: (input: 'testsuite.txt',output: 'testsuite.tmp')
delete whitespaces at end of line, ignore comments( %...),
backslash every single quote,print: '<Name>'. ,
lowercase name, concatenate chars with comma in between,
put special chars (comma, apostrophe, blank,
                   round brackets) in double quotes,
print in prolog-list format: [a,b,c,...]
*/

    system('perl␣-n␣process.perl␣testsuite.txt␣>␣testsuite.tmp'),
    open('testsuite.tmp',read,InStream),
    open('testsuite.out',write,OutStream),
    processNames(InStream,OutStream),
    close(InStream),
    close(OutStream).

%% processNames/2:
/*
queries org_compd(...<Name>...) from InStream
and prints all semantic representations in OutStream
InStream should contain pairs of <Name>, format:
'name'.
[n,a,m,e].
*/

processNames(InStream,OutStream) :-
    read(InStream,Name),
    Name \== end_of_file,     %% not at end of file
    !,
    format(OutStream,"~w~n",Name),format("~w~n",Name),
    read(InStream,Term),
    output(Term,OutStream),
    findall(Sem,org_compd(_Syn,Sem,Term,[]),AllSems),
%    sort(AllSems,AllSemsSorted), % comment to see syntactic ambiguities
    printSolutions(AllSems,OutStream),
    processNames(InStream,OutStream).

% do nothing (if at end_of_file)
processNames(_InStream,_OutStream).


%% printSolutions/2:
% beta-reduce list of semantic terms,
% generate for each SMILES string and
% classes; print them all
printSolutions([Sem1|RestSems],OutStream) :-
    beta_reduce(Sem1,Sem1Reduced),
    output(Sem1Reduced,OutStream),
    smile_list(_SList,SList,Sem1Reduced),
    output_smile(SList,OutStream),
    output_nl(OutStream),
    classes(Sem1Reduced,ClassList),
    output_classes(ClassList,OutStream),
    output(---,OutStream),   % separates alternative solutions
    printSolutions(RestSems,OutStream).

% all done; print additional newline
% (separating names w/ their solutions-)
printSolutions([],OutStream) :-
    output_nl(OutStream).


%% Output of anything but strings
% to both stdout and outputstream
% (~w: pass to write/2)
output(Anything,OutStream) :-
    format("~w~n",[Anything]),
    format(OutStream,"~w~n",[Anything]).

output_nl(OutStream) :-
    format("~n",[]),
    format(OutStream,"~n",[]).

output_no_nl(Anything,OutStream) :-
    format("~w",[Anything]),
    format(OutStream,"~w",[Anything]).

output_flexible(Format,AnythingList,OutStream) :-
    format(Format,AnythingList),
    format(OutStream,Format,AnythingList).


% Output of classes list
output_classes([Class],OutStream) :-
    !,
    output(Class,OutStream).

output_classes([Class|ClassRestList],OutStream) :-
    output_no_nl(Class,OutStream),
    output_no_nl(',',OutStream),
    output_classes(ClassRestList,OutStream).


%% output_smile/2:
% Output of SMILES strings
% change according to smile_output in smiles.pl

% done
output_smile([],_OutStream) :- !.

% output chain element w/ chirality sign given
output_smile(chain_el(El,_Loc,Branches,FList),OutStream) :-
    member(chir(Chir),FList),
    !,
    output_flexible("~w~w~w~w",['[',El,Chir,']'],OutStream),
    output_smile_ring_el1(FList,OutStream),
    output_smile_branches(Branches,FList,OutStream).

% output chain element (w/o chirality sign given)
output_smile(chain_el(El,_Loc,Branches,FList),OutStream) :-
    !,
    output_flexible("~w",[El],OutStream),
    output_smile_ring_el1(FList,OutStream),
    output_smile_branches(Branches,FList,OutStream).

% it's no underspecified SMILES
output_smile([uspecs([])|SList],OutStream) :-
    !,
    output_smile(SList,OutStream).

% output underspecified SMILES
output_smile([uspecs(USpecs)|SList],OutStream) :-
    !,
    output_flexible("~w~w",[underspecified,'('],OutStream),
    output_smile(SList,OutStream),
    output_flexible("~w~w",[',','['],OutStream),
    output_smile_uspecs(USpecs,OutStream),
    output_flexible("~w~w",[']',')'],OutStream).

% output atom (from a list)
output_smile(SmileHead,OutStream) :-
    atomic(SmileHead),
    !,
    output_flexible("~w",[SmileHead],OutStream).

% output list
output_smile([SmileHead|SmileList],OutStream) :-
    output_smile(SmileHead,OutStream),
    output_smile(SmileList,OutStream).

%% output_smile_branches/3:
% output branches

% done
output_smile_branches([],_FList,_OutStream) :- !.

% special rule when encountering ['O']
% (could be involved in ring connection)
output_smile_branches([[ring-El]|Branches],FList,OutStream) :-
    !,
    output_flexible("~w",['('],OutStream),
    output_smile([El],OutStream),
    output_smile_ring_el2(FList,OutStream),
    output_flexible("~w",[')'],OutStream),
    output_smile_branches(Branches,FList,OutStream).

% special rule when encountering
%e.g. [deoxy-'[H]']
output_smile_branches([[_Special-El]|Branches],FList,OutStream) :-
    !,
    output_flexible("~w",['('],OutStream),
    output_smile([El],OutStream),
    output_flexible("~w",[')'],OutStream),
    output_smile_branches(Branches,FList,OutStream).

% output one branch, then the others
output_smile_branches([Branch|Branches],FList,OutStream) :-
    output_flexible("~w",['('],OutStream),
    output_smile(Branch,OutStream),
    output_flexible("~w",[')'],OutStream),
    output_smile_branches(Branches,FList,OutStream).

%% output_smile_ring_el1/2:
% output ring element1 locant (ring index)
% output locant, if there is a ring element
output_smile_ring_el1(FList,OutStream) :-
    member(ring_el1(Loc),FList),
    !,
    output_flexible("~w",[Loc],OutStream).

% nothing to do (there is no ring element)
output_smile_ring_el1(_FList,_OutStream).

%% output_smile_ring_el2/2:
% output ring element2 locant (ring index)
% output locant, if there is a ring element
output_smile_ring_el2(FList,OutStream) :-
    member(ring_el2(Loc),FList),
    !,
    output_flexible("~w",[Loc],OutStream).

% nothing to do (there is no ring element)
output_smile_ring_el2(_FList,_OutStream).

%% output_smile_uspecs/2:
%output uspecs

%output_smile_uspecs([]) :- !.

% output last underspecification
output_smile_uspecs([Mult*Locs-Prefix],OutStream) :-
```

```
    !,
    output_flexible("~w~w~w",[Mult,'*','{'],OutStream),
    output_smile_uspecs(Locs,OutStream),
    output_flexible("~w~w~w",['}','-',Prefix],OutStream).

% output one underspecification, then the others
output_smile_uspecs([Mult*Locs-Prefix|USpecs],OutStream) :-
    !,
    output_flexible("~w~w~w",[Mult,'*','{'],OutStream),
    output_smile_uspecs(Locs,OutStream),
    output_flexible("~w~w~w~w",['}','-',Prefix,','],OutStream),
    output_smile_uspecs(USpecs,OutStream).

% output last locant for underspecification
output_smile_uspecs([Loc],OutStream) :-
    number(Loc),
    !,
    output_flexible("~w",[Loc],OutStream).

% output unknown locants
% (no locants found for which substitution could be made)
output_smile_uspecs(['??'],OutStream) :-
    !,
    output_flexible("~w",['??'],OutStream).

% output locants for underspecification
output_smile_uspecs([Loc|Locs],OutStream) :-
    number(Loc),
    !,
    output_flexible("~w~w",[Loc,','],OutStream),
    output_smile_uspecs(Locs,OutStream).
```

A.1.2 File preprocess.perl

```
# ------------------------------------------- #
# File:     preprocess.perl                   #
# Author:   Gerhard Kremer                    #
# Purpose:  input name preprocessing          #
# ------------------------------------------- #


# preprocessing for inout.pl

$_=<>; chomp; s/(\s+\.?|\.)$//g;
$_=lc; $_=join(",",split(//));
s/\'/\'\\\'\'/g;
s/,,/,\',\'/g;
s/([\(\)␣])/\'$1\'/g;
print"\norg_compd(Syn,Sem,[$_],[]),pp(Syn),beta_reduce(Sem,RSem),smile(RSem),classes(RSem,CList)
     .\n"
```

A.1.3 File process.perl

```
# ------------------------------------------- #
# File:     process.perl                      #
# Author:   Gerhard Kremer                    #
# Purpose:  input name processing             #
# ------------------------------------------- #

# processing of compound name input for inout.pl

# lowercase all; split
# to be treated non-special in Prolog:
#  quotes, commas, parentheses, space characters
# output in Prolo-list-format

s/\s+$//g; next if /^(\%.*|)$/;
$name = $_;
$name =~ s/\'/\\\'/g;
print"\'$name\'.\n";

$_=lc;␣$_=join(",",split(//));
s/\'/\'\\\'\'/g;
s/,,/,\',\'/g;
s/([\(\)␣])/\'$1\'/g;
print"[$_].\n"
```

A.2 Parser Module

A.2.1 File compd_lex.pl

```
% ------------------------------------------- %
% File:      compd_lex.pl                     %
% Author:    Stefanie Anstein, Gerhard Kremer  %
% Purpose:   morpheme lexicon                 %
% ------------------------------------------- %

%%% common lexicon entries

%% special characters
comma(comma(','),listmem_sep) --> [','].
hyph(hyph(-),affix_sep) --> [-].
blank(blank('␣'),word_sep) --> ['␣'].
colon(colon(:),:) --> [:].
apostrophe(apostrophe('\''),'\'') --> ['\''].
leftparenthesis(leftparenthesis('('),'(') --> ['('].
rightparenthesis(rightparenthesis(')'),')') --> [')'].

%% locants
loc(loc(Lex),Sem,All,Rest) :-
    lex(Lex,loc,Sem),
    append(Lex,Rest,All).

%% multipliers
mult(mult(Lex),Sem,All,Rest) :-
    lex(Lex,mult,Sem),
    append(Lex,Rest,All).

%% functional group suffixes
suff_zero(suff_zero(Lex),Sem,All,Rest) :-
    lex(Lex,suff_zero,Sem),
    append(Lex,Rest,All).

%% saturation parent suffix
pns_suff_sat(pns_suff_sat(Lex),Sem,All,Rest) :-
    lex(Lex,pns_suff_sat,Sem),
    append(Lex,Rest,All).

%% unsaturation parent suffix
pns_suff_unsat(pns_suff_unsat(Lex),Sem,All,Rest) :-
    lex(Lex,pns_suff_unsat,Sem),
    append(Lex,Rest,All).

%% trivial ring names
ring_triv(ring_triv(Lex),Sem,All,Rest) :-
    lex(Lex,ring_triv,Sem),
    append(Lex,Rest,All).

%% ring stem suffix
ring_stem_suff(ring_stem_suff(Lex),Sem,All,Rest) :-
    lex(Lex,ring_stem_suff,Sem),
    append(Lex,Rest,All).

%% trivial names for sugar parent structures
ps_triv(ps_triv(Lex),Sem,All,Rest) :-
    lex(Lex,ps_triv,Sem),
    append(Lex,Rest,All).

%[...]

%% locants
lex([1],loc,1).
lex([2],loc,2).
lex([3],loc,3).
%[...]

%% multipliers,
% table 11 (IUPAC 1996)
lex([p,e,n,t,a],mult,5).
lex([h,e,x,a],mult,6).
lex([h,e,p,t,],mult,7).
%[...]
```

```
%% functional group suffixes,
% table 5 (IUPAC 1996)
lex([o,l],suff_zero,ol).
lex([o,n,e],suff_zero,one).
lex([a,l,d,e,h,y,d,e],suff_zero,aldehyde).
%[...]


%%% nonsugar lexicon entries

%% a-terms/replacement prefixes
lex([o,x,a],pref_zero,oxa).
lex([p,h,o,s,p,h,a],pref_zero,phospha).
lex([h,y,d,r,o,x,y],pref_zero,hydroxy).
%[...]

%% monomers
lex([o,x],monomer,'O').
lex([p,h,o,s,p,h],monomer,'P').
lex([a,r,s],monomer,'As').
%[...]

%% saturation parent suffix
lex([a,n],pns_suff_sat,_).
lex([a,n,e],pns_suff_sat,lam(X,lam(Y,ane(X*Y)))).

%% unsaturation parent suffix
lex([e,n,e],pns_suff_unsat,lam(X,lam(Y,ene(X,Y)))).
lex([y,n,e],pns_suff_unsat,lam(X,lam(Y,yne(X,Y)))).
%[...]

%% trivial names
lex([b,e,n,z,e,n,e],pns_triv,benzene).
lex([t,h,i,o,p,h,e,n,e],pns_triv,thiophene).
lex([f,u,r,a,n],pns_triv,furan).
%[...]

%% class names
lex([a,l,k,e,n,e],pns_class,alkene).
lex([a,l,c,o,h,o,l],pns_class,alcohol).
lex([a,c,i,d],pns_class,acid).
%[...]

%% element names
lex([o,x,y,g,e,n],pns_elem,oxygen).
lex([h,y,d,r,o,g,e,n],pns_elem,hydrogen).

%% cyclo- prefix
lex([c,y,c,l,o],cyc_pref,lam(X,cyclo('??'*['??'],X))).

%% stereo prefixes
lex([c,i,s],stereo_pref,lam(X,cis(X))).
lex(['(',z,')'],stereo_pref,lam(X,stereo:z(X))).
%[...]


%%% sugar lexicon entries

%% prefixes
lex([d,e,o,x,y],pref_zero,deoxy).
lex([a,m,i,n,o],pref_zero,amino).
lex([t,h,i,o],pref_zero,thio).
%[...]

%% element prefixes
lex([c],pref_elem,'C').
lex([o],pref_elem,'O').

%% trivial ring names
lex([o,x,i,r,o,s,e],ring_triv,lam(X,lam(Y,anose(X,3,Y)))).
lex([o,x,e,t,o,s,e],ring_triv,lam(X,lam(Y,anose(X,4,Y)))).
lex([f,u,r,a,n,o,s,e],ring_triv,lam(X,lam(Y,anose(X,5,Y)))).
%[...]

%% ring stem suffix
```

```
lex([a,n,o,s,e],ring_stem_suff,lam(Z,lam(X,lam(Y,anose(X,Z,Y))))).
lex([a,n,o,s],ring_stem_suff,lam(Z,lam(X,lam(Y,anose(X,Z,Y))))).

%% parent structure stem suffixes
lex([o,s,e],stem_suff,lam(X,lam(Y,ose(X,Y)))).
lex([o,s],stem_suff,lam(X,lam(Y,ose(X,Y)))).
lex([a,l,d,o,s,e],stem_suff,lam(X,lam(Y,ose(X,Y)))).
%[...]

%% parent structure stem prefixes
lex([a,l,d,o,s,e],stem_pref,ose).
lex([a,l,d,o],stem_pref,ose).
lex([k,e,t,o],stem_pref,ketose).
%[...]

%% configurational symbols
lex([d],cfg_symb,'D').
lex([l],cfg_symb,'L').
lex([d,l],cfg_symb,'DL').
%[...]

%% anomeric configurational symbols
lex([a,l,p,h,a],cfg_symb_anom,alpha).
lex([b,e,t,a],cfg_symb_anom,beta).
%[...]

%% configurational prefixes
lex([g,l,y,c,e,r,o],cfg_pref,glycero).
lex([e,r,y,t,h,r,o],cfg_pref,erythro).
lex([t,h,r,e,o],cfg_pref,threo).
%[...]

%% trivial names
lex([e,r,y,t,h,r,o,s,e],ps_triv,erythrose).
lex([t,h,r,e,o,s,e],ps_triv,threose).
lex([r,i,b,o,s,e],ps_triv,ribose).
%[...]

%% class names for sugar parent structures
lex([k,e,t,o,s,e],ps_class,ketose).
```

A.2.2 File compd.pl

```
% ------------------------------------------- %
% File:      compd.pl                          %
% Author:    Stefanie Anstein, Gerhard Kremer  %
% Purpose:   organic compound grammar          %
% ------------------------------------------- %

:- use_module(library(system)). % system utilities (shell commands)


%% org_compd/4: organic compound (root rule)

% parent structure w/o affix(es),
% e.g. hexane
org_compd(org_compd(Syn_parent),
          compd(Sem_parent,pref([]),suff([]))
         ) -->
    parent(Syn_parent,Sem_parent).

% parent w/ suffix(es),
% e.g. methanol
org_compd(org_compd(Syn_parent,Syn_suffs),
          compd(Sem_parent,pref([]),suff(Sem_suffs))
         ) -->
    parent(Syn_parent,Sem_parent),
    suffs(Syn_suffs,Sem_suffs).

% parent w/ both prefix(es) and suffix(es),
% e.g. 2-oxahexanol
org_compd(org_compd(Syn_prefs,Syn_parent,Syn_suffs),
          compd(Sem_parent,pref(Sem_prefs),suff(Sem_suffs))
         )   -->
    prefs(Syn_prefs,Sem_prefs),
    parent(Syn_parent,Sem_parent),
    suffs(Syn_suffs,Sem_suffs).

% parent w/ prefix(es),
% e.g. 2,4,8-trioxaundecane, 1-methyl-pent-2-ulose
org_compd(org_compd(Syn_prefs,Syn_parent),
          compd(Sem_parent,pref(Sem_prefs),suff([]))
         ) -->
    prefs(Syn_prefs,Sem_prefs),
    parent(Syn_parent,Sem_parent).

