Balsa: A Compact Line Notation Based on SMILES

Richard L. Apodaca*

October 31, 2022

Abstract

Despite its widespread use, Simplified Molecular Input Line Entry System (SMILES) remains underspecified. The lack of a detailed specification encourages improvisation by software developers, complicates data standardization efforts, and undermines extension development. Balsa, a reformulation of SMILES, addresses these problems. A formal, machine-readable grammar defines Balsa's syntax. Semantics are described at three levels of resolution: constitution; configuration; and conformation. To estimate the compatibility of Balsa strings with SMILES software, all differences between the two languages were enumerated. To the extent that SMILES documentation is unambiguous, every difference was consistent with Balsa being a language subset.

Introduction

Simplified Molecular Input Line Entry System (SMILES) was first described by Weininger in 1988 [1]. As a line notation [2], SMILES represents molecules as single line character sequences, or strings. SMILES has since been widely adopted. Read and write functionalities are routinely supported by popular cheminformatics toolkits, including: Open Babel [3]; RDKit [4]; Chemistry Development Kit [5]; JChem Base [6]; the Daylight Toolkit [7]; and OEChem TK [8]. SMILES encodings can be found in many public-facing databases, including: PubChem [9]; ChEBI [10]; ZINC [11]; ChEMBL [12]; and Wikipedia [13]. Raw SMILES strings have been used extensively in machine learning applications [14]. Algorithmic selection of a single SMILES encoding has been used for molecular identification [15, 16]. SMILES has also been extended to carry various forms of metadata, as exemplified by Jmol SMILES [17], CurleySMILES [18], BigSMILES [19], and CXSMILES [20].

Despite the central role that SMILES plays in cheminformatics, public descriptions of the language remain imprecise. A 1987 report to the US Environmental Protection Agency by Weininger described a language similar to

*rapodaca@metamolecular.com

SMILES, but with important differences [21]. Weininger's 1988 paper offered an overview of the language, but omitted or only partially addressed points crucial to correctly implementing SMILES, including: (1) stereochemical configuration; (2) double bond conformation; (3) the encoding and decoding of "aromatic" features; (4) the computation of implicit hydrogen count, especially in the context of aromatic features; (5) syntax; (6) constraints on quantities such as mass number and charge; and (7) enumeration of error states.

Since 1988, two authoritative sources have attempted to clarify or revise Weininger's original description of SMILES. A 2003 book chapter by Weininger addressed some of the limitations of the paper [22]. The Daylight Theory Manual, a website maintained by SMILES's corporate sponsor, Daylight Chemical Information Systems, Inc. ("Daylight"), recapitulates published material [23]. The lack of other authoritative sources describing SMILES is consistent with Weininger's reported disinterest in traditional scientific publication [24].

In 2007 a documentation effort that would become known as OpenSMILES began [25]. OpenSMILES was conceived as "a non-proprietary specification for the SMILES language." Many of the points left open by previous SMILES documentation efforts were addressed. Noteworthy contributions included: the first formal grammar, later adapted to a standard parser toolchain [26]; many refinements around stereochemistry; and the introduction of "standard form." Absent were detailed procedures for composing and interpreting aromatic features, and a detailed procedure for computing implicit hydrogen count. OpenSMILES also left several points of syntactic and semantic ambiguity unsettled, while introducing extensions of its own.

In 2019 IUPAC announced the SMILES+ initiative [27]. Noting the limitations of existing SMILES documentation, the SMILES+ effort seeks to "establish a formalized recommended up-to-date specification of the SMILES format." SMILES+ took as its starting point the documentation produced by the OpenSMILES project. Efforts to extend this starting point are in progress online through a public repository, but no formal recommendation has to date resulted [28].

The lack of a comprehensive SMILES specification has caused several practical problems. Maintainers of SMILES implementations lack a blueprint for working toward a common set of features and behaviors. At the same time, authors of new SMILES implementations must address ambiguities on an ad hoc basis. The diversity of implementations combined with language underspecification leaves standards bodies with a limited pool of source material upon which to draw. Divergence at the edges of the SMILES language hampers the development of compliance suites, amplifying the problems faced by software teams trying to build compliant software. Finally, extensions to SMILES only make sense in the context of a rigorously defined base language.

More broadly, this situation undermines ongoing data integrity efforts underway in other scientific disciplines. As an example, the FAIR Guiding Principles identify four qualities essential for extracting maximum utility from published scholarly data: Findability; Accessibility; Interoperability; and Reusability [29]. The lack of precision in existing SMILES documentation means that chemical data sets based on SMILES will suffer in all four areas.