% 'additive names formed by the use of a separate word',
% ('formerly called radiofunctional nomenclature'),
% R-1.2.3.3.2, R-5.6.2.1,
% e.g. methyl alcohol, ethyl methyl ketone
org_compd(org_compd(Syn_rads,Syn_pns_class),
          compd(Sem_rads+Sem_pns_class,pref([]),suff([]))
         ) -->
    rads(Syn_rads,Sem_rads),
    pns_class(Syn_pns_class,Sem_pns_class).

% one radical ...
rads(rads(Syn_rad),
     Sem_rad
    ) -->
    rad(Syn_rad,Sem_rad).

% ... or several radicals
rads(rads(Syn_rad,Syn_rads),
     Sem_rad+Sem_rads
    ) -->
    rad(Syn_rad,Sem_rad),
    rads(Syn_rads,Sem_rads).

% for efficiency reasons not org_compd,
% but only parent (most frequent case) allowed
rad(rad(Syn_parent,Syn_blank),
    compd(Sem_parent,pref([]),suff([]))
   ) -->
    parent(Syn_parent,Sem_parent),
    blank(Syn_blank,_Sem_blank).
```

```
%% prefs/4: prefixes

% single prefix or last prefix w/ hyphen,
% e.g. 1-methyl- in 1-methyl-pent-2-ulose
prefs(prefs(Syn_pref,Syn_hyph),
      [Sem_pref]
     ) -->
    pref(Syn_pref,Sem_pref),
    hyph(Syn_hyph,_).

% single prefix / last prefix w/o hyphen,
% e.g. 2,4,8-trioxa in 2,4,8-trioxaundecane
prefs(prefs(Syn_pref),
      [Sem_pref]
     ) -->
    pref(Syn_pref,Sem_pref).

% several prefixes,
% e.g. 2-oxa-3-phospha in 2-oxa-3-phosphahexane
prefs(prefs(Syn_pref,Syn_hyph,Syn_prefs),
      [Sem_pref|Sem_prefs]
     ) -->
    pref(Syn_pref,Sem_pref),
    hyph(Syn_hyph,_),
    prefs(Syn_prefs,Sem_prefs).


%% one prefix

% configurational prefix,
% e.g. alpha-D-threo
pref(pref(Syn_cfg),
     Sem_cfg
     ) -->
    cfg(Syn_cfg,Sem_cfg).

% prefix w/o locs and/or mult,
% R-2.3.3.2,
% e.g. oxa in oxacyclohexane
pref(pref(Syn_pref_zero),
     ('??'*['??']-Sem_pref_zero)
    ) -->
    pref_zero(Syn_pref_zero,Sem_pref_zero),
    !.

% prefix w/ locs and/or mult,
% R-1.2.2,
% e.g. 1,2-dioxa in 1,2-dioxahexane, 1-oxa in 1-oxahexane
pref(pref(Syn_locs_mult,Syn_pref_zero),
     Sem_locs_mult-Sem_pref_zero
    ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    pref_zero(Syn_pref_zero,Sem_pref_zero),
    !.

% substitution/replacement prefix,
% e.g. 2-C-amino
pref(pref(Syn_locs_mult,Syn_hyph,Syn_pref_elem,Syn_hyph,Syn_pref_zero),
     Sem_locs_mult-Sem_pref_elem-Sem_pref_zero
    ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    hyph(Syn_hyph,_),
    pref_elem(Syn_pref_elem,Sem_pref_elem),
    hyph(Syn_hyph,_),
    pref_zero(Syn_pref_zero,Sem_pref_zero).

% recursive call of org_compd for nested compds w/ parentheses,
% e.g. 1,2-dimethyl- in 1,2-dimethyl-hex-3-ulose,
% 1-methyl- in 1-methyl-pent-2-ulose
pref(pref(Syn_locs_mult,Syn_leftparenthesis,Syn_org_compd,Syn_rightparenthesis),
     Sem_locs_mult-Sem_org_compd
    ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    leftparenthesis(Syn_leftparenthesis,_),
    !,
    org_compd(Syn_org_compd,Sem_org_compd),
    rightparenthesis(Syn_rightparenthesis,_).

% recursive call of org_compd for nested compds,
% --- last rule ---
% e.g. 1,2-dimethyl- in 1,2-dimethyl-hex-3-ulose,
% 1-methyl- in 1-methyl-pent-2-ulose
pref(pref(Syn_locs_mult,Syn_org_compd),
     Sem_locs_mult-Sem_org_compd
    ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    org_compd(Syn_org_compd,Sem_org_compd),
    !.


%% parent/4: parent structures (nonsugar or sugar)

% nonsugar parent structure,
% e.g. hexane
parent(Syn_par_phr_nonsugar,
       Sem_par_phr_nonsugar
      ) -->
    par_phr_nonsugar(Syn_par_phr_nonsugar,Sem_par_phr_nonsugar).

% nonsugar parent structure with 'infix'/double suffix,
% e.g. 3-penten-1-yne
parent(Syn_infix_par_phr_nonsugar,
       Sem_infix_par_phr_nonsugar
      ) -->
    infix_par_phr_nonsugar(Syn_infix_par_phr_nonsugar,Sem_infix_par_phr_nonsugar).

% sugar parent phrase structure,
% e.g. hexose
parent(Syn_par_sugar,
       Sem_par_sugar
      ) -->
    par_sugar(Syn_par_sugar,Sem_par_sugar).


%% suffs/4: suffixes

% with preceding hyphen
suffs(suffs(Syn_hyph,Syn_suffs),
     Sem_suffs) -->
    hyph(Syn_hyph,_),
    !,
    suffs(Syn_suffs,Sem_suffs).

% one (functional group) suffix
% e.g. ol in hexanol
suffs(suffs(Syn_suff),
      [Sem_suff]
     ) -->
    suff(Syn_suff,Sem_suff).

% several (functional group) suffixes w/o locs_mult,
% R-5.8.3,
% e.g. olate in methanolate
suffs(suffs(Syn_suff,Syn_suffs),
      [Sem_suff|Sem_suffs]
     ) -->
    suff(Syn_suff,Sem_suff),
    suffs(Syn_suffs,Sem_suffs).

% (functional group) suffix w/ locs_mult,
% e.g. 2,3-diol in hexane-2,3-diol
suff(suff(Syn_locs_mult,Syn_suff_zero),
      Sem_locs_mult-Sem_suff_zero
     ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    suff_zero(Syn_suff_zero,Sem_suff_zero).

% (functional group) suffix w/o locs_mult,
% e.g. ol in ethanol
suff(suff(Syn_suff_zero),
     ('??'*['??']-Sem_suff_zero)
     ) -->
     suff_zero(Syn_suff_zero,Sem_suff_zero).

% complex (functional group) suffix(es) w/o locs_mult,
% e.g. 'oic anhydride' in 'pentanoic anhydride'
suff(suff(Syn_suff_adj),
     ('??'*['??']-Sem_suff_adj)
     ) -->
     suff_adj(Syn_suff_adj,Sem_suff_adj).

suff_adj(suff_adj(Syn_adj_suff,
                  Syn_blank,
                  Syn_pns_class),
         Sem_adj_suff+Sem_pns_class
        ) -->
    adj_suff(Syn_adj_suff,Sem_adj_suff),
    blank(Syn_blank,_Sem_blank),
    pns_class(Syn_pns_class,Sem_pns_class).
```

A.2.3 File compd_common.pl

```
% ------------------------------------------ %
% File:      compd_common.pl                 %
% Author:    Stefanie Anstein, Gerhard Kremer %
% Purpose:   common predicates                %
% ------------------------------------------ %


%%% grammar

%% locs_mult/4: locs-mult affix,
% e.g. ...-3,4-di<affix>
% => Sem: affix(2*[3,4],...)

% only multiplier specified
locs_mult(locs_mult(Syn_mult),
         Sem_mult*['??']
         ) -->
    mult(Syn_mult,Sem_mult),
    !.

% locants and multiplier specified
locs_mult(locs_mult(Syn_locs,Syn_mult),
         Sem_mult*Sem_locs
         ) -->
    locs(Syn_locs,Sem_locs),
    mult(Syn_mult,Sem_mult).

% only locants specified
locs_mult(locs_mult(Syn_locs,Sem_locs),
         '??'*Sem_locs
          ) -->
    locs(Syn_locs,Sem_locs).


%% locs/4: locants

% single or last locant,
% e.g. 8- in 2,4,8-trioxaundecane
locs(locs(Syn_loc,Syn_hyph),
     [Sem_loc]) -->
    loc(Syn_loc,Sem_loc),
    hyph(Syn_hyph,_),
    !. % if hyphen after locant, cut other rules

% locant(s) as infix (hyphen also at beginning),
% e.g. -2- in pent-2-ene
locs(locs(Syn_hyph,Syn_locs),
     Sem_locs) -->
    hyph(Syn_hyph,_),
    !,  % if hyphen is first, no need to try other rules
    locs(Syn_locs,Sem_locs).

% several locants,
% e.g. 2,4,8 in 2,4,8-trioxaundecane
locs(locs(Syn_loc,Syn_comma,Syn_locs),
     [Sem_loc|Sem_locs]) -->
    loc(Syn_loc,Sem_loc),
    comma(Syn_comma,_),
    !,  % after comma, this is the only matching rule
    locs(Syn_locs,Sem_locs).

% locant grouping w/ colons,
% e.g. 1,2:3,4- in ring specification
locs(locs(Syn_loc,Syn_colon,Syn_locs),
     [Sem_loc|Sem_locs]) -->
    loc(Syn_loc,Sem_loc),
    colon(Syn_colon,_),
    !, % after colon found, this is the only matching rule


%% pp/1: pretty print a prolog structure
% call: pp(PrologStructure)
pp(P) :- pp([],P).
pp(L,P) :- atomic(P), !, indented_print(L,P).
pp(L,P) :- var(P), !, indented_print(L,P).
pp(L,[H|T]) :- atomic(H), !, indented_print(L,[H|T]).
pp(L,[H|T]) :- indented_print(L,'['), pp(['␣␣|'|L],H),
    ppa(['|'|L],T), indented_print(L,']').
pp(L,P) :- P=..[F|Arg], !, pp(L,F), ppa(['␣␣|'|L], Arg).

ppa(L,[H|T]) :- !, pp(L,H), ppa(L,T).
ppa(_,[]) :- true.

indented_print([H|T], P) :- !, write(H), indented_print(T, P).
indented_print([],P) :- is_list(P), !, write('=['), printm(P), write(']'), nl.
indented_print([],P) :- write('='), write(P), nl.

printm([]) :- !, write('␣').
printm([H|T]) :- write('␣'), write(H), printm(T).


%% beta-reduce a semantic representation
% call: beta_reduce(Sem_Rep,Reduced_Sem_Rep)
% use lam(Var,Formula) vor lambda-expression,
% Formula@Arg for functional application

beta_reduce(F,Reduced) :- atom(F), !, Reduced = F.
beta_reduce(F,Reduced) :- var(F), !, Reduced = F.

beta_reduce('@'(lam(X,F),Y),Reduced) :- !,
    beta_reduce(Y,YR), YR=X, beta_reduce(F,Reduced).

beta_reduce('@'(F,X),Reduced) :- !,
    beta_reduce(F,FR), beta_reduce(X,XR),
    beta_reduce('@'(FR,XR),Reduced).

beta_reduce(Term,Reduced) :-
        Term =.. [Functor|Args],
        reduce_args(Args,RArgs),
        Reduced =.. [Functor|RArgs].

reduce_args([],[]) :- !.
reduce_args([Head|Rest],[RHead|RRest]) :-
        beta_reduce(Head,RHead),
        reduce_args(Rest,RRest).
```

A.2.4 File parent_nonsugar.pl

```
% ------------------------------------------ %
% File:     parent_nonsugar.pl               %
% Author:   Stefanie Anstein                 %
% Purpose:  nonsugar parent grammar          %
% ------------------------------------------ %

%% nonsugar parent phrases = parent structures with modifications

% saturated 'default' parent structure,
% e.g. pentaoxane
par_phr_nonsugar(par_phr_nonsugar(Syn_par_nonsugar),
                 Sem_par_nonsugar
                ) -->
    par_nonsugar(Syn_par_nonsugar,Sem_par_nonsugar).

% unsaturated par str w/ locants,
% e.g. 2-pentene
par_phr_nonsugar(par_phr_nonsugar(Syn_locs_mult,
                                  Syn_par_base,
                                  Syn_pns_suffs_unsat),
                 (Sem_pns_suffs_unsat@Sem_locs_mult)@(lam(X,ane(X*'C'))@Sem_par_base)
                ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    par_base(Syn_par_base,Sem_par_base),
    pns_suffs_unsat(Syn_pns_suffs_unsat,Sem_pns_suffs_unsat).

% unsaturated par str w/o locants,
% e.g. pentene meaning 1-pentene
par_phr_nonsugar(par_phr_nonsugar(Syn_par_base,
                                  Syn_pns_suffs_unsat),
                 (Sem_pns_suffs_unsat@'??'*['??'])@(lam(X,ane(X*'C'))@Sem_par_base)
                ) -->
    par_base(Syn_par_base,Sem_par_base),
    pns_suffs_unsat(Syn_pns_suffs_unsat,Sem_pns_suffs_unsat).

% class names w/ locants,
% e.g. 2-alkene
par_phr_nonsugar(par_phr_nonsugar(Syn_locs_mult,
                                  Syn_pns_class),
                 (Sem_locs_mult,Sem_pns_class)
                ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    pns_class(Syn_pns_class,Sem_pns_class).

% unsaturated par str w/ locant(s and mult),
% e.g. pent-2-ene, pent-2,3-diene
par_phr_nonsugar(par_phr_nonsugar(Syn_par_base,
                                  Syn_locs_mult,
                                  Syn_pns_suffs_unsat),
                 (Sem_pns_suffs_unsat@Sem_locs_mult)@(lam(X,ane(X*'C'))@Sem_par_base)
                ) -->
    par_base(Syn_par_base,Sem_par_base),
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    pns_suffs_unsat(Syn_pns_suffs_unsat,Sem_pns_suffs_unsat).

% cyclic structure,
% R-2.3.1.1, R-2.3.2,
% e.g. cyclopentane
par_phr_nonsugar(par_phr_nonsugar(Syn_cyc_pref,Syn_par_phr_nonsugar),
                 Sem_cyc_pref@Sem_par_phr_nonsugar
                ) -->
    cyc_pref(Syn_cyc_pref,Sem_cyc_pref),
    par_phr_nonsugar(Syn_par_phr_nonsugar,Sem_par_phr_nonsugar).

% stereochemical specification,
% R-7,
% e.g. trans-but-2-ene, (E)-but-2-ene
par_phr_nonsugar(par_phr_nonsugar(Syn_stereo_pref,Syn_hyph,Syn_par_phr_nonsugar),
                 Sem_stereo_pref@Sem_par_phr_nonsugar
                ) -->
    stereo_pref(Syn_stereo_pref,Sem_stereo_pref),
    hyph(Syn_hyph,_Sem_hyph),
    par_phr_nonsugar(Syn_par_phr_nonsugar,Sem_par_phr_nonsugar).
```

```
% 'additional suffix',
% one unsaturation suffix added to an unsaturated parent phrase,
% R-3.1.4,
% e.g. -1-yne in 3-penten-1-yne or in pent-3-en-1-yne
infix_par_phr_nonsugar(infix_par_phr_nonsugar(Syn_par_phr_nonsugar,
                                  Syn_locs_mult,
                                  Syn_pns_suff_unsat),
                      (Sem_pns_suff_unsat@Sem_locs_mult)@Sem_par_phr_nonsugar
                      ) -->
    par_phr_nonsugar(Syn_par_phr_nonsugar,Sem_par_phr_nonsugar),
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    pns_suff_unsat(Syn_pns_suff_unsat,Sem_pns_suff_unsat),
    !.


%% nonsugar parent structures

% trivial name,
% e.g. benzene
par_nonsugar(par_nonsugar(Syn_pns_triv),
             Sem_pns_triv
            ) -->
    pns_triv(Syn_pns_triv,Sem_pns_triv),
    !.

% class name,
% e.g. alkene
par_nonsugar(par_nonsugar(Syn_pns_class),
             Sem_pns_class
            ) -->
    pns_class(Syn_pns_class,Sem_pns_class),
    !.

% element name,
% e.g. oxygen
par_nonsugar(par_nonsugar(Syn_pns_elem),
             Sem_pns_elem
            ) -->
    pns_elem(Syn_pns_elem,Sem_pns_elem),
    !.

% mononuclear hydrides,
% table 2,
% e.g. borane
par_nonsugar(par_nonsugar(Syn_monomer,Syn_pns_suff_sat),
             lam(X,ane(1*X))@Sem_monomer
            ) -->
    monomer(Syn_monomer,Sem_monomer),
    pns_suff_sat(Syn_pns_suff_sat,_Sem_pns_suff_sat).

% saturated 'default' par str; operation: C-replacement,
% R-2.2.2,
% e.g. pentaoxane
par_nonsugar(par_nonsugar(Syn_mult,
                          Syn_monomer,
                          Syn_pns_suff_sat),
             Sem_pns_suff_sat@Sem_mult@Sem_monomer
            ) -->
    mult(Syn_mult,Sem_mult),
    monomer(Syn_monomer,Sem_monomer),
    pns_suff_sat(Syn_pns_suff_sat,Sem_pns_suff_sat).

% saturated 'default' par str;
% operation: C-replacement, alternating elements,
% R-2.2.3.2,
% e.g. tetraarsazane
par_nonsugar(par_nonsugar(Syn_mult,
                          Syn_par_base,
                          Syn_pns_suff_sat),
             Sem_pns_suff_sat@Sem_mult@Sem_par_base
            ) -->
    mult(Syn_mult,Sem_mult),
    par_base(Syn_par_base,Sem_par_base),
    pns_suff_sat(Syn_pns_suff_sat,Sem_pns_suff_sat).
```

```
% locs to precede, pns_suffs_unsat to follow,
% e.g. pentaox in 2-pentaoxene
par_nonsugar(par_nonsugar(Syn_mult,
                          Syn_monomer),
             lam(X,lam(Y,ane(X*Y)))@Sem_mult@Sem_monomer
            ) -->
    mult(Syn_mult,Sem_mult),
    monomer(Syn_monomer,Sem_monomer).

% simplest acyclic hydrocarbons,
% R-2.2.1, table 11,
% e.g. hexane
par_nonsugar(par_nonsugar(Syn_mult,Syn_pns_suff_sat),
             lam(X,ane(X*'C'))@Sem_mult
            ) -->
    mult(Syn_mult,Sem_mult),
    pns_suff_sat(Syn_pns_suff_sat,_Sem_pns_suff_sat).


%% parent bases

% either monomers
par_base(par_base(Syn_monomers),
         Sem_monomers
        ) -->
    monomers(Syn_monomers,Sem_monomers),
    !.

% or multipliers
par_base(par_base(Syn_mult),
         Sem_mult
        ) -->
    mult(Syn_mult,Sem_mult),
    !.

% or trivial names
par_base(par_base(Syn_pns_triv),
         Sem_pns_triv
                ) -->
    pns_triv(Syn_pns_triv,Sem_pns_triv),
    !.


%% single molecules/monomers

% two monomers, e.g. arsaz in tetraarsazane
monomers(monomers(Syn_monomer_1,
                  Syn_monomer_2),
         Sem_monomer_1+Sem_monomer_2
        ) -->
    monomer(Syn_monomer_1,Sem_monomer_1),
    monomer(Syn_monomer_2,Sem_monomer_2).


%% nonsugar unsaturation parent suffixes (non-recursive)

% one unsaturation suffix,
% e.g. ene in 2-pentene
pns_suffs_unsat(pns_suffs_unsat(Syn_pns_suff_unsat),
                Sem_pns_suff_unsat
               ) -->
    pns_suff_unsat(Syn_pns_suff_unsat,Sem_pns_suff_unsat).
```

A.2.5 File parent_sugar.pl

```
% ------------------------------------------- %
% File:     parent_sugar.pl                   %
% Author:   Gerhard Kremer                    %
% Purpose:  sugar parent grammar              %
% ------------------------------------------- %


%% parent structures (sugar,carbohydrate)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% parent phrase


%% par_sugar/4: parent (sugar)

% trivial parent structure name,
% e.g. D-Glucose, beta-L-Fructose
par_sugar(par_sugar(Syn_ps_triv),
          triv_name(Sem_ps_triv)
         ) -->
    ps_triv(Syn_ps_triv,Sem_ps_triv),
    !.

% trivial parent (ketose-)prefix and ring,
% e.g. D-Fructopyranose, D-Glucopyranose
par_sugar(par_sugar(Syn_ps_triv_pref,Syn_ring_stem),
          Sem_ring_stem@triv_name(Sem_ps_triv_pref)
         ) -->
    ps_triv_pref(Syn_ps_triv_pref,Sem_ps_triv_pref),
    ring_stem(Syn_ring_stem,Sem_ring_stem),
    !.

% parent class name
par_sugar(par_sugar(Syn_ps_class),
          class_name(Sem_ps_class)
         ) -->
    ps_class(Syn_ps_class,Sem_ps_class),
    !.

% anosuloses, (as w/ only ring stem suffix default aldose)
% e.g. hexopyranos-4-ulose
par_sugar(par_sugar(Syn_mult,Syn_ring_stem,Syn_ps_zero),
          Sem_ps_zero@(Sem_ring_stem@(ose('??'*['??'],Sem_mult*'C')))
          ) -->
    mult(Syn_mult,Sem_mult),
    ring_stem(Syn_ring_stem,Sem_ring_stem),
    ps_zero(Syn_ps_zero,Sem_ps_zero),
    !.

% only ring stem suffix (-> default: aldose),
% e.g. hexopyranose
par_sugar(par_sugar(Syn_mult,Syn_ring_stem),  %% later --> par-sugar-class ?
          Sem_ring_stem@(ose('??'*['??'],Sem_mult*'C'))
         ) -->
    mult(Syn_mult,Sem_mult),
    ring_stem(Syn_ring_stem,Sem_ring_stem),
    !.

% 2 ps_zeroes (ketoaldoses)
% e.g. hexos-3-ulose
% [2-Carb-12]
par_sugar(par_sugar(Syn_mult,Syn_ps_zero,Syn_ps_zero_bar),
          Sem_ps_zero_bar@(Sem_ps_zero@Sem_mult*'C')
         ) -->
    mult(Syn_mult,Sem_mult),
    ps_zero(Syn_ps_zero,Sem_ps_zero),
    ps_zero(Syn_ps_zero_bar,Sem_ps_zero_bar),
    !.

% aldoses, ketoses,
% e.g. hexose, hex-3-ulose
par_sugar(par_sugar(Syn_mult,Syn_ps_zero),
          Sem_ps_zero@(Sem_mult*'C')
         ) -->
    mult(Syn_mult,Sem_mult),
    ps_zero(Syn_ps_zero,Sem_ps_zero),
    !.

% ketoses w/ loc-prefix (CAS nomenclature):
% locants before (C-chain-)multiplier
% e.g. 3-pentulose
par_sugar(par_sugar(Syn_locs,Syn_mult,Syn_stem_suff),
          (Sem_stem_suff@'??'*Sem_locs)@(Sem_mult*'C')
         ) -->
    locs(Syn_locs,Sem_locs),
    mult(Syn_mult,Sem_mult),
    stem_suff(Syn_stem_suff,Sem_stem_suff).


%% stem suffixes, possibly locants(,mult,ring suffixes)

%% ps_zero/4:
% basic parent elements (sugar)

% aldoses, ketoses w/o locs_mult
% (no explicit loc for C=O - group),
% e.g. ose, ulose
% [2-Carb-8]
ps_zero(ps_zero(Syn_stem_suff),
       Sem_stem_suff@'??'*['??']
       ) -->
    stem_suff(Syn_stem_suff,Sem_stem_suff),
    !.

% several ring suffixes and stem
% multi-ketoses w/ (possibly more than 1) ring suffix,
% e.g. -2,3-diulo-2,5-furanose
% [2-Carb-9],[2-Carb-10]
ps_zero(ps_zero(Syn_locs_mult,Syn_stem_suff,Syn_ring_stems),
       lam(X,Sem_ring_stems@((Sem_stem_suff@Sem_locs_mult)@X))
       ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    stem_suff(Syn_stem_suff,Sem_stem_suff),
    ring_stems(Syn_ring_stems,Sem_ring_stems),
    !.

% multi-aldoses,-ketoses w/ loc-infix,
% e.g. hexodialdose, hexo-2,3-diulose
% [2-Carb-9],[2-Carb-10]
ps_zero(ps_zero(Syn_locs_mult,Syn_stem_suff),
       Sem_stem_suff@Sem_locs_mult
       ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    stem_suff(Syn_stem_suff,Sem_stem_suff).


% [2-Carb-5]

% only 1 ring stem
ring_stems(ring_stems(Syn_ring_stem),
           Sem_ring_stem
          ) -->
    ring_stem(Syn_ring_stem,Sem_ring_stem).

% more than 1 ring stem
ring_stems(ring_stems(Syn_ring_stem,Syn_ring_stems),
           lam(X,Sem_ring_stem@(Sem_ring_stems@X))) -->
    ring_stem(Syn_ring_stem,Sem_ring_stem),
    ring_stems(Syn_ring_stems,Sem_ring_stems).

% ring stem can be either trivial... (up to size 7)
% ... or systematically constructed:

% trivial,
% e.g. pyranose
ring_stem(ring_stem(Syn_ring_triv),
          Sem_ring_triv@'??'*['??']
         ) -->
    ring_triv(Syn_ring_triv,Sem_ring_triv),
    !.

% systematic,
% e.g. octanose
% (ring_stem_suff: only 'anose')
ring_stem(ring_stem(Syn_mult,Syn_ring_stem_suff),
          (Sem_ring_stem_suff@Sem_mult)@'??'*['??']
         ) -->
    mult(Syn_mult,Sem_mult),
    ring_stem_suff(Syn_ring_stem_suff,Sem_ring_stem_suff),
    !.

% trivial w/ mult or/and locs,
% e.g. 2,3:5,6-dioxirose
ring_stem(ring_stem(Syn_locs_mult,Syn_ring_triv),
          Sem_ring_triv@Sem_locs_mult
         ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    ring_triv(Syn_ring_triv,Sem_ring_triv),
    !.

% systematic w/ mult or/and locs,
% e.g. 2,8-octanose
ring_stem(ring_stem(Syn_locs_mult,Syn_mult,Syn_ring_stem_suff),
          (Sem_ring_stem_suff@Sem_mult)@Sem_locs_mult
         ) -->
    locs_mult(Syn_locs_mult,Sem_locs_mult),
    mult(Syn_mult,Sem_mult),
    ring_stem_suff(Syn_ring_stem_suff,Sem_ring_stem_suff).



% configuration specifications root rule
% (rule for adding more stereochem. symbols)
 cfg(cfg(Syn_cfg_prefs),
    cfg(Sem_cfg_prefs)) -->
    cfg_prefs(Syn_cfg_prefs,Sem_cfg_prefs).

%% cfg_prefs/4: configurational prefixes
% (desired is the longest group;
%  root rule may contain several prefs)

% more than 1 config.prefix w/ config.symbols
cfg_prefs(cfg_prefs(Syn_cfg_symbols,Syn_hyph,Syn_cfg_pref,Syn_hyph,Syn_cfg_prefs),
          [Sem_cfg_symbols-Sem_cfg_pref|Sem_cfg_prefs]) -->
    cfg_symbols(Syn_cfg_symbols,Sem_cfg_symbols),
    hyph(Syn_hyph,_),
    cfg_pref(Syn_cfg_pref,Sem_cfg_pref),
    hyph(Syn_hyph,_),
    cfg_prefs(Syn_cfg_prefs,Sem_cfg_prefs),
    !.

% 1 config.prefix w/ config.symbols
% [2-Carb-4.3]
cfg_prefs(cfg_prefs(Syn_cfg_symbols,Syn_hyph,Syn_cfg_pref),
          [Sem_cfg_symbols-Sem_cfg_pref]) -->
    cfg_symbols(Syn_cfg_symbols,Sem_cfg_symbols),
    hyph(Syn_hyph,_),
    cfg_pref(Syn_cfg_pref,Sem_cfg_pref).

% config.symbol(s) only
cfg_prefs(cfg_prefs(Syn_cfg_symbols),
          [Sem_cfg_symbols]) -->
    cfg_symbols(Syn_cfg_symbols,Sem_cfg_symbols).


%% cfg_symbols/4:
% (anomeric) configurational symbols

% only config.symbol,
% e.g. D-
% [2-Carb-4.1]
cfg_symbols(cfg_symbols(Syn_cfg_symb),
         Sem_cfg_symb
        ) -->
    cfg_symb(Syn_cfg_symb,Sem_cfg_symb),
    !.
```

```
% anom.config.symbol and config.symbol,
% e.g. beta-D-
% [2-Carb-6]
cfg_symbols(cfg_symbols(Syn_cfg_symb_anom,Syn_hyph,Syn_cfg_symb),
         Sem_cfg_symb_anom-Sem_cfg_symb
        ) -->
    cfg_symb_anom(Syn_cfg_symb_anom,Sem_cfg_symb_anom),
    hyph(Syn_hyph,_),
    cfg_symb(Syn_cfg_symb,Sem_cfg_symb).
```

A.3 SMILES Module

A.3.1 File smiles.pl

```
% ------------------------------------------- %
% File:      smiles.pl                        %
% Author:    Gerhard Kremer                   %
% Purpose:   SMILES string generator          %
% ------------------------------------------- %


% call: smile_list(_UninstantiatedList,SMILESList,SemRep)
% recursive; builds the list from innermost predicate

% format Mult*Element; innermost structure:
% generate skeleton structure
smile_list(SList,SList,Mult*El) :-
    !,
    number(Mult),
    atom(El),
    gen_skeleton(Mult,El,SList).

% trivial name (...part); innermost structure:
% look up smiles string in lexicon
smile_list(SList,SList,triv_name(TrivName)) :-
    !,
    lex_triv(TrivName,_SystematicName,Sem),
    smile_list(_SList,SList,Sem).
%...or for the case, Sem is not in the lexicon, yet:
% generate it (here or
%  just after parsing the compd-name which includes a triv_name)
% lex_triv-syst(TrivialName,SystematicName),
% name-to-list(SystematicName,Name), %%% as in process,preprocess
% org_compd(_,Sem,Name,[]),beta_reduce(Sem,RSem).

% Operator w/ 1 argument
% (e.g. ane)   %%%% TODO
smile_list(SList,SListNew,Expr) :-
    Expr =.. [Functor|[Arg]],
    Functor == ane,  %%%%%% ??
    smile_list(SList,SListNew,Arg),
    !.

% Operator w/ 2 arguments
% (ose,ulose,... )
smile_list(SList,SListNew,Expr) :-
    Expr =.. [Functor|[LocMult,Arg2]],
    smile_list(SList,SList1,Arg2),
    cons_check(LocMult,Functor),
    smile_parent(Functor,LocMult,SList1,SListNew),
    !.

% Operator w/ 3 arguments
% (here:compd)
smile_list(SList,SListNew,Expr) :-
    Expr =.. [compd|[Arg1,pref(Prefs),_Suffs]], %  Suffs ohne _ !!
    smile_list(SList,SList1,Arg1),
    smile_defaults(Arg1,SList1,SList2),
    smile_prefixes(Prefs,SList2,SListNew),
    !.



% Operator w/ 3 arguments
% (here:anose) -- rings
smile_list(SList,SListNew,Expr) :-
    Expr =.. [anose|[LocMult,Size,Arg3]],
    smile_list(SList,SList1,Arg3),
    cons_check(LocMult,anose),
    ring_construct(LocMult,Size,SList1,SListNew),
    !.

% Otherwise:
% NO SMILES available
smile_list(_SListOld,['NO␣SMILES'],_Expr).
```

```
% depending on functor (non-ring/ring:anose)
% call: cons_check(Mult*[Locs],Functor)

% nothing specified, any functor (ok)
cons_check('??'*['??'],_F) :-
    !.

% LocList is '??', any functor (underspecified, ok)
cons_check(_Mult*['??'],_F) :-
    !.

% Mult is '??', functor anose:
% LocList length has to be divisible by 2
cons_check('??'*LocList,anose) :-
    !,
    length(LocList,L),
    L mod 2 =:= 0.

% Mult is '??', any functor (ok)
cons_check('??'*_LocList,_F) :-
    !.

% fully specified, functor anose:
% Mult has to be LocList_Length/2
% the other cases are covered above
% (i.e. no LocList/nor multiplier specified)
cons_check(Mult*LocList,anose) :-
    !,
    length(LocList,L),
    AL is L/2,
    AL =:= Mult.

% Mult has to be equal LocList Length
cons_check(Mult*LocList,_F) :-
    length(LocList,L),
    L =:= Mult,
    !.

% alternatives for divisibility by 2:


% Generates smile_list skeleton structure
% (list of C-Atoms of specified chain length for sugars, e.g.)

% adds empty uspecs([])-term as first chain element
gen_skeleton(Mult,El,[uspecs([])|SList]) :-
    !,
    gen_skeleton(Mult,Mult,El,SList).

%% gen_skeleton/4:

% done at 0
gen_skeleton(0,_Length,_El,[]) :-
    !.

% generate chain element structure
% chain_el(Element,Locant,Branches,FeatureList)
gen_skeleton(Mult,Length,El,
             [chain_el(El,Loc,[],[])|DList]) :-
    NewMult is Mult-1,
    Loc is Length-NewMult,
    gen_skeleton(NewMult,Length,El,DList).



%% smile_parent/4:
% modify SMILES list according to parent-operators

% aldose (there is no mono-aldose)
smile_parent(ose,'??'*['??'],SmileList,SmileListNew) :-
    !,
    add2C(1,['=','O'],SmileList,SmileListNew).

% dialdose
smile_parent(ose,2*_LocList,SmileList,SmileListNew) :-
    !,
    add2C(1,['=','O'],SmileList,SmileList1),
    add2C(last,['=','O'],SmileList1,SmileListNew).

% ketose (no more locants: done)
smile_parent(ulose,_Mult*[],SmileListNew,SmileListNew) :-
    !.

% ketose (process locant by locant)
smile_parent(ulose,_Mult*[HeadLoc|LocList],SmileList,SmileListNew) :-
    !,
    add2C(HeadLoc,['=','O'],SmileList,SmileList1),
    smile_parent(ulose,_Mult*LocList,SmileList1,SmileListNew).