The very notion of a SMILES standard may not be realistic at this point. Today a dozen or more slightly different implementations are in use. None of them can be considered a reference standard. Some have introduced extensions not recognized by the others, contributing to language drift while compounding the already formidable difficulties faced by implementors. Others claim SMILES compatibility while ignoring certain aspects of the documentation that does exist. The Daylight implementation, like some of the others, is distributed under a commercial license that restricts use [7], and a publicly-accessible web service operated for many years was decommissioned [30]. Although SMILES may underpin a lot of modern chemical information exchange, the foundations of this system are weathered and showing signs of stress.

This paper attempts to solve these problems with Balsa, a fully-specified language designed as a subset of SMILES. The following sections present Balsa from three perspectives: as a compact system for molecular representation; as a text-based language; and as a specification for software capable of the lossless conveyance of chemical structure information across organizational and temporal boundaries.

Goals

Balsa's main goal is to solve the *molecular serialization* problem. Serialization refers to the process of translating a data structure into a format that can be transmitted over a communication channel for later reconstruction. This allows general-purpose devices and networks to convey complex domain-specific information. Receiver and recipient may be separated by time, space, or both. Molecular serialization extends the concept of serialization to molecular structure. Examples of molecular serialization formats other than SMILES include Chemical Markup Language (CML) [31], CDXML [32], and CTfile (e.g., "Molfile") [33]. Figure 1 presents some simple chemical structures together with their Balsa serializations.

A secondary goal is brevity. Molecules should be expressable using only a few characters for each atom and either zero or one character for each bond. Brevity benefits humans who manually encode chemical structures within interactive line-driven interfaces such as those found in terminal sessions and notebooks. Brevity also benefits computer handling, where the use of compact molecular representation can improve performance when storage, bandwidth, or processor cycles are limiting factors.

Molecular serialization schemes need to balance generality against complexity. As the diversity of the molecules that can be encoded increases, so do the rules needed to encode them and the software needed to interpret the rules. For this reason, Balsa's capabilities are directed at that region of chemical space populated by small organic molecules. Although Balsa can be used to represent a variety of other kinds of molecules, doing so in general risks data loss or corruption.

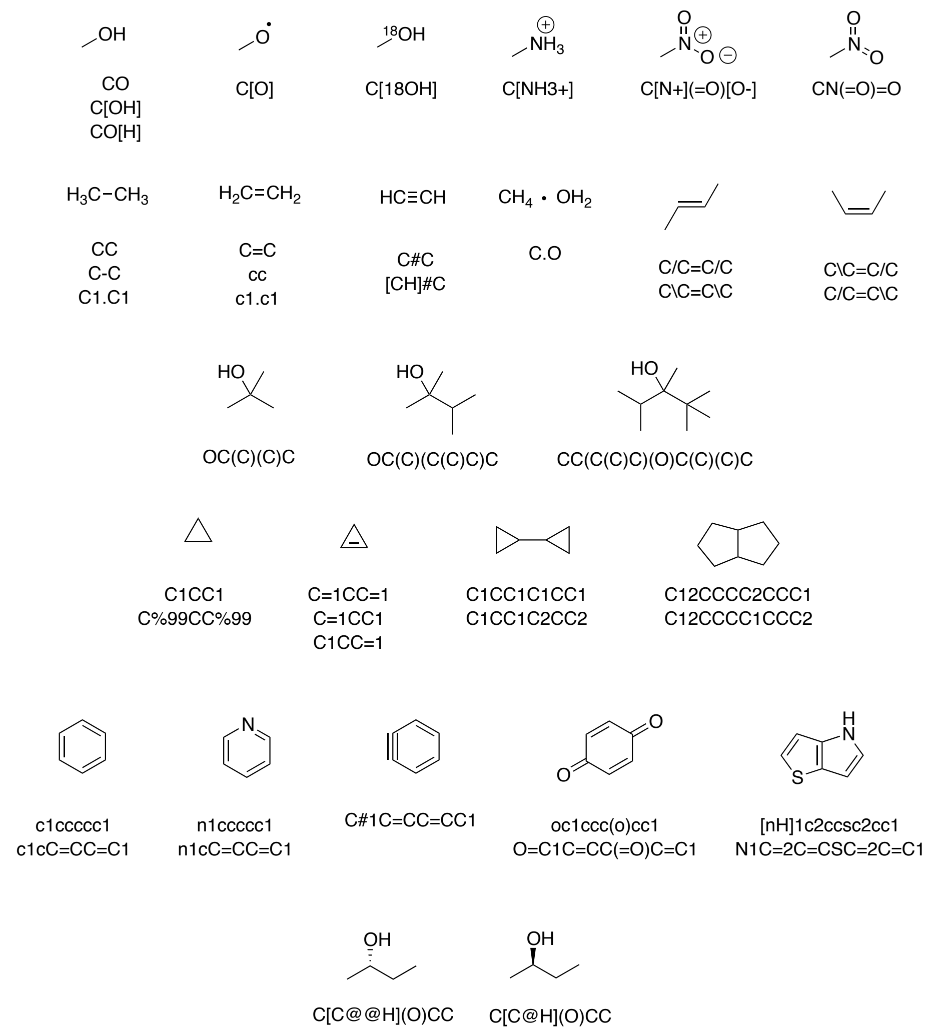


Figure 1: Representative Balsa examples.