% yl (unsaturated)
smile_parent(yl,_Mult*[],SmileListNew,SmileListNew) :-
    !.
% yl (unsaturated)
smile_parent(yl,'??'*['??'],SmileList,SmileListNew) :-
    !,
    add2flist([1],unsat,SmileList,SmileListNew).
% yl (unsaturated)
smile_parent(yl,_Mult*Locs,SmileList,SmileListNew) :-
    sort(Locs,LocsSorted),
    !,
    add2flist(LocsSorted,unsat,SmileList,SmileListNew).

% if all (functors) fail (hopefully there's nothing to do)
% e.g. ane
smile_parent(_Functor,_Mult*_Locs,SmileList,SmileList).


% traverses SmileList only once (-Locs must be in sorted order-)

% done
add2flist([],_Feature,SmileList,SmileList) :-
    !.
% locant found
add2flist([HeadLoc|Locs],Feature,
          [chain_el(El,HeadLoc,Branches,FList)|SList],
          [chain_el(El,HeadLoc,Branches,[Feature|FList])|SListNew]) :-
    !,
    add2flist(Locs,Feature,SList,SListNew).
% else (not appropriate locant)
add2flist(Locs,Feature,[Ch_El|SList],[Ch_El|SListNew]) :-
    add2flist(Locs,Feature,SList,SListNew).


%% add2C/4:
% adds Branch at locant Loc
% (as first of Branches of C in chain)
% call: add2C(Loc,Branch,InList,OutList)

% add El here
add2C(Loc,Branch,
      [chain_el(El,Loc,Branches,FList)|R],
      [chain_el(El,Loc,[Branch|Branches],FList)|R]) :-
    !.

% add Branch at last locant
% (i.e. only 1 chain element left in list)
add2C(last,Branch,[Ch_El],[Ch_ElNew]) :-
    !,
    add2C(_Loc,Branch,[Ch_El],[Ch_ElNew]).

% traverse list until specified locant reached
add2C(Loc,Branch,[Ch_El|R],[Ch_El|RNew]) :-
    add2C(Loc,Branch,R,RNew).


%% ring_construct/4:
% creates a ring connection
% call: ring_construct(Mult*Locs,RingSize,SmileListIn,SmileListOut)
% Mult or Locs or both may be underspecified ('??', ['??'] resp.)
% index for each ring connection will be the connections' lowest locant

% no mult, no locants specified: default 1 ring
ring_construct('??'*['??'],Size,SmileList,SmileListNew) :-
    !,
    ring_construct(1*['??'],Size,SmileList,SmileListNew).

% locants not specified, construct from ring size
ring_construct(Mult*['??'],Size,SmileList,SmileListNew) :-
    !,
    ring_construct_fromsize(Mult*['??'],Size,SmileList,SmileListNew).

% construct from specified locant list
% (multiplier is specified or not - consistency is already checked)
ring_construct(_Mult*Locs,Size,SmileList,SmileListNew) :-
    ring_construct_fromlocs(Locs,Size,SmileList,SmileListNew).


%% ring_construct_fromlocs/4:
% constructs ring from specified locants
% input:  2 locants and SmileList
% output: modified SmileList
% call: ring_construct_fromlocs(LocList,RingSize,SmileIn,SmileOut)
% modifies FeatureList of chain_el/4 (adds 'carbgrp' if appropriate,
% for remembering where the carbonyl group was - for config;
% adds 'ring_el...(...)' if a ring element is to be built)
% (consistency is already checked: cons_check())
% ringsize-locants-match is checked here
% locant order is checked and, if necessary, reversed

% done (empty locs-list)
ring_construct_fromlocs([],_Size,SmileList,SmileList) :- !.

% list of locants
% construct ring from first pair (then from the rest of locs)
ring_construct_fromlocs([Loc1,Loc2|LocList],Size,
                        SmileList,SmileListNew) :-
    Loc1 < Loc2,
    !,
    Size - 1 \== Loc2 - Loc1 + 1,
    ring_construct_loc1(Loc1,Loc2,SmileList,SmileList1),
    ring_construct_fromlocs(LocList,Size,SmileList1,SmileListNew).

% list of locants
% construct ring from first pair (then from the rest of locs)
ring_construct_fromlocs([Loc1,Loc2|LocList],Size,
                        SmileList,SmileListNew) :-
    Loc2 < Loc1,
    !,
    Size - 1 \== Loc1 - Loc2 + 1,
    ring_construct_loc1(Loc2,Loc1,SmileList,SmileList1),
    ring_construct_fromlocs(LocList,Size,SmileList1,SmileListNew).


%% ring_construct_loc1/4:
% looks for correct SmileList at locant1, and in that case
% modifies SmileList and looks for locant2
% ring_construct_loc1(Loc1,Loc2,InList,OutList)

% carbonyl group is at locant 1:
% change SmileList and FeatureList appropriately
ring_construct_loc1(Loc1,Loc2,
                    [chain_el(El,Loc1,[['=','O']],FList)|SmileList],
                    [chain_el(El,Loc1,[],[carbgrp,ring_el1(Loc1)|FList])|SmileListNew]) :-
    !,
    ring_construct_loc2(non-carb,Loc1,Loc2,SmileList,SmileListNew).

% change SmileList at locant 1;
% carbonyl group C(=O) must be at locant 2
ring_construct_loc1(Loc1,Loc2,
                    [chain_el(El,Loc1,[],FList)|SmileList],
                    [chain_el(El,Loc1,[],[ring_el1(Loc1)|FList])|SmileListNew]) :-
    !,
    ring_construct_loc2(carb,Loc1,Loc2,SmileList,SmileListNew).

% go on looking for first locant, where to change SmileList
ring_construct_loc1(Loc1,Loc2,
                    [Ch_El|SmileList],
                    [Ch_El|SmileListNew]) :-
    ring_construct_loc1(Loc1,Loc2,SmileList,SmileListNew).
```

```
%% ring_construct_loc2/5:
% looks for correct structure at locant 2 (i.e. carbonyl group required or not ?)
% and changes SmileList accordingly to build a ring (uses Loc1 as index),
% changes FeatureList also

% carbonyl group required, which exists at locant 2
ring_construct_loc2(carb,Loc1,Loc2,
                    [chain_el(El,Loc2,[['=','O']],FList)|SmileList],
                    [chain_el(El,Loc2,[[ring-'O']],[carbgrp,ring_el2(Loc1)|FList])|SmileList]) :-
    !.

% non-carbonyl-group required and there is no such at locant 2
ring_construct_loc2(non-carb,Loc1,Loc2,
                    [chain_el(El,Loc2,[],FList)|SmileList],
                    [chain_el(El,Loc2,[[ring-'O']],[ring_el2(Loc1)|FList])|SmileList]) :-
    !.

% traverse list unless we are at locant 2
ring_construct_loc2(CarbTag,Loc1,Loc2,
                    [Ch_El|SmileList],
                    [Ch_El|SmileListNew]) :-
    ring_construct_loc2(CarbTag,Loc1,Loc2,SmileList,SmileListNew).


%% ring_construct_fromsize/4:
% construct ring only from ringsize by looking for a carbonyl group
% and trying to connect at first with higher locant (forwards), otherwise
% -if impossible- try to connect with lower locant (backwards)
% modifies FeatureList of chain_el/4 (adds 'carb' if appropriate)

% done (Mult is 0)
ring_construct_fromsize(0*['??'],_Size,SmileList,SmileList) :- !.

% try connecting forwards
ring_construct_fromsize(Mult*['??'],Size,SmileList,SmileListNew) :-
    ring_construct_fromsize_fwd1(_Loc1,_Loc2,Size,SmileList,SmileList1),
    !,
    MultNew is Mult - 1,
    ring_construct_fromsize(MultNew*['??'],Size,SmileList1,SmileListNew).

% try connecting backwards (if forwards connecting failed)
ring_construct_fromsize(Mult*['??'],Size,SmileList,SmileListNew) :-
    ring_construct_fromsize_bkwd(_Loc1,Size,SmileList,SmileList1),
    !,
    MultNew is Mult - 1,
    ring_construct_fromsize(MultNew*['??'],Size,SmileList1,SmileListNew).


%% ring_construct_fromsize_fwd1/5:
% search fwd for carbonyl group '=O' and construct ring element 1,
% then go on and construct ring element 2;
% use ringsize for calculating index of ring element 2
% -- looks for first matching possibility, currently --
% first call w/ these arguments: (_,_,Ringsize,InList,OutList)

% found a carbonyl group
ring_construct_fromsize_fwd1(Loc1,Loc2,RSize,
                             [chain_el(El,Loc1,[['=','O']],FList)|SList],
                             [chain_el(El,Loc1,[],[carbgrp,ring_el1(Loc1)|FList])|SListNew]) :-
    Loc2 is Loc1 + RSize - 2, % (2 b/c 'O' counts as ring element)
    ring_construct_fromsize_fwd2(Loc1,Loc2,SList,SListNew),
    !.

% go on looking for a carbonyl group
ring_construct_fromsize_fwd1(Loc1,Loc2,RSize,
                             [Ch_El|SList],
                             [Ch_El|SListNew]) :-
    ring_construct_fromsize_fwd1(Loc1,Loc2,RSize,SList,SListNew).


%% ring_construct_fromsize_fwd2/4:
% search fwd for ring element 2 and construct it
% by adding the index number (from locant 1, where carbonyl group was)

% ring closure here (at calculated second locant)
```

```
ring_construct_fromsize_fwd2(Loc1,Loc2,
                             [chain_el(El,Loc2,[],FList)|SList],
                             [chain_el(El,Loc2,[[ring-'O']],[ring_el2(Loc1)|FList])|SList]) :-
    !.

% go on looking for element w/ index for ring element 2
ring_construct_fromsize_fwd2(Loc1,Loc2,
                             [Ch_El|SList],
                             [Ch_El|SListNew]) :-
    ring_construct_fromsize_fwd2(Loc1,Loc2,SList,SListNew).


%% ring_construct_fromsize_bkwd/4:
% constructs a ring by looking for a carbonyl group and
% then calculating (locants) downwards for building the ring connection
% -- looks for first matching possibility, currently --
% first call w/ arguments: (_,RingSize,InList,OutList)
% - recursive-trick... -

% no more possibilities
ring_construct_fromsize_bkwd(_Loc1,_RSize,[],[]) :- fail.

% carbonyl group found
% build ring connection and
% instantiate Loc1 with calculated locant number for ring element 1
ring_construct_fromsize_bkwd(Loc1,RSize,
                             [chain_el(El,Loc2,[['=','O']],FList)|SmileList],
                             [chain_el(El,Loc2,[[ring-'O']],[carbgrp,ring_el2(Loc1)|FList])|
                                   SmileList]) :-
    Loc1 is Loc2 - (RSize -2),
    Loc1 > 0.

% recursive call... until carbonyl group is found
% calculated Loc1 has to match with current locant number to
% construct ring element 1 here
ring_construct_fromsize_bkwd(Loc1,RSize,
                             [chain_el(El,Loc1,[],FList)|SmileList],
                             [chain_el(El,Loc1,[],[ring_el1(Loc1)|FList])|SmileListNew]) :-
    ring_construct_fromsize_bkwd(Loc1,RSize,SmileList,SmileListNew),
    !.

% recursive rule for elements that don't have to be changed
% (i.e. non-carbonyl group,
% not the calculated locant for ring element 1,
% or no possible connection there)
ring_construct_fromsize_bkwd(Loc1,RSize,
                             [Ch_El|SmileList],
                             [Ch_El|SmileListNew]) :-
    ring_construct_fromsize_bkwd(Loc1,RSize,SmileList,SmileListNew).



%% smile_defaults/3:
% adds default structural parts to SmileList
% (first decides, if sugar or nonsugar defaults to impose)
% usually called when processing compd(...), after smile_list-call

% if functor belongs to sugar_group
smile_defaults(Expr,SList,SListNew) :-
    Expr =.. [Functor|_Args],
    (
      Functor == anose
    ;
      Functor == ose
    ;
      Functor == ulose
    ),
    !,
    smile_defaults_s(SList,SListNew).

% if functor belongs to nonsugar_group
smile_defaults(Expr,SList,SListNew) :-
    Expr =.. [Functor|_Args],
    Functor == yl,
    smile_defaults_ns(SList,SListNew),
    !.
```

```
% no defaults for... triv_name
smile_defaults(Expr,SList,SList) :-
    Expr =.. [Functor|_Args],
    Functor == triv_name.


%% smile_defaults_ns/2:
% smile defaults for nonsugars

% done (traversed whole SmileList)
smile_defaults_ns([],[]) :- !.

% add unsat to FeatureList
smile_defaults_ns([chain_el(_El,1,[],FList)|SList],
                  [chain_el(_El,1,[['[H]'],['[H]'],['[H]']],FList)|SListNew]) :-
    member(unsat,FList),
    smile_defaults_ns(SList,SListNew).

% (last chain element)
smile_defaults_ns([chain_el(_El,_LastLoc,[],FList)],
                  [chain_el(_El,_LastLoc,[['[H]'],['[H]'],['[H]']],FList)]) :-
    !,
    member(unsat,FList).

% if nothing matched:
% (e.g. wrong locant)
% go to next chain element
smile_defaults_ns([Ch_El|SList],[Ch_El|SListNew]) :-
    smile_defaults_ns(SList,SListNew).


%% smile_defaults_s/2:
% defaults for sugar group
% adds ([H])(O) to C in chain, where no other element or index
% ([H]) is for configuration and substitution purposes

% done (whole list traversed)
smile_defaults_s([],[]) :- !.

% ring connection (ring element 1 or 2)  or
% some element already there ('(=O)'), or ring element
% go on w/o adding (O)
smile_defaults_s([chain_el(El,Loc,Branches,FList)|SmileList],
               [chain_el(El,Loc,Branches,FList)|SmileListNew]) :-
    (
      length(Branches,L), L > 0
    ;
      member(ring_el1(_L),FList)
    ;
      member(ring_el2(_L),FList)
    ),
    !,
    smile_defaults_s(SmileList,SmileListNew).

% uspecs are at the beginning of C-chain in pre-representation
smile_defaults_s([uspecs(USpecs)|SmileList],
               [uspecs(USpecs)|SmileListNew]) :-
    !,
    smile_defaults_s(SmileList,SmileListNew).

% add '[H]','O' to (empty) list of branches
smile_defaults_s([chain_el(El,Loc,[],FList)|SmileList],
               [chain_el(El,Loc,[['[H]'],['O']],FList)|SmileListNew]) :-
    !,     % not necessary since not the last rule...
    smile_defaults_s(SmileList,SmileListNew).


%% smile_prefixes/3:
% process the various types of operators
% appearing in the prefixes-list (2nd arg of compd)

% nothing to do, or done
smile_prefixes([],SList,SList).

% first: select configurations
```

```
smile_prefixes(PrefList,SList,SListNew) :-
    select(cfg(CfgList),PrefList,PrefListNew),
    !,
    smile_prefix(cfg(CfgList),SList,SList1),
    smile_prefixes(PrefListNew,SList1,SListNew).

% select deoxy-prefix (before processing other prefixes)
smile_prefixes(PrefList,SList,SListNew) :-
    select(Mult*Locs-deoxy,PrefList,PrefListNew),
    !,
    smile_prefix(Mult*Locs-deoxy,SList,SList1),
    smile_prefixes(PrefListNew,SList1,SListNew).

% process prefix by prefix
smile_prefixes([Pref1|PrefList],SList,SListNew) :-
    smile_prefix(Pref1,SList,SList1),
    smile_prefixes(PrefList,SList1,SListNew).


%% smile_prefix/3:
% process the different types of prefixes

% double prefix, (C-substituted monosaccharide)
% e.g. 'C'-methyl, 'C'-Acetamido,...
smile_prefix(Mult*Locs-'C'-Prefix,SList,SListNew) :-
    cons_check(Mult*Locs,Prefix),
    sort(Locs,SortedLocs),   % noetig?
    smile_list(_L,SListAdd,Prefix),
    substitute(SortedLocs,SList,SListNew,'[H]',SListAdd),
    !.

%% single prefix
% at least one Locant must have been found
smile_prefix(Mult*['??']-Prefix,
             [uspecs(USpecs)|SList],
             [uspecs([Mult*AllLocs-Prefix|USpecs])|SList]) :-
    lex_smile(Prefix,Action,ArgList),
    CallAction =.. [Action,[Loc],SList,_SListNew|ArgList],
    CallFindall =.. [findall,Loc,CallAction,AllLocs],
    call(CallFindall),
    AllLocs \== [],
    !.

%% alternative clause for underspecified prefixes
% (but no substitution was possible)
% last rule for this type
smile_prefix(Mult*['??']-Prefix,
             [uspecs(USpecs)|SList],
             [uspecs([Mult*['??']-Prefix|USpecs])|SList]) :-
    !.


%% single prefix,
% e.g. deoxy, amino, thio, seleno, telluro
smile_prefix(Mult*Locs-Prefix,SList,SListNew) :-
    cons_check(Mult*Locs,Prefix),
    sort(Locs,SortedLocs), % deletes doubles
    lex_smile(Prefix,Action,ArgList),
    CallAction =.. [Action,SortedLocs,SList,SListNew|ArgList],
    call(CallAction),
    !.

% prefix which is a compound (not listed in lex_smile)
% i.e. a substituted derivative of a deoxy sugar
smile_prefix(Mult*Locs-Prefix,SList,SListNew) :-
    cons_check(Mult*Locs,Prefix),
    sort(Locs,SortedLocs),
    smile_list(_L,SListAdd,Prefix),
    substitute(SortedLocs,SList,SListNew,deoxy-'[H]',SListAdd),
    !.



% configurational prefix list,
% e.g. cfg([alpha-'D'-gluco,'D'-xylo])
% reverse order because group nearest C1 is cited last
% in case first C-Atom is involved in ring connection:
smile_prefix(cfg(CfgList),[chain_el(El,Loc,Branches,FList)|SList],SListNew) :-
    member(ring_el1(_L),FList),
    !,
    reverse(CfgList,CfgListRev),
    smile_config(CfgListRev,[chain_el(El,Loc,Branches,FList)|SList],SListNew).

% ... otherwise keep from adding chirality to first C-Atom:
smile_prefix(cfg(CfgList),[Ch_El|SList],[Ch_El|SListNew]) :-
    reverse(CfgList,CfgListRev),
    smile_config(CfgListRev,SList,SListNew).


%% replace_el/4:
% replaces first occurence in List1 of A by B, Result in List2
% replace_el(List1,A,B,List2)
% Element has to be in a separate list: [...,[A],...],[...],...

replace_el([],_A,_B,[]) :- !.

replace_el([[H]|T],A,B,[[B]|Result]) :-
    H=A,
    T=Result,
    !.

replace_el([H|T],A,B,[H|Result]) :-
    replace_el(T,A,B,Result).


%% substitute/5:
% substitute(Locants,SMILEList,SMILEListNew,DeletionElement,AdditionElement)
% traverses SMILEList and substitutes at Locants (given in low-high-order!)
% AdditionElement for DeletionElement

% all locants have been processed
substitute([],SList,SList,_DelEl,_AddEL) :- !.

% element with appropriate locant found
% and branches-list is not empty
% replaced means that BranchesList should have changed
substitute([Loc|LocsRestList],
        [chain_el(El,Loc,Branches,FList)|SListRest],
        [chain_el(El,Loc,BranchesNew,FList)|SListRestNew],
        DelEl,AddEl) :-
    length(Branches,L), L > 0,
    replace_el(Branches,DelEl,AddEl,BranchesNew),
    Branches \== BranchesNew,
    substitute(LocsRestList,SListRest,SListRestNew,DelEl,AddEl).

% go on looking for appropriate locant
substitute(LocsList,
        [Ch_El|SListRest],
        [Ch_El|SListRestNew],
        DelEl,AddEl) :-
    substitute(LocsList,SListRest,SListRestNew,DelEl,AddEl).


%%% smile_config/3:
% processes configs (given as list-prefixes)
% and generates chirality information in FeatureList

% no more configs: done
smile_config([],SList,SList) :- !.

% process 1st config.,
% then restlist of config list
smile_config([Cfg|CfgRestList],SList,SListNew) :-
    !,
    smile_config(Cfg,SList,SList1),
    smile_config(CfgRestList,SList1,SListNew).

% config. consists of anom.config.symb.-config.symb.-config.prefix,
% e.g. alpha-'D'-xylo
smile_config(CfgSymbAnom-CfgSymb-CfgPref,SList,SListNew) :-
    !,
    lex_smile(CfgPref,ChiralityCombis),
    smile_config_symb(CfgSymb,ChiralityCombis,_RingIndex,SList,SList1),
    lex_smile(CfgSymbAnom-CfgSymb,Chir),
    smile_config_anom(Chir,_RingIndex,SList1,SListNew).

% config. consists of config.symb.-config.prefix,
% e.g. 'D'-gulo
smile_config(CfgSymb-CfgPref,SList,SListNew) :-
    !,
    lex_smile(CfgPref,ChiralityCombis),
    smile_config_symb(CfgSymb,ChiralityCombis,_RingIndex,SList,SListNew).

% config. consists of anom.config.symb.-config.symb.
% happens to appear in front of trivial name
% similar to rule above (which fails -- b/c of lex_smile)
smile_config(CfgSymbAnom-CfgSymb,SList,SListNew) :-
    !,
    smile_config_symb(CfgSymb,SList,SList1),
    lex_smile(CfgSymbAnom-CfgSymb,Chir),
    smile_config_anom(Chir,_RingIndex,SList1,SListNew).

% only config.prefix
%happens to appear after having converted trivial name
smile_config(CfgPref,SList,SListNew) :-
    lex_smile(CfgPref,ChiralityCombis),
    !,
    smile_config_symb('D',ChiralityCombis,_RingIndex,SList,SListNew).

% only config. symbol;
% happens to appear in front of trivial names
smile_config(CfgSymb,SList,SListNew) :-
%    !, % not necessary as long as it's the last rule
    smile_config_symb(CfgSymb,SList,SListNew).



%%% smile_config_symb/5:
% assigns chirality symbols from chirality list,
% considers config. symbol, hereby
% ring connection atoms aren't yet assigned their chirality (see ..._anom)

% ChiralityCombi-List empty, done
smile_config_symb(_CfgSymb,[],_RingIndex,SList,SList) :- !.

% config.symbol L:
% turns @ into @@, and @@ into @
% (no chirality for first C-atom or ring elements)
smile_config_symb('L',[Chir|ChiralityCombisRest],
                  RingIndex,
                  [chain_el(El,Loc,[['[H]'],['O']],FList)|SList],
                  [chain_el(El,Loc,[['[H]'],['O']],[chir(OtherChir)|FList])|SListNew]) :-
    Loc > 1,
    \+ member(ring-el1(_RLoc),FList),
    \+ member(ring-el2(_RLoc),FList),
    !,
    lex_smile(Chir,OtherChir),
    smile_config_symb('L',ChiralityCombisRest,RingIndex,SList,SListNew).

% config.symbol D:
% chiralities need no converting
% (...and DL, and meso- ,too? )
% (no chirality for first C-atom or ring elements)
smile_config_symb(CfgSymb,[Chir|ChiralityCombisRest],
                  RingIndex,
                  [chain_el(El,Loc,[['[H]'],['O']],FList)|SList],
                  [chain_el(El,Loc,[['[H]'],['O']],[chir(Chir)|FList])|SListNew]) :-
    Loc > 1,
    \+ member(ring-el1(_RLoc),FList),
    \+ member(ring-el2(_RLoc),FList),
    !,
    smile_config_symb(CfgSymb,ChiralityCombisRest,RingIndex,SList,SListNew).

% when encountering 1st ring-atom
% and it hadn't had a carbonyl group:
% remove one chirality symbol from list
% (it is to be given later),
% indicate that 2nd ring-Atom is to be found (instantiate RingIndex)
smile_config_symb(CfgSymb,[_Chir|ChiralityCombis],
                  RingIndex,
                  [chain_el(El,RingIndex,Branches,FList)|SList],
                  [chain_el(El,RingIndex,Branches,FList)|SListNew]) :-
    member(ring_el1(RingIndex),FList),
    \+ member(carbgrp,FList),
    !,
    smile_config_symb(CfgSymb,ChiralityCombis,RingIndex,SList,SListNew).

% when encountering 1st ring atom
% and it had a carbonyl group:
% don't change chirality symbols list;
% indicate that 2nd ring-Atom is to be found (instantiate RingIndex)
smile_config_symb(CfgSymb,ChiralityCombis,
                  RingIndex,
                  [chain_el(El,RingIndex,Branches,FList)|SList],
                  [chain_el(El,RingIndex,Branches,FList)|SListNew]) :-
    member(ring_el1(RingIndex),FList),
    member(carbgrp,FList),
    !,
    smile_config_symb(CfgSymb,ChiralityCombis,RingIndex,SList,SListNew).

% when encountering 2nd ring-atom:
% check if it has the RingIndex of the 1st found ring element 1,
% remove chirality symbol from list
% (it is to be given later),
% if it wasn't the carbonyl group
smile_config_symb(CfgSymb,[_Chir|ChiralityCombisRest],
                  RingIndex,
                  [chain_el(El,Loc2,Branches,FList)|SList],
                  [chain_el(El,Loc2,Branches,FList)|SListNew]) :-
    member(ring_el2(RingIndex),FList),
    \+ member(carbgrp,FList),
    !,
    smile_config_symb(CfgSymb,ChiralityCombisRest,RingIndex,SList,SListNew).

% any other config.symbol, (or case)
% w/o matching the above cases
% e.g. ['=','O'], or ring element (w/ former carbonyl group),...
% ignore from giving a chirality symbol
smile_config_symb(CfgSymb,ChiralityCombis,
                  RingIndex,
                  [Ch_El|SList],
                  [Ch_El|SListNew]) :-
    smile_config_symb(CfgSymb,ChiralityCombis,RingIndex,SList,SListNew).



%%% smile_config_symb/3:
% lonely config.symbol w/o config.prefix,
% happens to appear before trivial names,
% which means that a config.prefix has already been applied
% lonely should mean, that there is no ring connection in SMILES
% string

% no need to change config in SMILES string
smile_config_symb('D',SList,SList) :- !.

% L changes @ into @@ and @@ into @
smile_config_symb('L',
                  [chain_el(El,Loc,Branches,FList)|SList],
                  [chain_el(El,Loc,Branches,[chir(OtherChir)|FListNew])|SListNew]) :-
%    Loc > 1,  %% has to be some but the 1st element? (treated below)
    \+ member(ring_el1(_L),FList),
    \+ member(ring_el2(_L),FList),
    select(chir(Chir),FList,FListNew), %% 1st has no chir-Feature
    !,
    lex_smile(Chir,OtherChir),
    smile_config_symb('L',SList,SListNew).

% done (traversed whole SMILES list)
smile_config_symb('L',[],[]) :- !.

% otherwise (there is no chirality here;
% it's the 1st chain-atom;...)
```

```
smile_config_symb('L',[Ch_El|SList],[Ch_El|SListNew]) :-
    smile_config_symb('L',SList,SListNew).



%%% smile_config_anom/4:
% chirality sign and appropriate notation at anomeric centre
% call w/ arguments: (ChirSymb,_RingIndex,SmileList,SmileListNew)

smile_config_anom(_Chir,_RingIndex,[],[]) :- !.

% done
% (no further config.prefix processed,
% and its's not a ring element)
smile_config_anom(_Chir,_RingIndex,
                  [chain_el(El,Loc,Branches,FList)|SListRest],
                  [chain_el(El,Loc,Branches,FList)|SListRest]) :-
    Loc > 1,
    \+ member(chir(_C),FList),
    \+ member(ring_el1(_RLoc),FList),
    \+ member(ring_el2(_RLoc),FList),
    !.

% location of 1st RingElement at locant 1 (Pos.1 in C-chain)
smile_config_anom(Chir,1,
                  [chain_el(El,1,[],FList)|SList],
                  [chain_el(El,1,[['[H]'],['O']],[chir(OtherChir)|FList])|SListNew]) :-
    member(ring_el1(1),FList),
    !,
    lex_smile(Chir,OtherChir),
    smile_config_anom(Chir,1,SList,SListNew).

% location of 1st RingElement (not at 1st locant)
smile_config_anom(Chir,RingIndex,
                  [chain_el(El,RingIndex,[],FList)|SList],
                  [chain_el(El,RingIndex,[['O']],[chir(OtherChir)|FList])|SListNew]) :-
    member(ring_el1(RingIndex),FList),
    RingIndex =\= 1,
    !,
    lex_smile(Chir,OtherChir),
    smile_config_anom(Chir,RingIndex,SList,SListNew).

% location of 2nd RingElement
smile_config_anom(Chir,RingIndex,
                  [chain_el(El,Loc2,[[ring-'O']],FList)|SList],
                  [chain_el(El,Loc2,[['[H]'],[ring-'O']],[chir(Chir)|FList])|SListNew]) :-
%    member(ring_el2(RingIndex),FList), % not necessary, theres a ring-'O'
    !,
    smile_config_anom(Chir,RingIndex,SList,SListNew).

% any other... : go on
smile_config_anom(Chir,RingIndex,
                  [Ch_El|SList],
                  [Ch_El|SListNew]) :-
    smile_config_anom(Chir,RingIndex,SList,SListNew).



%% lex_smile/2:
% lexicon for
% prefix-chirality-combinations,
% chirality-sign at anomeric centre

% Chirality combinations list,
% given in D-configuration, starting from C1
lex_smile(glycero,['@']).
lex_smile(erythro,['@','@']).
lex_smile(threo,['@@','@']).
lex_smile(ribo,['@','@','@']).
lex_smile(arabino,['@@','@','@']).
lex_smile(xylo,['@','@@','@']).
lex_smile(lyxo,['@@','@@','@']).
lex_smile(allo,['@','@','@','@']).
lex_smile(altro,['@@','@','@','@']).
lex_smile(gluco,['@','@@','@','@']).
lex_smile(manno,['@@','@@','@','@']).
lex_smile(gulo,['@','@','@@','@']).
lex_smile(ido,['@@','@','@@','@']).
lex_smile(galacto,['@','@@','@@','@']).
lex_smile(talo,['@@','@@','@@','@']).



% chirality at anomeric centre
lex_smile(alpha-'D','@'). % formally cis to last C (D is in smiles.pl @)
lex_smile(alpha-'L','@@').% L is here @@
lex_smile(beta-'D','@@'). % formally trans to last C (D is in smiles.pl @)
lex_smile(beta-'L','@').  % L is here @@

% other chirality
lex_smile('@','@@').
lex_smile('@@','@').



% Actions for specified prefixes
lex_smile(deoxy,substitute,['O',deoxy-'[H]']).
lex_smile(amino,substitute,[deoxy-'[H]','N']).
lex_smile(thio,substitute,['O','S']).
lex_smile(thio,substitute,[ring-'O',ring-'S']). % may be ring connection
lex_smile(seleno,substitute,['O','Se']).


%% lex_triv/3:
% lexicon for trivial names associating
% their correpsonding systematic name and
% their corresponding semantic representation

% D-Ribose
lex_triv(ribose,'D-ribo-pentose',compd(ose(?? *[??],5*'C'),pref([cfg(['D'-ribo])]),suff([]))).
lex_triv(fructose,'D-arabino-Hex-2-ulose',compd(ulose(?? *[2],6*'C'),pref([cfg(['D'-arabino])]),
     suff([]))).
lex_triv(glucose,'D-gluco-Hexose',compd(ose(?? *[??],6*'C'),pref([cfg(['D'-gluco])]),suff([]))).
```

A.4 Classes Module

A.4.1 File `classes.pl`

```
% ------------------------------------------ %
% File:     classes.pl                        %
% Author:   Stefanie Anstein                  %
% Purpose:  classifier                        %
% ------------------------------------------ %


%%% determine classes from semantic representation term

%% Prolog call: classes(Sem_Repr_Term,ClassList).


%%%% fully specified names

%%% main rule for class calculus
classes(compd(Par,pref(PrefPredList),suff(SuffPredList)),ClassList) :-
    parpredlist(Par,ParPredList),
    predlistscomb2classes(ParPredList,PrefPredList,SuffPredList,ClassList1),
    addsuperclasses(ClassList1,ClassList),
    !.

%%%% class/underspecified names

%% class names w/o affixes
% e.g. alkene: compd(class_name(alkene),pref([]),suff([]))
classes(compd(class_name(ClassNameforList),pref([]),suff([])),[ClassNameforList]) :- !.

%% class names w/ affixes
% e.g. 2-alkene: compd((?? *[2],class_name(alkene)),pref([]),suff([]))
classes(compd((_Mult*_Locs,class_name(ClassNameforList)),pref([]),suff([])),[ClassNameforList])
     :- !.

%% class names 'within' functional group suffix
% e.g. pentanoic anhydride:
% compd(ane(5*'C'),pref([]),suff([??? *[??]-(adj_suff(oic)+class_name(anhydride))]))
classes(compd(Par,
              pref([]),
              suff([_Mult1*_Locs-(adj_suff(_AdjSuff)+class_name(ClassName))])),
        [ClassName,SecClassName]) :-
    parpredlist(Par,ParPredList),
    member(_Mult2*_Elem-ParPred,ParPredList),
    prefchangeclass(ParPred,none,none,SecClassName),
    !.

%% class names in 'radiofunctional nomenclature'
% e.g. methyl alcohol:
% compd(compd(yl(?? *[??],ane(1*'C')),pref([]),suff([]))+class_name(alcohol),pref([]),suff([]))
classes(compd(compd(_PAR,_PREF,_SUFF)+class_name(ClassName),pref([]),suff([])),[ClassName]) :- !.

% more complex
classes(compd(compd(_PAR1,_PREF1,_SUFF1)+compd(_PAR2,_PREF2,_SUFF2)+class_name(ClassName),pref
     ([]),suff([])),[ClassName]) :- !.


%%%% trivial names

%% trivial names: lookup
% e.g. benzene: compd(triv_name(benzene),pref([]),suff([]))
classes(compd(triv_name(benzene),pref([]),suff([])),['AROMATIC']) :- !.


%%% cases not covered
classes(_Compd_Expression,['NO␣CLASSES']).


%% collect all predicates from parent argument in a list

% stop at deepest embedding
parpredlist(_Mult*_Locs,[]) :-
      !.
parpredlist(ClassName,[]) :-
```

```
      atom(ClassName), !.

% add functors; proceed with second ...
parpredlist(Term,[Arg1-Functor|List]) :-
    Term =.. [Functor|[Arg1,Arg2]],
    parpredlist(Arg2,List).

% ... or with first (only) argument
parpredlist(Term,[Arg1-Functor|List]) :-
    Term =.. [Functor|[Arg1]],
    parpredlist(Arg1,List).


%%% rules for 'predicate interaction'  (order important!)