Molecular Representation

Balsa's rules can be divided into two broad categories: *syntax* and *semantics*. Syntax defines the set of grammatically valid strings for a language, whereas semantics determines their meaning. Balsa semantics are complex due to the brevity of the syntax and the inherently complex nature of molecular representation.

Underpinning molecular serialization are conventions and abstractions for molecular representation. These are divided into three broad categories:

- *Constitution*, the atoms present in a molecule and their bonding relationships. Balsa's extends the electron-pair bonding model with a limited form of delocalized bonding.
- *Conformation*, a kind of stereoisomerism resulting from restricted rotation about a bond. Balsa supports restricted rotation about double bonds capable of producing alkene isomerism.
- *Configuration*, a kind of stereoisomerism resulting from the arrangement of neighboring atoms in three-dimensional space. Balsa supports configurations for atoms having tetrahedral symmetry attached to four neighboring atoms.

Molecular Tree

Balsa represents molecular structure as a *molecular tree* (Figure 2). A tree is a data structure comprised of a set of nodes and pairwise relationships between them called edges. Any two nodes in a tree are connected through exactly one path of alternating nodes and edges. As such, a tree contains no cycles and exactly one connected component. A molecular tree is a specialized tree onto which chemically-relevant metadata have been overlaid. A Balsa molecule is a molecular tree having zero or more nodes and zero or more edges. The empty molecule, devoid of atoms and bonds, is therefore allowed. There is no upper bound on the number of nodes or edges, although limitations in hardware and software may restrict maximum molecular size.

A related but more common concept is the *molecular graph* [34]. Unlike molecule graphs, molecular trees do not support cycles or disconnected components. This difference might appear to make molecular trees unsuitable for molecular representation, in which both rings and aggregation play important structural roles. However, these apparent shortcomings can be mitigated and even harnessed.

The main advantage of trees over graphs is information density. Graphs must accommodate cycles, a requirement fulfilled by assigning a unique identifier to each node. Each edge then references two of these identifiers. Graph representations therefore incur the storage overhead of identifiers associated with both nodes and edges. In contrast, a tree can be directly encoded as a set

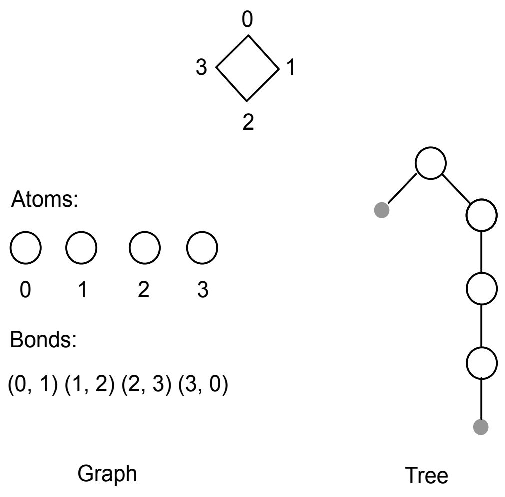


Figure 2: Molecular Graph vs. Molecular Tree. The bonds in a molecular graph reference atoms indirectly through unique indexes (lower left). In contrast, a bond in a molecular tree defines its target atom inline (lower right). This increases the potential efficiency of trees over graphs for molecular representation. To simulate ring closure, a molecular tree introduces bridge nodes (solid grey circles).

of nested, alternating paths of nodes and edges. This simplifies representation and eliminates the overhead of assigning and using node identifiers.

When discussing molecular trees, it's often useful to refer to *node order* (Figure 3). An order is a property of a set that allows its members to be sorted through pairwise comparisons. Node order is the order in which the nodes of a molecular tree are added through depth-first construction. A node added before another node is said to "precede" it. A node added after another node is said to "succeed" it. In this way nodes can be prioritized. The root atom, uniquely lacking a parent, is ordered first because it precedes every other node. Similarly, a parent always precedes its children. Finally, the first child added to a given parent precedes all of its siblings. This relationship is transitive. If a node precedes (or succeeds) a sibling, then so do all of its descendants.

To support cycles and disconnected components, a molecular tree decouples connectivity from bonding (Figure 4). This means that nodes and edges don't generally map to atoms and bonds as they often do in molecular graphs. This indirection enables rings and disconnections to be encoded despite the fact that trees can contain no cycles or disconnected components.