%% (a) specific combination of par, pref and suff
% e.g. 5-deoxypentan-5-ol:
% compd(ane(5*'C'),pref([?? *[5]-deoxy]),suff([?? *[5]-ol]))
predlistscomb2classes(ParPredList,PrefPredList,SuffPredList,Class) :-
    member(_Mult*_Elem-ParPred,ParPredList),
    member(_Mult*[Loc|_Rest]-PrefPred,PrefPredList),
    member(_Mult*[Loc|_Rest]-SuffPred,SuffPredList),
    prefchangeclass(ParPred,PrefPred,SuffPred,Class),
    !.

%% (b) only combination of par and pref, no suff
% e.g. 2-oxahexane:
% compd(ane(6*'C'),pref([?? *[2]-oxa]),suff([]))
predlistscomb2classes(ParPredList,PrefPredList,[],Class) :-
    member(_Loc*_Elem-ParPred,ParPredList),
    member(_Mult*_Locs-PrefPred,PrefPredList),
    prefchangeclass(ParPred,PrefPred,none,Class),
    !.

% e.g. 1-methyl-hexane:
% compd(ane(6*'C'),pref([?? *[1]-compd(yl(?? *[?],ane(1*'C')),pref([]),suff([]))]),suff([]))
predlistscomb2classes(ParPredList,PrefPredList,[],Class) :-
    member(_Loc*_Elem-ParPred,ParPredList),
    member(_Mult*_Locs-_compd_Any,PrefPredList),
    prefchangeclass(ParPred,compd(_Any),none,Class),
    !.

% e.g. 5-hydroxypentane:
% compd(ane(5*'C'),pref([?? *[5]-hydroxy]),suff([]))
predlistscomb2classes(ParPredList,PrefPredList,[],Class) :-
    member(Loc*_Elem-ParPred,ParPredList),
    member(_Mult*[Loc|_Rest]-PrefPred,PrefPredList),
    prefchangeclass(ParPred,PrefPred,none,Class),
    !.

% e.g. 2,3-dihydropent-2-ene:
% compd(ene(?? *[2],ane(5*'C')),pref([2*[2,3]-hydro]),suff([]))
% e.g. 2,3-dihydropent-2-yne:
% compd(yne(?? *[2],ane(5*'C')),pref([2*[2,3]-hydro]),suff([]))
% e.g. 1,2-didehydropent-1-ene:
% compd(ene(?? *[1],ane(5*'C')),pref([2*[1,2]-dehydro]),suff([]))
predlistscomb2classes(ParPredList,PrefPredList,[],Class) :-
    member(_Mult*[CommonLoc|_Rest]-ParPred,ParPredList),
    member(2*[CommonLoc,FollowingLoc|_Rest]-PrefPred,PrefPredList),
    prefchangeclass(ParPred,PrefPred,none,Class),
    FollowingLoc is CommonLoc+1.

% e.g. 1,2-didehydropentane:
% compd(ane(5*'C'),pref([2*[1,2]-dehydro]),suff([]))
predlistscomb2classes(ParPredList,PrefPredList,[],Class) :-
    member(_Mult*'C'-ParPred,ParPredList),
    member(2*[Loc,FollowingLoc|_Rest]-PrefPred,PrefPredList),
    prefchangeclass(ParPred,PrefPred,none,Class),
    FollowingLoc is Loc+1.


%% (c) only combination of par and suff, no pref

% e.g. methanolate:
% compd(ane(1*'C'),pref([]),suff([?? *[?]-ol,?? *[?]-ate]))
predlistscomb2classes(ParPredList,[],SuffPredList,Class) :-
```

```
    member(_Mult*_Elem-ParPred,ParPredList),
    member(_Mult1*_Locs-SuffPred1,SuffPredList),
    member(_Mult1*_Locs-SuffPred2,SuffPredList),
    prefchangeclass(ParPred,none,[SuffPred1,SuffPred2],Class),
    !.

% e.g. pentan-5-ol: compd(ane(5*'C'),pref([]),suff([?? *[5]-ol]))
predlistscomb2classes(ParPredList,[],SuffPredList,Class) :-
    member(Loc*_Elem-ParPred,ParPredList),
    member(_Mult*[Loc|_Rest]-SuffPred,SuffPredList),
    prefchangeclass(ParPred,none,SuffPred,Class),
    !.

% e.g. propanal: compd(ane(3*'C'),pref([]),suff([?? *[?]-al]))
predlistscomb2classes(_ParPredList,[],SuffPredList,Class) :-
    member(_Mult*[_Loc|_Rest]-SuffPred,SuffPredList),
    prefchangeclass(_ParPred,none,SuffPred,Class),
    !.


%% (d) neither pref nor suff

% e.g. hexane: compd(ane(6*'C'),pref([]),suff([]))
predlistscomb2classes([_Mult*_Elem-ane],[],[],['ALKANE']).

% e.g. methyl pentane:
% compd(add(yl(?? *[?],ane(1*C)),ane(5*C)),pref([]),suff([]))
predlistscomb2classes(ParPredList,[],[],Class) :-
    member(_AddPart-ParPred1,ParPredList),
    member(_Mult*_Elem-ParPred2,ParPredList),
    prefchangeclass([ParPred1,ParPred2],none,none,Class),
    !.

% e.g. 3-penten-1-yne:
% compd(yne(?? *[1],ene(?? *[3],ane(5*'C'))),pref([]),suff([]))
predlistscomb2classes(ParPredList,[],[],Class) :-
    member(_Mult1*_Loc1-ParPred1,ParPredList),
    member(_Mult2*_Loc2-ParPred2,ParPredList),
    prefchangeclass([ParPred1,ParPred2],none,none,Class),
    !.

% e.g. hex-2-ene: compd(ene(?? * [2],ane(6*'C')),pref([]),suff([]))
predlistscomb2classes(ParPredList,[],[],Class) :-
    member(_Mult*_Loc-ParPred,ParPredList),
    prefchangeclass(ParPred,none,none,Class),
    !.

% e.g. cyclopentane: compd(cyclo(?? *[1,5],ane(5*'C')),pref([]),suff([]))
predlistscomb2classes(ParPredList,[],[],Class) :-
    member(_Mult1*_Loc-ParPred1,ParPredList),
    member(_Mult2*_Elem-ParPred2,ParPredList),
    prefchangeclass([ParPred1,ParPred2],none,none,Class),
    !.


%% (e) only combination of par and suff, pref = any w/ exceptions

% e.g. 2-phosphapentan-5-ol:
% compd(ane(5*'C'),pref([?? *[2]-phospha]),suff([?? *[5]-ol]))

predlistscomb2classes(ParPredList,PrefPredList,SuffPredList,Class) :-
    member(Loc*_Elem-ParPred,ParPredList),
    member(_Mult*_LocList-PrefPred,PrefPredList),
    member(_Mult*[Loc|_Rest]-SuffPred,SuffPredList),
    prefchangeclass(ParPred,PrefPred,SuffPred,Class),
    !.


%%% 'lexicon entries' for predicate interaction  (order important !!)
%%% listed for finite number of predicates and combinations

%% 'specials' (for class names 'within' functional group suffix)
prefchangeclass(ane,none,none,'ALKANE').

%% (a) specific combination of par, pref and suff
prefchangeclass(ane,deoxy,ol,['ALKANE']).
prefchangeclass(ene,deoxy,ol,['ALKENE']).

%% (b) only combination of par and pref, no suff
prefchangeclass(ane,oxa,none,['ALKANE']).
prefchangeclass(ane,compd(_Any),none,['ALKANE']).
prefchangeclass(ane,hydroxy,none,['ALKANE','PRIMARY␣ALCOHOL␣(-OH)']).
prefchangeclass(ene,hydro,none,['ALKANE']).
prefchangeclass(yne,hydro,none,['ALKENE␣(double␣bond)']).
prefchangeclass(ene,dehydro,none,['ALKYNE␣(triple␣bond)']).
prefchangeclass(ane,dehydro,none,['ALKENE␣(double␣bond)']).
prefchangeclass(ane,amino,none,['ALKANE','AMINE␣(-NH2)']).

%% (c) only combination of par and suff, no pref
prefchangeclass(ane,none,[ol,ate],['ALKANE','ALCOHOLATE','PHENOLATE']).
prefchangeclass(ane,none,ol,['ALKANE','PRIMARY␣ALCOHOL␣(-OH)']).
prefchangeclass(ene,none,ol,['ALKENE','PRIMARY␣ALCOHOL␣(-OH)']).
prefchangeclass(ane,none,al,['ALKANE','ALDEHYDE␣(-CHO)']).
prefchangeclass(ene,none,al,['ALKENE','ALDEHYDE␣(-CHO)']).
prefchangeclass(ane,none,aldehyde,['ALKANE','ALDEHYDE␣(-CHO)']).
prefchangeclass(ane,none,carbaldehyde,['ALKANE','ALDEHYDE␣(-CHO)']).
prefchangeclass(ane,none,one,['ALKANE','KETONE␣(>(C)=O)']).
prefchangeclass(ane,none,amide,['ALKANE','AMIDE␣(-CO-NH2)']).
prefchangeclass(ane,none,carboxamide,['ALKANE','AMIDE␣(-CO-NH2)']).
prefchangeclass(ane,none,amine,['ALKANE','AMINE␣(-NH2)']).
prefchangeclass(ane,none,olate,['ALKANE','ALCOHOLATE','PHENOLATE']).

%% (d) neither pref nor suff
prefchangeclass([add,ane],none,none,['ALKANE']).
prefchangeclass([yne,ene],none,none,['ALKENE␣(double␣bond)','ALKYNE␣(triple␣bond)']).
prefchangeclass(ene,none,none,['ALKENE␣(double␣bond)']).
prefchangeclass(yne,none,none,['ALKYNE␣(triple␣bond)']).
prefchangeclass(yl,none,none,['ALKYL']).
prefchangeclass([cyclo,ene],none,none,['CYCLOALKENE']).
prefchangeclass([cyclo,ane],none,none,['CYCLOALKANE']).

%% (e) only combination of par and suff, pref = any w/ exceptions
prefchangeclass(ane,AnyPref,ol,['ALKANE','PRIMARY␣ALCOHOL␣(-OH)']) :-
    no_change(AnyPref).

prefchangeclass(ene,AnyPref,ol,['ALKENE','PRIMARY␣ALCOHOL␣(-OH)']) :-
    no_change(AnyPref).

prefchangeclass(ane,AnyPref,one,['ALKANE','KETONE␣(>(C)=O)']) :-
    no_change(AnyPref).

no_change(phospha).
no_change(oxa).
no_change(thia).
no_change(hydroxy). % TBC; define different categories of prefixes


%%% add superclasses for a class hierarchy

% e.g. PRIMARY ALCOHOLS are ALCOHOLS
addsuperclasses(ClassList,['ALCOHOL␣(-OH)'|ClassList]) :-
    member('PRIMARY␣ALCOHOL␣(-OH)',ClassList).

% none
addsuperclasses(ClassList,ClassList) :- !.
```

B EBNF and Testsuite

B.1 EBNF for the Semantic Representation

B.1.1 File `ebnf.semrep.txt`

```
% ------------------------------------------ %
% File:     ebnf.semrep.txt                   %
% Author:   Stefanie Anstein, Gerhard Kremer  %
% Purpose:  definition of sem. representation %
% ------------------------------------------ %


COMPD = "compd", "(", PAR_PHR,
        "pref", "(", P_LIST, ")",
        "suff", "(", S_LIST, ")", ")";

PAR_PHR = OP1_CONSTRUCT
        | OP2_CONSTRUCT
        | OP3_CONSTRUCT       % sugar-specific
        | OP4_CONSTRUCT
        | OP5_CONSTRUCT
        | CLOSED_CLASS;

P_LIST = "[", {PREF_CONSTRUCT}, "]"
       | "[", PREF_CONSTRUCT, {",", PREF_CONSTRUCT}, "]"
       | "[", "]";         % no prefix

S_LIST =  "[", {SUFF_CONSTRUCT}, "]";
       | "[", SUFF_CONSTRUCT, {",", SUFF_CONSTRUCT}, "]"
       | "[", "]";         % no suffix

OP1_CONSTRUCT = OP1, "(", STRUCT, ")";

OP2_CONSTRUCT = OP2, "(", LOCS_MULT, ",", STRUCT, ")";

OP3_CONSTRUCT = OP3, "(", LOCS_MULT, ",", NUM, ",", STRUCT, ")";

OP4_CONSTRUCT = "(", LOCS_MULT, ",", CLOSED_CLASS, ")";

OP5_CONSTRUCT = COMPD, {"+", COMPD}, "+", CLOSED_CLASS;

CLOSED_CLASS = CLOSED_CLASS_PRED, "(", CLOSED_CLASS_MORPH, ")";

PREF_CONSTRUCT = LOCS_MULT, "-", PREFS
               | LOCS_MULT, "-", COMPD;

SUFF_CONSTRUCT = LOCS_MULT, "-", SUFF
               | LOCS_MULT, "-", ADJSUFF;

STRUCT = PAR_PHR
       | NUM, "*", ELEMENTS;

LOCS_MULT = NUM, "*", "[", NUMS, "]"
          | UNKNO, "*", "[", NUMS, "]"
          | NUM, "*", "[", UNKNO, "]"
          | UNKNO, "*", "[", UNKNO, "]";

PREFS = PREF
      | ELEMENT, "-", PREF;    % sugar-specific (?)

ADJSUFF = ADJSUFF_PRED, "(", ADJSUFF_MORPH, ")",
          "+", CLASSNAME_PRED, "(", CLASSNAME_MORPH, ")";

ELEMENTS = ELEMENT
         | "(", ELEMENT, "+", ELEMENT, ")";

NUMS = NUM
     | NUM, {",", NUM};
```

B.2 Testsuite

B.2.1 File testsuite.txt

```
% ------------------------------------------- %
% File:     testsuite.txt                     %
% Author:   Stefanie Anstein, Gerhard Kremer  %
% Purpose:  testsuite for compd.pl            %
% ------------------------------------------- %

% allowed are:
% organic chemical compound names
% (w/ whitespaces at end of line, capitals, alpha, etc.),
% comment lines beginning with %,
% empty lines / lines containing only whitespace


%%% parents (nonsugar)

%% systematic names saturated, acyclic
hexane
phosphane
% replacement operation
pentaphosphane
% alternating elements
tetraarsazane

%% systematic names unsaturated
ethene
2-pentene
pent-2-ene
pent-2,3-diene
hex-3-yne
% double parent suffix
3-penten-1-yne

%% systematic names saturated, cyclic
cyclopentane
cyclopentaoxane
cyclotetraarsazane

% trivial names
benzene
% and corresponding systematic names
cyclohexatriene
cyclohex-1,3,5-triene

% semisystematic names
1,2-dihydrobenzene
% and corresponding systematic names
1,2-dihydrocyclohex-1,3,5-triene

% underspecified names
ethene
butene

% radicals
methyl


%%% organic compounds (nonsugar)
% multi-word; radiofunctional nomenclature
methyl alcohol
ethyl methyl ketone

% w/ prefix(es) (substitutive operation)
2-oxahexane
2,4,8-trioxaundecane
oxacyclopentane
2,4-dioxa-6,8-diphosphadecane
1,2-didehydrocyclohex-1,3,5-triene
1-methyl-hexane
1-(methyl)-hexane
1,2-dimethyl-hexane
2,3-dihydropent-2-ene
```

```
2,3-dihydropent-2-yne
1,2-didehydropent-1-ene
1,2-didehydropentane
5-hydroxypentane
5-deoxypentan-5-ol

% w/ suffix
methanol
propanal
propan-2-al
propan-2,3-dial
butanone
pentanoic acid
hexanaldehyde
hexancarbaldehyde
pentanoic anhydride
heptanamine
octanamide
octancarboxamide
methanolate

% w/ prefix(es) and suffix
2-oxahex-4-enol


%%% miscellaneous
% synonymous names
2-aminopentane
pent-2-yl-amine
pentane-2-amine
2-propylamine
propaneamine
2-propyl-2-amine

% class names
alkene
2-alkene
2,3-dialkene

% stereochemistry/orientation
cis-but-2-ene
(E)-but-2-ene

% element names
oxygen


% ------------------------------------------- %


%% sugar parents %%
hexose
hexodialdose
% altern. nomenclature (CAS):
3-pentulose
%
pent-2-ulose
hepto-2,3-diulose
hexos-3-ulose
hexopyranose
hex-3-ulooxirose
hexodialdo-6,3-furanose
hexodialdo-6,3-furanose-1,5-pyranose
hexopyranos-4-ulose
hexos-2-ulo-2,5-furanose


%% par_sugar w/ configuration %%
D-glycero-L-gulo-heptose
L-threo-tetrodialdose
D-arabino-hex-2-ulose
alpha-D-altro-hept-2-ulopyranose
meso-xylo-hepto-2,6-diulose
alpha-D-threo-Hexo-2,4-diulo-2,5-furanose
L-glycero-L-manno-Nono-2,7-diulose
D-arabino-hexos-3-ulose
```

```
%% deoxy sugars %%
% ... systematic names: %
2-Deoxy-alpha-D-allo-heptopyranose
4-Deoxy-beta-D-xylo-hexopyranose
2-Deoxy-D-ribo-hexose
1-Deoxy-L-glycero-D-altro-oct-2-ulose
% ... chiral centres divided into two centres: %
3-Deoxy-D-ribo-hexose
5-Deoxy-D-arabino-hept-3-ulose
6-Deoxy-L-gluco-oct-2-ulose
% ... derived from trivial names %
2-deoxyribose
2-Deoxy-D-erythro-pentofuranose