Cycles are supported through *bridges*. A bridge consists of two bond-node pairs that together represent one bond. On each side of a bridge an atom connects to a bond, but the two atoms are not connected to each other. Instead, each atom is succeeded by a *bridge node*. Unlike an atom, a bridge node has no children and so may not serve as the root of the tree. A bridge preserves the

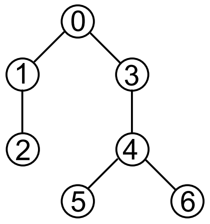


Figure 3: Node Order. Nodes in a molecular tree assume the order in which they were added through depth-first construction. Parents precede children, and children added first preceded those added later.

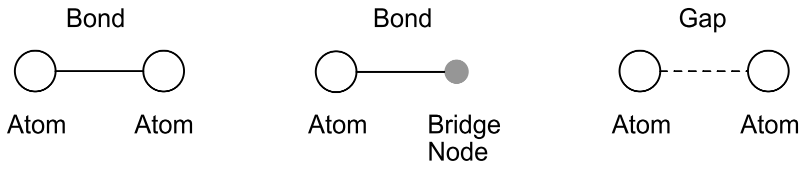


Figure 4: Nodes and Edge Associations. Three node/edge associations are possible within a molecular tree. A parent atom can be connected to a child atom through a bond (left). A parent atom can also be connected to a child bridge node through a bond (center). Finally, an atom can be connected, without bonding, to a child atom through a gap (right).

immediate atomic environment around each atom, allowing many of the same atomic computations possible in molecular graphs. Any bond, whether it closes a ring or not, can be represented as a bridge.

Each bridge node serves as a proxy to a *referenced atom.* Should a bridge node be present during an operation that requires an atom, the referenced atom is used instead. Although a bridge node always succeeds its parent, the referenced atom may precede or succeed that parent.

Multiple molecules are supported through *gaps.* A gap is an edge that defines no bonding relationship. Gaps can be used whenever two or more disconnected molecules need to be encoded within the same molecular tree. Specific applications include ionic bonding and non-ionic complexes. However, no bonding relationship whatsoever is implied by the presence of a gap. In particular, the zero-order bond interpretation is excluded [35].

Gaps and bridges decouple connectivity from bonding, but for opposite purposes. Whereas a gap connects two atoms without bonding them, a bridge creates a bond between two atoms without connecting them. These relationships can complicate the analysis of connectivity and bonding. For example, the atom succeeding a gap behaves in most situations as if it were a root. When tallying the bonds surrounding such an atom, only children are considered. Similar considerations apply when a gap succeeds an atom. Likewise, a bridge node

Table 1: Primitives.

Name	Description	Notation	Values
`Option`	Optional value	{`T`}	`None`, t
`List`	Ordered list of values	`[T]`	$t_0, t_1, ...t_n$
`Range`	Bounded integer range	m..n	from `m` to `n`, excluding `n`
`Boolean`	Boolean value	`?`	`true`, `false`

behaves as if it were an atom in the sense that it connects to an atom through a bond. Nevertheless, a bridge node merely stands in for a referenced atom found elsewhere in the tree. Bonding and connectivity analyses within molecular trees must therefore account for the possibility of connection-free bonds and bond-free connections.

Atoms and Bonds

Within a molecular tree, atoms and bonds play complementary roles. Atoms define nuclear particle counts, valence electron counts, and configuration. Bonds define bonding electron counts and conformation. Constitution, configuration, and conformation are all influenced by the individual and collective behavior of atoms and bonds.

At the lowest level, atom and bond behavior are governed by their respective *attributes*. An attribute is a key/value pair in which the key is a name composed of a character sequence and the value is constrained by a single, immutable *type*. Types can be composed, a feature supported by many programming languages.

Balsa types are composed of four *primitives* (Table 1). The first, `Option`, represents a value of a particular type that may either be present or absent. If not present, the special value `None` is used. A `List` is a possibly empty ordered collection of values of the same type. The `Range` type represents an integer value constrained by a lower inclusive bound (`m`) and an upper, exclusive bound (`n`). The `Boolean` type assumes one of two values: `true` or `false`.

An atom is composed of seven attributes (Table 2). Default values are applied at the time of atom creation if no other values are used. The specific values assigned to an atom collectively define an *atomic state*.

The attributes `element` ("element") and `isotope` ("isotope") define atomic number and mass number, respectively. Elements are selected from a subset of those approved by IUPAC [36] (Figure 5). An atom whose element is unknown must have an `element` attribute equal to `None`. The `isotope` attribute is an optional integer value representing an atom's nuclear mass number, defined as the sum of proton and neutron count. Setting the `isotope` property to `None` means that the element's isotopic composition equals natural abundance. Otherwise, the lower bound of the value of `isotope` is one. This lower limit allows for physically nonsensical states such as atoms of negative implied neutron count (e.g., carbon-5). Implementations may reject such atomic states as invalid.

Table 2: Atomic Attributes.

Attribute	Description	Type	Default
`element`	Elemental symbol	{`Element`}	`None`
`isotope`	Mass number	{`1..1000`}	`None`
`hydrogens`	Hydrogen count	`Implicit, 0..10`	0
`charge`	Formal charge	`-9..10`	0
`selected`	Selection status	`?`	`false`
`parity`	Configuration	{`AtomParity`}	`None`
`edges`	Edges directed toward child nodes	`[Edge]`	`[]`

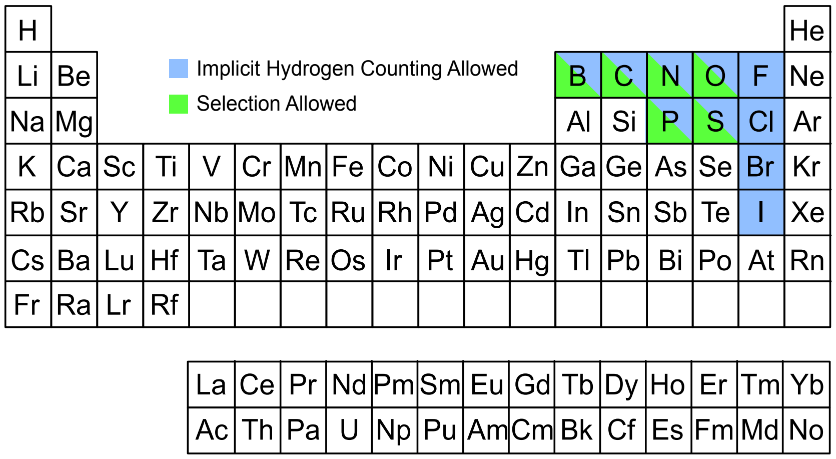

Implicit Hydrogen Counting Allowed
Selection Allowed
H He Li Be B C N O F Ne Na Mg Al Si P S Cl Ar
K Ca Sc Ti V Cr Mn Fe Co Ni Cu Zn Ga Ge As Se Br Kr
Rb Sr Y Zr Nb Mo Tc Ru Rh Pd Ag Cd In Sn Sb Te I Xe
Cs Ba Lu Hf Ta W Re Os Ir Pt Au Hg Tl Pb Bi Po At Rn
Fr Ra Lr Rf
La Ce Pr Nd Pm Sm Eu Gd Tb Dy Ho Er Tm Yb
Ac Th Pa U Np Pu Am Cm Bk Cf Es Fm Md No

Figure 5: Elements. Available chemical elements and their allowed roles. Not all IUPAC-approved elements are available. Hydrogen counts of atoms associated with elements on blue background can be computed algorithmically. Atoms associated with elements on green background can be selected.

Table 3: `AtomParity` Variants.

Variant	Description
`Clockwise`	Configuration runs with node order.
`Counterclockwise`	Configuration runs against node order.

The `hydrogens` attribute sets an atom's mode of *hydrogen suppression*. Hydrogen suppression replaces an atomic hydrogen of natural isotopic abundance and its associated bond with an integer tally associated with the parent atom. Even so, a suppressed hydrogen and its bond are understood to be present. The high relative abundance and regular bonding of hydrogen atoms makes hydrogen suppression an effective technique for increasing information density. Two mutually-exclusive types of suppressed hydrogen are supported: *virtual hydrogen* and *implicit hydrogen*.

A virtual hydrogen is one that appears as an integer unit contribution associated with the parent atom. For example, methane can be represented using five atoms and four bonds. But methane can also be represented as a single atom whose `hydrogens` attribute equals four. Virtual hydrogen count defaults to zero, and may be set to a maximum value of nine. By default, atoms use virtual hydrogen counting.

An implicit hydrogen is similar to a virtual hydrogen in that both the hydrogen atom and its bond have been replaced with a tally. Unlike a virtual hydrogen, however, this tally is not present as an attribute on the parent atom but rather as a computation. Setting the `hydrogens` attribute to `Implicit` activates implicit hydrogen counting. Not all atoms are eligible for this treatment (see: Computing Implicit Hydrogen Count).

It is sometimes convenient to interconvert virtual and atomic hydrogens. The transformation of an atomic hydrogen into a virtual hydrogen is called *virtualization*. The reverse process is called *reification*.

The `charge` ("charge") and `selected` ("selected") attributes determine valence electron count (see: Electron Counting). Charge refers to formal charge, or the difference between the valence electron count of the element and the sum of bonding and nonbonding electrons of the bound atom. By default, an atom is assigned a charge of zero. Minimum and maximum values for charge are -9 and +9, respectively. The `selected` attribute adjusts an atom's selection status. Selecting an atom allows it to participate in extended bonding (see: Delocalization Subgraph).

Configuration is determined by the `parity` attribute ("atom parity"). The type of this value, `AtomParity` is an enumeration supporting two values: `Clockwise` and `Counterclockwise` (Table 3). The interpretation and encoding of atom parity will be described in detail later (see: Configuration).