%% amino sugars %%
2-amino-2-deoxy-D-galacto-hexose
% ... other amino-prefixes %
%
% ... w/ triv_name-parent %
2-amino-2-deoxy-D-galactose
2-amino-2,6-dideoxy-D-glucose
2,4-diamino-2,4,6-trideoxy-D-glucose


%% thio sugars and other chalcogen analogues %%
4-Thio-beta-D-galacto-hexopyranose
% ... w/ triv_name-Teil %
4-Thio-beta-D-galactopyranose
5-Thio-beta-D-glucopyranose


%% trivial names %%
% ... trivial names w/o configuration %
glucose
glucopyranose
% ... trivial names w/ configuration %
D-ribose
L-Ribose
D-ribulose
D-fructose
alpha-D-glucooxirose


%% underspecified/class names %%
pyranose
ketose
aldohexose
%deoxy sugar
deoxypentose
% aminodeoxypentose %
amino-3-deoxypentose

%% embedded compound %%
% made-up name; for testing SMILES
2-deoxy-2-methyl-hexose
```

B.2.2 File `testsuite.out`

```
hexane
[h,e,x,a,n,e]
compd(ane(6*C),pref([]),suff([]))
NO SMILES
ALKANE
---

phosphane
[p,h,o,s,p,h,a,n,e]
compd(ane(1*P),pref([]),suff([]))
NO SMILES
ALKANE
---

pentaphosphane
[p,e,n,t,a,p,h,o,s,p,h,a,n,e]
compd(ane(5*P),pref([]),suff([]))
NO SMILES
ALKANE
---

tetraarsazane
[t,e,t,r,a,a,r,s,a,z,a,n,e]
compd(ane(4*(As+N)),pref([]),suff([]))
NO SMILES
ALKANE
---

ethene
[e,t,h,e,n,e]
compd(ene(?? *[??],ane(2*C)),pref([]),suff([]))
NO SMILES
ALKENE (double bond)
---

2-pentene
[2,-,p,e,n,t,e,n,e]
compd(ene(?? *[2],ane(5*C)),pref([]),suff([]))
NO SMILES
ALKENE (double bond)
---

pent-2-ene
[p,e,n,t,-,2,-,e,n,e]
compd(ene(?? *[2],ane(5*C)),pref([]),suff([]))
NO SMILES
ALKENE (double bond)
---

pent-2,3-diene
[p,e,n,t,-,2,,,3,-,d,i,e,n,e]
compd(ene(2*[2,3],ane(5*C)),pref([]),suff([]))
NO SMILES
ALKENE (double bond)
---

hex-3-yne
[h,e,x,-,3,-,y,n,e]
compd(yne(?? *[3],ane(6*C)),pref([]),suff([]))
NO SMILES
ALKYNE (triple bond)
---

3-penten-1-yne
[3,-,p,e,n,t,e,n,-,1,-,y,n,e]
compd(yne(?? *[1],ene(?? *[3],ane(5*C))),pref([]),suff([]))
NO SMILES
ALKENE (double bond),ALKYNE (triple bond)
---

cyclopentane
[c,y,c,l,o,p,e,n,t,a,n,e]
compd(cyclo(?? *[??],ane(5*C)),pref([]),suff([]))
NO SMILES
```

```
CYCLOALKANE
---

cyclopentaoxane
[c,y,c,l,o,p,e,n,t,a,o,x,a,n,e]
compd(cyclo(?? *[??],ane(5*O)),pref([]),suff([]))
NO SMILES
CYCLOALKANE
---

cyclotetraarsazane
[c,y,c,l,o,t,e,t,r,a,a,r,s,a,z,a,n,e]
compd(cyclo(?? *[??],ane(4*(As+N))),pref([]),suff([]))
NO SMILES
CYCLOALKANE
---

benzene
[b,e,n,z,e,n,e]
compd(triv_name(benzene),pref([]),suff([]))
NO SMILES
AROMATIC
---

cyclohexatriene
[c,y,c,l,o,h,e,x,a,t,r,i,e,n,e]
compd(cyclo(?? *[??],ene(6*[??],ane(3*C))),pref([]),suff([]))
NO SMILES
CYCLOALKENE
---
compd(cyclo(?? *[??],ene(3*[??],ane(6*C))),pref([]),suff([]))
NO SMILES
CYCLOALKENE
---

cyclohex-1,3,5-triene
[c,y,c,l,o,h,e,x,-,1,,,3,,,5,-,t,r,i,e,n,e]
compd(cyclo(?? *[??],ene(3*[1,3,5],ane(6*C))),pref([]),suff([]))
NO SMILES
CYCLOALKENE
---

1,2-dihydrobenzene
[1,,,2,-,d,i,h,y,d,r,o,b,e,n,z,e,n,e]
compd(triv_name(benzene),pref([2*[1,2]-hydro]),suff([]))
NO SMILES
NO CLASSES
---

1,2-dihydrocyclohex-1,3,5-triene
[1,,,2,-,d,i,h,y,d,r,o,c,y,c,l,o,h,e,x,-,1,,,3,,,5,-,t,r,i,e,n,e]
compd(cyclo(?? *[??],ene(3*[1,3,5],ane(6*C))),pref([2*[1,2]-hydro]),suff([]))
NO SMILES
ALKANE
---

ethene
[e,t,h,e,n,e]
compd(ene(?? *[??],ane(2*C)),pref([]),suff([]))
NO SMILES
ALKENE (double bond)
---

butene
[b,u,t,e,n,e]
compd(ene(?? *[??],ane(4*C)),pref([]),suff([]))
NO SMILES
ALKENE (double bond)
---

methyl
[m,e,t,h,y,l]
compd(yl(?? *[??],ane(1*C)),pref([]),suff([]))
C([H])([H])([H])
ALKYL
---
```

```
methyl alcohol
[m,e,t,h,y,l, ,a,l,c,o,h,o,l]
compd(compd(yl(?? *[??],ane(1*C)),pref([]),suff([]))+class_name(alcohol),pref([]),suff([]))
NO SMILES
alcohol
---

ethyl methyl ketone
[e,t,h,y,l, ,m,e,t,h,y,l, ,k,e,t,o,n,e]
compd(compd(yl(?? *[??],ane(2*C)),pref([]),suff([]))+compd(yl(?? *[??],ane(1*C)),pref([]),suff
     ([]))+class_name(ketone),pref([]),suff([]))
NO SMILES
ketone
---

2-oxahexane
[2,-,o,x,a,h,e,x,a,n,e]
compd(ane(6*C),pref([?? *[2]-oxa]),suff([]))
NO SMILES
ALKANE
---

2,4,8-trioxaundecane
[2,,,4,,,8,-,t,r,i,o,x,a,u,n,d,e,c,a,n,e]
compd(ane(11*C),pref([3*[2,4,8]-oxa]),suff([]))
NO SMILES
ALKANE
---

oxacyclopentane
[o,x,a,c,y,c,l,o,p,e,n,t,a,n,e]
compd(cyclo(?? *[??],ane(5*C)),pref([?? *[??]-oxa]),suff([]))
NO SMILES
ALKANE
---

2,4-dioxa-6,8-diphosphadecane
[2,,,4,-,d,i,o,x,a,-,6,,,8,-,d,i,p,h,o,s,p,h,a,d,e,c,a,n,e]
compd(ane(10*C),pref([2*[2,4]-oxa,2*[6,8]-phospha]),suff([]))
NO SMILES
ALKANE
---

1,2-didehydrocyclohex-1,3,5-triene
[1,,,2,-,d,i,d,e,h,y,d,r,o,c,y,c,l,o,h,e,x,-,1,,,3,,,5,-,t,r,i,e,n,e]
compd(cyclo(?? *[??],ene(3*[1,3,5],ane(6*C))),pref([2*[1,2]-dehydro]),suff([]))
NO SMILES
ALKYNE (triple bond)
---

1-methyl-hexane
[1,-,m,e,t,h,y,l,-,h,e,x,a,n,e]
compd(ane(6*C),pref([?? *[1]-compd(yl(?? *[??],ane(1*C)),pref([]),suff([]))]),suff([]))
NO SMILES
ALKANE
---

1-(methyl)-hexane
[1,-,(,m,e,t,h,y,l,),-,h,e,x,a,n,e]
compd(ane(6*C),pref([?? *[1]-compd(yl(?? *[??],ane(1*C)),pref([]),suff([]))]),suff([]))
NO SMILES
ALKANE
---

1,2-dimethyl-hexane
[1,,,2,-,d,i,m,e,t,h,y,l,-,h,e,x,a,n,e]
compd(ane(6*C),pref([2*[1,2]-compd(yl(?? *[??],ane(1*C)),pref([]),suff([]))]),suff([]))
NO SMILES
ALKANE
---

2,3-dihydropent-2-ene
[2,,,3,-,d,i,h,y,d,r,o,p,e,n,t,-,2,-,e,n,e]
compd(ene(?? *[2],ane(5*C)),pref([2*[2,3]-hydro]),suff([]))
NO SMILES
```

```
ALKANE
---

2,3-dihydropent-2-yne
[2,,,3,-,d,i,h,y,d,r,o,p,e,n,t,-,2,-,y,n,e]
compd(yne(?? *[2],ane(5*C)),pref([2*[2,3]-hydro]),suff([]))
NO SMILES
ALKENE (double bond)
---

1,2-didehydropent-1-ene
[1,,,2,-,d,i,d,e,h,y,d,r,o,p,e,n,t,-,1,-,e,n,e]
compd(ene(?? *[1],ane(5*C)),pref([2*[1,2]-dehydro]),suff([]))
NO SMILES
ALKYNE (triple bond)
---

1,2-didehydropentane
[1,,,2,-,d,i,d,e,h,y,d,r,o,p,e,n,t,a,n,e]
compd(ane(5*C),pref([2*[1,2]-dehydro]),suff([]))
NO SMILES
ALKENE (double bond)
---

5-hydroxypentane
[5,-,h,y,d,r,o,x,y,p,e,n,t,a,n,e]
compd(ane(5*C),pref([?? *[5]-hydroxy]),suff([]))
NO SMILES
ALCOHOL (-OH),ALKANE,PRIMARY ALCOHOL (-OH)
---

5-deoxypentan-5-ol
[5,-,d,e,o,x,y,p,e,n,t,a,n,-,5,-,o,l]
compd(ane(5*C),pref([?? *[5]-deoxy]),suff([?? *[5]-ol]))
NO SMILES
ALKANE
---

methanol
[m,e,t,h,a,n,o,l]
compd(ane(1*C),pref([]),suff([?? *[??]-ol]))
NO SMILES
ALCOHOL (-OH),ALKANE,PRIMARY ALCOHOL (-OH)
---

propanal
[p,r,o,p,a,n,a,l]
compd(ane(3*C),pref([]),suff([?? *[??]-al]))
NO SMILES
ALKANE,ALDEHYDE (-CHO)
---

propan-2-al
[p,r,o,p,a,n,-,2,-,a,l]
compd(ane(3*C),pref([]),suff([?? *[2]-al]))
NO SMILES
ALKANE,ALDEHYDE (-CHO)
---

propan-2,3-dial
[p,r,o,p,a,n,-,2,,,3,-,d,i,a,l]
compd(ane(3*C),pref([]),suff([2*[2,3]-al]))
NO SMILES
ALKANE,ALDEHYDE (-CHO)
---

butanone
[b,u,t,a,n,o,n,e]
compd(ane(4*C),pref([]),suff([?? *[??]-one]))
NO SMILES
ALKANE,KETONE (>(C)=O)
---

pentanoic acid
[p,e,n,t,a,n,o,i,c, ,a,c,i,d]
compd(ane(5*C),pref([]),suff([?? *[??]-(adj_suff(oic)+class_name(acid))]))
```

```
NO SMILES
acid,ALKANE
---

hexanaldehyde
[h,e,x,a,n,a,l,d,e,h,y,d,e]
compd(ane(6*C),pref([]),suff([?? *[??]-aldehyde]))
NO SMILES
ALKANE,ALDEHYDE (-CHO)
---

hexancarbaldehyde
[h,e,x,a,n,c,a,r,b,a,l,d,e,h,y,d,e]
compd(ane(6*C),pref([]),suff([?? *[??]-carbaldehyde]))
NO SMILES
ALKANE,ALDEHYDE (-CHO)
---

pentanoic anhydride
[p,e,n,t,a,n,o,i,c, ,a,n,h,y,d,r,i,d,e]
compd(ane(5*C),pref([]),suff([?? *[??]-(adj_suff(oic)+class_name(anhydride))]))
NO SMILES
anhydride,ALKANE
---

heptanamine
[h,e,p,t,a,n,a,m,i,n,e]
compd(ane(7*C),pref([]),suff([?? *[??]-amine]))
NO SMILES
ALKANE,AMINE (-NH2)
---

octanamide
[o,c,t,a,n,a,m,i,d,e]
compd(ane(8*C),pref([]),suff([?? *[??]-amide]))
NO SMILES
ALKANE,AMIDE (-CO-NH2)
---

octancarboxamide
[o,c,t,a,n,c,a,r,b,o,x,a,m,i,d,e]
compd(ane(8*C),pref([]),suff([?? *[??]-carboxamide]))
NO SMILES
ALKANE,AMIDE (-CO-NH2)
---

methanolate
[m,e,t,h,a,n,o,l,a,t,e]
compd(ane(1*C),pref([]),suff([?? *[??]-olate]))
NO SMILES
ALKANE,ALCOHOLATE,PHENOLATE
---
compd(ane(1*C),pref([]),suff([?? *[??]-ol,?? *[??]-ate]))
NO SMILES
ALKANE,ALCOHOLATE,PHENOLATE
---

2-oxahex-4-enol
[2,-,o,x,a,h,e,x,-,4,-,e,n,o,l]
compd(ene(?? *[4],ane(6*C)),pref([?? *[2]-oxa]),suff([?? *[??]-ol]))
NO SMILES
ALCOHOL (-OH),ALKENE,PRIMARY ALCOHOL (-OH)
---

2-aminopentane
[2,-,a,m,i,n,o,p,e,n,t,a,n,e]
compd(ane(5*C),pref([?? *[2]-amino]),suff([]))
NO SMILES
ALKANE,AMINE (-NH2)
---

pent-2-yl-amine
[p,e,n,t,-,2,-,y,l,-,a,m,i,n,e]
compd(yl(?? *[2],ane(5*C)),pref([]),suff([?? *[??]-amine]))
NO SMILES
ALKANE,AMINE (-NH2)
```

```
---

pentane-2-amine
[p,e,n,t,a,n,e,-,2,-,a,m,i,n,e]
compd(ane(5*C),pref([]),suff([?? *[2]-amine]))
NO SMILES
ALKANE,AMINE (-NH2)
---

2-propylamine
[2,-,p,r,o,p,y,l,a,m,i,n,e]
compd(yl(?? *[2],ane(3*C)),pref([]),suff([?? *[??]-amine]))
NO SMILES
ALKANE,AMINE (-NH2)
---

propaneamine
[p,r,o,p,a,n,e,a,m,i,n,e]
compd(ane(3*C),pref([]),suff([?? *[??]-amine]))
NO SMILES
ALKANE,AMINE (-NH2)
---

2-propyl-2-amine
[2,-,p,r,o,p,y,l,-,2,-,a,m,i,n,e]
compd(yl(?? *[2],ane(3*C)),pref([]),suff([?? *[2]-amine]))
NO SMILES
ALKANE,AMINE (-NH2)
---

alkene
[a,l,k,e,n,e]
compd(class_name(alkene),pref([]),suff([]))
NO SMILES
alkene
---

2-alkene
[2,-,a,l,k,e,n,e]
compd((?? *[2],class_name(alkene)),pref([]),suff([]))
NO SMILES
alkene
---

2,3-dialkene
[2,,,3,-,d,i,a,l,k,e,n,e]
compd((2*[2,3],class_name(alkene)),pref([]),suff([]))
NO SMILES
alkene
---

cis-but-2-ene
[c,i,s,-,b,u,t,-,2,-,e,n,e]
compd(cis(ene(?? *[2],ane(4*C))),pref([]),suff([]))
NO SMILES
ALKENE (double bond)
---

(E)-but-2-ene
[(,e,),-,b,u,t,-,2,-,e,n,e]
compd(stereo:e(ene(?? *[2],ane(4*C))),pref([]),suff([]))
NO SMILES
ALKENE (double bond)
---

oxygen
[o,x,y,g,e,n]
compd(elem_name(oxygen),pref([]),suff([]))
NO SMILES
NO CLASSES
---

hexose
[h,e,x,o,s,e]
compd(ose(?? *[??],6*C),pref([]),suff([]))
C(=O)C([H])(O)C([H])(O)C([H])(O)C([H])(O)C([H])(O)
```

```
NO CLASSES
---

hexodialdose
[h,e,x,o,d,i,a,l,d,o,s,e]
compd(ose(2*[??],6*C),pref([]),suff([]))
C(=O)C([H])(O)C([H])(O)C([H])(O)C([H])(O)C(=O)
NO CLASSES
---

3-pentulose
[3,-,p,e,n,t,u,l,o,s,e]
compd(ulose(?? *[3],5*C),pref([]),suff([]))
C([H])(O)C([H])(O)C(=O)C([H])(O)C([H])(O)
NO CLASSES
---

pent-2-ulose
[p,e,n,t,-,2,-,u,l,o,s,e]
compd(ulose(?? *[2],5*C),pref([]),suff([]))
C([H])(O)C(=O)C([H])(O)C([H])(O)C([H])(O)
NO CLASSES
---

hepto-2,3-diulose
[h,e,p,t,o,-,2,,,3,-,d,i,u,l,o,s,e]
compd(ulose(2*[2,3],7*C),pref([]),suff([]))
C([H])(O)C(=O)C(=O)C([H])(O)C([H])(O)C([H])(O)C([H])(O)
NO CLASSES
---

hexos-3-ulose
[h,e,x,o,s,-,3,-,u,l,o,s,e]
compd(ulose(?? *[3],ose(?? *[??],6*C)),pref([]),suff([]))
C(=O)C([H])(O)C(=O)C([H])(O)C([H])(O)C([H])(O)
NO CLASSES
---

hexopyranose
[h,e,x,o,p,y,r,a,n,o,s,e]
compd(anose(?? *[??],6,ose(?? *[??],6*C)),pref([]),suff([]))
C1C([H])(O)C([H])(O)C([H])(O)C(O1)C([H])(O)
NO CLASSES
---

hex-3-ulooxirose
[h,e,x,-,3,-,u,l,o,o,x,i,r,o,s,e]
compd(anose(?? *[??],3,ulose(?? *[3],6*C)),pref([]),suff([]))
C([H])(O)C([H])(O)C3C(O3)C([H])(O)C([H])(O)
NO CLASSES
---

hexodialdo-6,3-furanose
[h,e,x,o,d,i,a,l,d,o,-,6,,,3,-,f,u,r,a,n,o,s,e]
compd(anose(?? *[6,3],5,ose(2*[??],6*C)),pref([]),suff([]))
C(=O)C([H])(O)C3C([H])(O)C([H])(O)C(O3)
NO CLASSES
---

hexodialdo-6,3-furanose-1,5-pyranose
[h,e,x,o,d,i,a,l,d,o,-,6,,,3,-,f,u,r,a,n,o,s,e,-,1,,,5,-,p,y,r,a,n,o,s,e]
compd(anose(?? *[6,3],5,anose(?? *[1,5],6,ose(2*[??],6*C))),pref([]),suff([]))
C1C([H])(O)C3C([H])(O)C(O1)C(O3)
NO CLASSES
---

hexopyranos-4-ulose
[h,e,x,o,p,y,r,a,n,o,s,-,4,-,u,l,o,s,e]
compd(ulose(?? *[4],anose(?? *[??],6,ose(?? *[??],6*C))),pref([]),suff([]))
C1C([H])(O)C([H])(O)C(=O)C(O1)C([H])(O)
NO CLASSES
---

hexos-2-ulo-2,5-furanose
[h,e,x,o,s,-,2,-,u,l,o,-,2,,,5,-,f,u,r,a,n,o,s,e]
compd(anose(?? *[2,5],5,ulose(?? *[2],ose(?? *[??],6*C))),pref([]),suff([]))
```

```
C(=O)C2C([H])(O)C([H])(O)C(O2)C([H])(O)
NO CLASSES
---

D-glycero-L-gulo-heptose
[d,-,g,l,y,c,e,r,o,-,l,-,g,u,l,o,-,h,e,p,t,o,s,e]
compd(ose(?? *[??],7*C),pref([cfg([D-glycero,L-gulo])]),suff([]))
C(=O)[C@]([H])(O)[C@@]([H])(O)[C@]([H])(O)[C@@]([H])(O)C([H])(O)C([H])(O)
NO CLASSES
---

L-threo-tetrodialdose
[l,-,t,h,r,e,o,-,t,e,t,r,o,d,i,a,l,d,o,s,e]
compd(ose(2*[??],4*C),pref([cfg([L-threo])]),suff([]))
C(=O)[C@]([H])(O)[C@@]([H])(O)C(=O)
NO CLASSES
---

D-arabino-hex-2-ulose
[d,-,a,r,a,b,i,n,o,-,h,e,x,-,2,-,u,l,o,s,e]
compd(ulose(?? *[2],6*C),pref([cfg([D-arabino])]),suff([]))
C([H])(O)C(=O)[C@@]([H])(O)[C@]([H])(O)[C@]([H])(O)C([H])(O)
NO CLASSES
---

alpha-D-altro-hept-2-ulopyranose
[a,l,p,h,a,-,d,-,a,l,t,r,o,-,h,e,p,t,-,2,-,u,l,o,p,y,r,a,n,o,s,e]
compd(anose(?? *[??],6,ulose(?? *[2],7*C)),pref([cfg([alpha-D-altro])]),suff([]))
C([H])(O)[C@@]2(O)[C@@]([H])(O)[C@]([H])(O)[C@]([H])(O)[C@]([H])(O2)C([H])(O)
NO CLASSES
---

meso-xylo-hepto-2,6-diulose
[m,e,s,o,-,x,y,l,o,-,h,e,p,t,o,-,2,,,6,-,d,i,u,l,o,s,e]
compd(ulose(2*[2,6],7*C),pref([cfg([meso-xylo])]),suff([]))
C([H])(O)C(=O)[C@]([H])(O)[C@@]([H])(O)[C@]([H])(O)C(=O)C([H])(O)
NO CLASSES
---

alpha-D-threo-Hexo-2,4-diulo-2,5-furanose
[a,l,p,h,a,-,d,-,t,h,r,e,o,-,h,e,x,o,-,2,,,4,-,d,i,u,l,o,-,2,,,5,-,f,u,r,a,n,o,s,e]
compd(anose(?? *[2,5],5,ulose(2*[2,4],6*C)),pref([cfg([alpha-D-threo])]),suff([]))
C([H])(O)[C@@]2(O)[C@@]([H])(O)C(=O)C(O2)C([H])(O)
NO CLASSES
---

L-glycero-L-manno-Nono-2,7-diulose
[l,-,g,l,y,c,e,r,o,-,l,-,m,a,n,n,o,-,n,o,n,o,-,2,,,7,-,d,i,u,l,o,s,e]
compd(ulose(2*[2,7],9*C),pref([cfg([L-glycero,L-manno])]),suff([]))
C([H])(O)C(=O)[C@@]([H])(O)[C@]([H])(O)[C@@]([H])(O)[C@@]([H])(O)C(=O)C([H])(O)C([H])(O)
NO CLASSES
---

D-arabino-hexos-3-ulose
[d,-,a,r,a,b,i,n,o,-,h,e,x,o,s,-,3,-,u,l,o,s,e]
compd(ulose(?? *[3],ose(?? *[??],6*C)),pref([cfg([D-arabino])]),suff([]))
C(=O)[C@@]([H])(O)C(=O)[C@]([H])(O)[C@]([H])(O)C([H])(O)
NO CLASSES
---

2-Deoxy-alpha-D-allo-heptopyranose
[2,-,d,e,o,x,y,-,a,l,p,h,a,-,d,-,a,l,l,o,-,h,e,p,t,o,p,y,r,a,n,o,s,e]
compd(anose(?? *[??],6,ose(?? *[??],7*C)),pref([?? *[2]-deoxy,cfg([alpha-D-allo])]),suff([]))
[C@@]1([H])(O)[C@]([H])([H])[C@]([H])(O)[C@]([H])(O)[C@]([H])(O1)C([H])(O)C([H])(O)
NO CLASSES
---

4-Deoxy-beta-D-xylo-hexopyranose
[4,-,d,e,o,x,y,-,b,e,t,a,-,d,-,x,y,l,o,-,h,e,x,o,p,y,r,a,n,o,s,e]
compd(anose(?? *[??],6,ose(?? *[??],6*C)),pref([?? *[4]-deoxy,cfg([beta-D-xylo])]),suff([]))
[C@]1([H])(O)[C@]([H])(O)[C@@]([H])(O)[C@]([H])([H])[C@@]([H])(O1)C([H])(O)
NO CLASSES
---

2-Deoxy-D-ribo-hexose
[2,-,d,e,o,x,y,-,d,-,r,i,b,o,-,h,e,x,o,s,e]
```

```
compd(ose(?? *[??],6*C),pref([?? *[2]-deoxy,cfg([D-ribo])]),suff([]))
C(=O)[C@]([H])([H])[C@]([H])(O)[C@]([H])(O)C([H])(O)C([H])(O)
NO CLASSES
---

1-Deoxy-L-glycero-D-altro-oct-2-ulose
[1,-,d,e,o,x,y,-,l,-,g,l,y,c,e,r,o,-,d,-,a,l,t,r,o,-,o,c,t,-,2,-,u,l,o,s,e]
compd(ulose(?? *[2],8*C),pref([?? *[1]-deoxy,cfg([L-glycero,D-altro])]),suff([]))
C([H])([H])C(=O)[C@@]([H])(O)[C@]([H])(O)[C@]([H])(O)[C@]([H])(O)C([H])(O)C([H])(O)
NO CLASSES
---

3-Deoxy-D-ribo-hexose
[3,-,d,e,o,x,y,-,d,-,r,i,b,o,-,h,e,x,o,s,e]
compd(ose(?? *[??],6*C),pref([?? *[3]-deoxy,cfg([D-ribo])]),suff([]))
C(=O)[C@]([H])(O)[C@]([H])([H])[C@]([H])(O)C([H])(O)C([H])(O)
NO CLASSES
---

5-Deoxy-D-arabino-hept-3-ulose
[5,-,d,e,o,x,y,-,d,-,a,r,a,b,i,n,o,-,h,e,p,t,-,3,-,u,l,o,s,e]
compd(ulose(?? *[3],7*C),pref([?? *[5]-deoxy,cfg([D-arabino])]),suff([]))
C([H])(O)[C@@]([H])(O)C(=O)[C@]([H])(O)[C@]([H])([H])C([H])(O)C([H])(O)
NO CLASSES
---

6-Deoxy-L-gluco-oct-2-ulose
[6,-,d,e,o,x,y,-,l,-,g,l,u,c,o,-,o,c,t,-,2,-,u,l,o,s,e]
compd(ulose(?? *[2],8*C),pref([?? *[6]-deoxy,cfg([L-gluco])]),suff([]))
C([H])(O)C(=O)[C@@]([H])(O)[C@]([H])(O)[C@@]([H])(O)[C@@]([H])([H])C([H])(O)C([H])(O)
NO CLASSES
---

2-deoxyribose
[2,-,d,e,o,x,y,r,i,b,o,s,e]
compd(triv_name(ribose),pref([?? *[2]-deoxy]),suff([]))
C(=O)[C@]([H])([H])[C@]([H])(O)[C@]([H])(O)C([H])(O)
NO CLASSES
---

2-Deoxy-D-erythro-pentofuranose
[2,-,d,e,o,x,y,-,d,-,e,r,y,t,h,r,o,-,p,e,n,t,o,f,u,r,a,n,o,s,e]
compd(anose(?? *[??],5,ose(?? *[??],5*C)),pref([?? *[2]-deoxy,cfg([D-erythro])]),suff([]))
C1[C@]([H])([H])[C@]([H])(O)C(O1)C([H])(O)
NO CLASSES
---

2-amino-2-deoxy-D-galacto-hexose
[2,-,a,m,i,n,o,-,2,-,d,e,o,x,y,-,d,-,g,a,l,a,c,t,o,-,h,e,x,o,s,e]
compd(ose(?? *[??],6*C),pref([?? *[2]-amino,?? *[2]-deoxy,cfg([D-galacto])]),suff([]))
C(=O)[C@]([H])(N)[C@@]([H])(O)[C@@]([H])(O)[C@]([H])(O)C([H])(O)
NO CLASSES
---

2-amino-2-deoxy-D-galactose
[2,-,a,m,i,n,o,-,2,-,d,e,o,x,y,-,d,-,g,a,l,a,c,t,o,s,e]
compd(triv_name(galactose),pref([?? *[2]-amino,?? *[2]-deoxy,cfg([D])]),suff([]))
NO SMILES
NO CLASSES
---

2-amino-2,6-dideoxy-D-glucose
[2,-,a,m,i,n,o,-,2,,,6,-,d,i,d,e,o,x,y,-,d,-,g,l,u,c,o,s,e]
compd(triv_name(glucose),pref([?? *[2]-amino,2*[2,6]-deoxy,cfg([D])]),suff([]))
C(=O)[C@]([H])(N)[C@@]([H])(O)[C@]([H])(O)[C@]([H])(O)C([H])([H])
NO CLASSES
---

2,4-diamino-2,4,6-trideoxy-D-glucose
[2,,,4,-,d,i,a,m,i,n,o,-,2,,,4,,,6,-,t,r,i,d,e,o,x,y,-,d,-,g,l,u,c,o,s,e]
compd(triv_name(glucose),pref([2*[2,4]-amino,3*[2,4,6]-deoxy,cfg([D])]),suff([]))
C(=O)[C@]([H])(N)[C@@]([H])(O)[C@]([H])(N)[C@]([H])(O)C([H])([H])
NO CLASSES
---

4-Thio-beta-D-galacto-hexopyranose
```

```
[4,-,t,h,i,o,-,b,e,t,a,-,d,-,g,a,l,a,c,t,o,-,h,e,x,o,p,y,r,a,n,o,s,e]
compd(anose(?? *[??],6,ose(?? *[??],6*C)),pref([?? *[4]-thio,cfg([beta-D-galacto])]),suff([]))
[C@]1([H])(O)[C@]([H])(O)[C@@]([H])(O)[C@@]([H])(S)[C@@]([H])(O1)C([H])(O)
NO CLASSES
---

4-Thio-beta-D-galactopyranose
[4,-,t,h,i,o,-,b,e,t,a,-,d,-,g,a,l,a,c,t,o,p,y,r,a,n,o,s,e]
compd(anose(?? *[??],6,triv_name(galactose)),pref([?? *[4]-thio,cfg([beta-D])]),suff([]))
NO SMILES
NO CLASSES
---

5-Thio-beta-D-glucopyranose
[5,-,t,h,i,o,-,b,e,t,a,-,d,-,g,l,u,c,o,p,y,r,a,n,o,s,e]
compd(anose(?? *[??],6,triv_name(glucose)),pref([?? *[5]-thio,cfg([beta-D])]),suff([]))
NO SMILES
NO CLASSES
---

glucose
[g,l,u,c,o,s,e]
compd(triv_name(glucose),pref([]),suff([]))
C(=O)[C@]([H])(O)[C@@]([H])(O)[C@]([H])(O)[C@]([H])(O)C([H])(O)
NO CLASSES
---

glucopyranose
[g,l,u,c,o,p,y,r,a,n,o,s,e]
compd(anose(?? *[??],6,triv_name(glucose)),pref([]),suff([]))
NO SMILES
NO CLASSES
---

D-ribose
[d,-,r,i,b,o,s,e]
compd(triv_name(ribose),pref([cfg([D])]),suff([]))
C(=O)[C@]([H])(O)[C@]([H])(O)[C@]([H])(O)C([H])(O)
NO CLASSES
---

L-Ribose
[l,-,r,i,b,o,s,e]
compd(triv_name(ribose),pref([cfg([L])]),suff([]))
C(=O)[C@@]([H])(O)[C@@]([H])(O)[C@@]([H])(O)C([H])(O)
NO CLASSES
---

D-ribulose
[d,-,r,i,b,u,l,o,s,e]
compd(triv_name(ribulose),pref([cfg([D])]),suff([]))
NO SMILES
NO CLASSES
---

D-fructose
[d,-,f,r,u,c,t,o,s,e]
compd(triv_name(fructose),pref([cfg([D])]),suff([]))
C([H])(O)C(=O)[C@@]([H])(O)[C@]([H])(O)[C@]([H])(O)C([H])(O)
NO CLASSES
---

alpha-D-glucooxirose
[a,l,p,h,a,-,d,-,g,l,u,c,o,o,x,i,r,o,s,e]
compd(anose(?? *[??],3,triv_name(glucose)),pref([cfg([alpha-D])]),suff([]))
NO SMILES
NO CLASSES
---

pyranose
[p,y,r,a,n,o,s,e]

ketose
[k,e,t,o,s,e]
compd(class_name(ketose),pref([]),suff([]))
NO SMILES
```

```
ketose
---

aldohexose
[a,l,d,o,h,e,x,o,s,e]

deoxypentose
[d,e,o,x,y,p,e,n,t,o,s,e]
compd(ose(?? *[??],5*C),pref([?? *[??]-deoxy]),suff([]))
underspecified(C(=O)C([H])(O)C([H])(O)C([H])(O)C([H])(O),[??*{2,3,4,5}-deoxy])
NO CLASSES
---

amino-3-deoxypentose
[a,m,i,n,o,-,3,-,d,e,o,x,y,p,e,n,t,o,s,e]
compd(ose(?? *[??],5*C),pref([?? *[??]-amino,?? *[3]-deoxy]),suff([]))
underspecified(C(=O)C([H])(O)C([H])([H])C([H])(O)C([H])(O),[??*{3}-amino])
NO CLASSES
---

2-deoxy-2-methyl-hexose
[2,-,d,e,o,x,y,-,2,-,m,e,t,h,y,l,-,h,e,x,o,s,e]
compd(ose(?? *[??],6*C),pref([?? *[2]-deoxy,?? *[2]-compd(yl(?? *[??],ane(1*C)),pref([]),suff([])
      )]),suff([]))
C(=O)C([H])(C([H])([H])([H]))C([H])(O)C([H])(O)C([H])(O)C([H])(O)
NO CLASSES
---

2-Deoxy-D-erythro-pentofuranose
[2,-,d,e,o,x,y,-,d,-,e,r,y,t,h,r,o,-,p,e,n,t,o,f,u,r,a,n,o,s,e]
compd(anose(?? *[??],5,ose(?? *[??],5*C)),pref([?? *[2]-deoxy,cfg([D-erythro])]),suff([]))
C1[C@]([H])([H])[C@]([H])(O)C(O1)C([H])(O)
NO CLASSES
---
```