The `edges` ("edges") attribute is a possibly empty ordered list of edges associated with an atom. As noted in the previous section, an edge may represent either a bond or a gap (Table 4). The mere presence of an edge does not guarantee a bonding relationship.

Table 4: Edge Variants.

Variant	Description
`Bond`	A bonding and connectivity relationship.
`Gap`	A connectivity relationship only.

Table 5: Bond Attributes.

Attribute	Description	Type	Default
`order`	Formal bond order	`1..4`	1
`elided`	Elision status	`?`	`false`
`direction`	Partial conformational parity	{`Direction`}	`None`
`target`	An atom or bridge node	`Target`	-

Not all atomic states are valid. Implementations must ensure either the impossibility of creating an invalid state, or an error condition in the event that one is created. Atomic state is restricted in the following two ways:

1. If implicit hydrogen counting is used, default values of isotope, charge, and parity must also be used. Implicit hydrogen counting is only supported for elements in the list: B; C, N; O; P; S; F; Cl; Br; or I.

2. An atom can only be selected if its element attribute is one of: B; C; N; O; P; or S.

A bond is composed of four attributes (Table 5). The `order` attribute ("order") represents the concept of formal bond order, or the number of bonding electrons divided by two. A bond's order may only assume the values one, two, or three. When set to `true`, the boolean `elided` attribute ("elided") allows a bond to be omitted during serialization, and to participate in extended bonding. The `direction` attribute ("direction") is used with other information to conformationally restrict a bond. The two allowed variants for `Direction`, `Up` and `Down` (Table 6), will be described in detail later (see: Conformation). The `target` attribute identifies the child node at which the bond is directed (Table 7).

As with atomic state, not all bond states are valid. The following restrictions apply:

1. An elided bond must have an order of one and default direction.

Table 6: Bond Direction Variants.

Variant	Description
`Up`	The target node lies above the parent.
`Down`	The target node lies below the parent.

Table 7: Target Variants.

Variant	Description
`Atom`	A parent and child atom are bound.
`Bridge`	A parent atom and bridge node are bound.

2. A bond with a non-default direction must have an order of one.

The atom and bond attributes defined here will be referenced throughout this paper as a way to connect syntax and semantics. Implementations may, of course, use any suitable internal data model. They must, however, ensure that the model used is consistent with the one provided here.

Electron Counting

A molecular tree explicitly encodes the presence of protons and neutrons through the atomic attributes `element` and `isotope`, respectively. The presence of electrons, in contrast, is merely implied. The principles of charge and mass conservation nevertheless require a method to determine explicit electron counts. This is possible through *electron counting*.

Electron counting in Balsa is based on the well-known electron-pair bonding model developed by Lewis and others ("Lewis model") [37]. This model assumes that a bond spans exactly two atoms and is associated with a nonzero, positive, and even electron count. These electrons are drawn in equal quantities from each atom's set of valence electrons. A single bond draws one electron from each atom (two total), a double bond two (four total), and a triple bond three (six total). The electron count of an atom whose `element` attribute equals `None` is always zero. Bonding does not change it.

Electrons are counted as follows (Figure 6). A molecular tree starts as a single root atom. The electron count of this atom equals the difference between proton count (atomic number) and charge. Adding a bond is a three-step process. First, a child atom is constructed with an electron count determined in the same way as the root atom. Next, the child atom and the bond are added to the molecular tree. Finally, the electron counts of each atom and the bond are updated. If parent and child are connected through a gap edge, no electrons are deducted from either atom. Should a parent atom be bound to a child bridge node, then the electron count of the referenced atom is not changed.

Whereas negative bond order is disallowed by definition, Balsa places no restrictions on *hypervalence*. Hypervalence occurs when an atom undergoes enough bonding operations to leave it with a negative implied valence electron count. Consider lithium, which possesses one valence electron. Formation of one single bond leaves lithium with zero implied valence electrons. Application of a second bond formation leaves lithium with a zero charge and an implied valence electron count of -1. Such an arrangement may be physically meaningless, but Balsa explicitly supports it. Software using Balsa may or may not reject such species on semantic grounds.

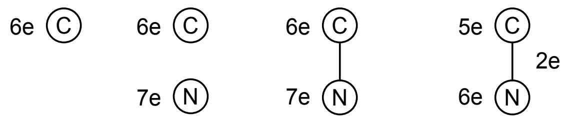


Figure 6: Electron Counting. A root atom and its child are added to a tree (left, center left). Next, a bond is added (center right). Finally, the appropriate electron count is deducted from each atom and added to the bond (right).

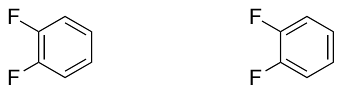


Figure 7: Delocalization-induced molecular equality (DIME). Two chemically equivalent molecular representations differ only due to delocalization.

Delocalization Subgraph

A molecular representation based solely on the Lewis model can yield artifacts resulting from *delocalization induced molecular equality* (DIME, Figure 7). DIME results from limitations of the Lewis model causing two representations to encode what would be considered the same molecule. When the two forms differ only in double bond placement, the terms "resonance" or "aromaticity" are sometimes used. These terms are avoided here due to their ambiguity [38, 39].

The main problem with DIME is its interference with *canonicalization*, or the selection of a single molecular representation among many alternatives. The presence of multiple equivalent molecular forms differing only in electron delocalization complicates the formulation of selection rules and invariants, which must be adapted to account for the artificial asymmetry.

To eliminate DIME and thereby streamline canonicalization, each molecular tree is augmented with a *delocalization subgraph* (DS). A DS is a possibly empty node-induced subgraph over a molecular tree. The membership of a DS will typically be drawn from the set of atoms and bonds that participate in DIME within a given molecular tree.

Membership of a DS is determined as follows. An atom is added to the DS by setting its `selected` attribute to `true` through a process called *selection*. As noted previously (see: Atoms and Bonds), only some atomic states are compatible with selection. A bond is added to the DS if both of the atoms it spans are selected and the bond itself is elided. This allows arbitrary bonds between selected atoms to be excluded from the DS by ensuring they are not elided. A bond to a bridge node will be added to the DS only if both the parent and the referenced atom are selected.

A non-empty DS must possess a *perfect matching* (Figure 8). A matching is a subgraph in which no two edges share a common node. Equivalently, a matching is a subgraph in which all nodes have degree one. A perfect matching

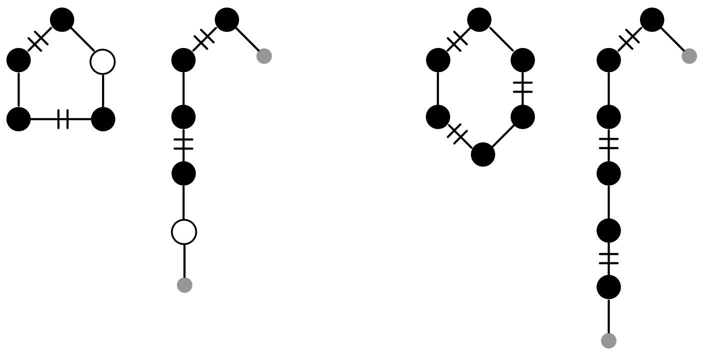


Figure 8: Perfect Matching. A matching is a subgraph in which all nodes have degree one. A perfect matching contains all of the nodes of its parent. Nodes and edges of a matching are noted with black circles and hashed lines, respectively. The five-membered cycle on the left has no perfect matching. Such graphs are not valid delocalization subgraphs. The six-membered cycle on the right has a perfect matching, and would be a valid delocalization subgraph. Either graph or tree representation can be used to construct a perfect matching, but published algorithms use graphs.

includes every node present in the parent graph. Every atom added to a DS must therefore become part of its perfect matching.

A filled DS can be emptied through a two-step process of *deselection* (Figure 9). First, a perfect matching over the DS is found. Next, the elided bond corresponding to each matched edge is replaced by a double bond. Because the presence of a filled DS implies a perfect matching over it, kekulization always succeeds. A widely-used algorithm for matching, the Edmunds "blossom algorithm," has a time complexity of $O(|E||V|^2)$, where $|E|$ is the number of edges and $|V|$ is the number of nodes [40]. Although more efficient algorithms are known, they are either much more difficult to implement or lack generality.

The opposite operation can be accomplished with a *selection algorithm.* A selection algorithm selects two or more atoms, thereby adding them to the DS. The only requirement for a selection algorithm is that the resulting DS must have a perfect matching. Depending on the application, other criteria may be applied. For example, a selection algorithm may restrict candidate atoms to those found in cycles. Electron-counting techniques may also be introduced to approximate the chemical concept of "aromaticity." Elision of the single bonds to be added to the DS completes the process.

Selection is nothing more than an alternative to explicit bond order encoding. No other properties whatsoever are conferred to selected atoms. The members of a DS may be assigned the labels "aromatic," "antiaromatic," "non-aromatic," or any number of other designations by downstream applications. Two encodings differing only in the presence of a DS should be considered equivalent. Balsa

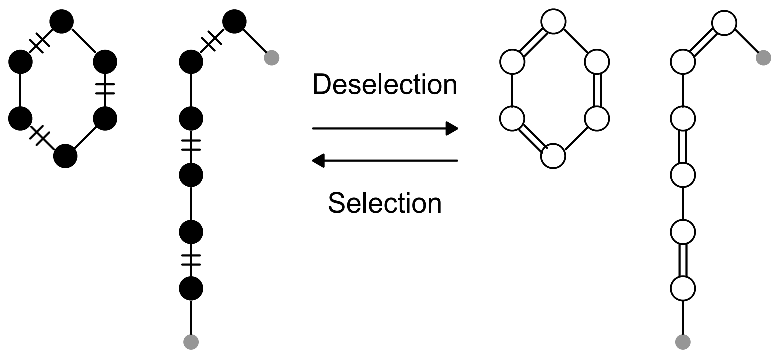


Figure 9: Deselection and Selection. Deselection unsets the atomic selected flag (open circle) with simultaneous promotion of matching bonds. Selection sets the atomic selected flag (closed circle) with simultaneous demotion of intervening single and double bonds to elided bonds.

imposes no rules around whether or not a selectable atom must be selected, although users may do so voluntarily.

Valence and Subvalence

To support implicit hydrogen counting, Balsa uses the concept of *valence*. Valence is a non-negative integer tally computed as the sum of bond orders at a given atom. Single and elided bonds contribute one to the tally, double bonds two, and triple bonds three. Each virtual hydrogen contributes one. For example, the valence of a methyl carbon having three virtual hydrogens and a hydrogen neighbor is four. The valence of an acetaldehyde oxygen atom is two. And so on.

Some elements are associated with one or more *default valences* (Table 8). A default valence is the number of hydrogens that can be attached to an isolated, fully-saturated atom using the corresponding element. For example, the default valence for a carbon-bearing atom is 4. This means that a fully saturated atom of carbon will have four hydrogens. Likewise, a fully-saturated atom of oxygen will have two hydrogens. Some elements such as nitrogen are associated with multiple default valences. In these cases, multiple saturated forms are possible. For example, nitrogen has the default valences three and five. Both ammonia (NH_3) and nitrogen pentahydride (NH_5) are therefore fully saturated forms of nitrogen according to Table 8.

Given an atom whose element has one or more default valences, *subvalence* can be computed (Algorithm 1). Subvalence is the number of hydrogens that can be added to an atom without exceeding the lowest possible default valence. If no suitable default valence exists, then subvalence equals zero. The algorithm begins by computing the atom's valence. Next, the ordered list of default va-

Table 8: Default Valences.

Element	Valences
B	3
C	4
N	3,5
O	2
F	1
P	3,5
S	2,4,6
Cl	1
Br	1
I	1

lences of the atom's element is found. For each default valence, the difference between it and the valence is compute. If this differences is positive, it is returned as the subvalence. Otherwise, the next default valence is considered. If no suitable default valence is found, zero is returned. The subvalence of an atom whose element has no default valences (i.e., no listing in Table 8) is zero. For example, an iron atom always has a subvalence of zero, regardless of its bonding.

```
input  : An atom $a$ with one or more default valences
output: The subvalence of $a$
begin
    $v \leftarrow$ Valence($a$);
    $T \leftarrow$ DefaultValences($a$);
    for $t \in T$ do
        $d \leftarrow t - v$;
        if $d >= 0$ then
            return $d$;
        end
    end
    return 0;
end
```

Algorithm 1: Computing subvalence.

For a representative subvalence computation, consider an isolated nitrogen atom without virtual hydrogens. The atom's bond order sum is zero. Its default valences are 3 and 5. The difference between the atom's bond order sum and the first default valence is found to be three $(3 - 0)$. Therefore, the subvalence of this atom is three.

The carbonyl carbon atom of acetaldehyde illustrates the effects of substitution and multiple bonding. Bond order sum is three $(2 + 1)$. From Table 8, default valence is four. Subtracting bond order sum from default valence yields one $(4 - 3)$, which is returned as the atom's subvalence.

The phosphorous atom in phosphorous acid (H_3PO_3) illustrates the use of Algorithm 1 for atoms having multiple target valences. Bond order sum is four $(2+1+1)$. The first default valence is 3, but subtracting valence from that value yields a negative number $(3-4)$. Continuing to the next default valence, five, a difference of 1 is obtained. Therefore, the subvalence of the phosphorous-bearing atom is reported as one.

The valence of some atoms exceeds the largest default valence. In these cases, subvalence is reported as zero. Consider sodium perchlorate ($NaClO_4$). The chlorine atom has a bond order sum of seven $(2+2+2+1)$. From Table 8, the only default valence for chlorine is one. Subtracting seven yields a negative number (-6). Therefore, the subvalence is reported as zero.

Computing Implicit Hydrogen Count

An atom whose `hydrogens` attribute equals `Implicit` signals that its hydrogens must be counted algorithmically. The exact algorithm depends on the value of the atoms's `selected` attribute (Figure 10).

For an unselected atom, implicit hydrogen count equals subvalence. For example, an oxygen atom with one single bond has an implicit hydrogen count of one because subvalence is one $(2 - 1)$. Similarly, an oxygen atom with two singly-bonded neighbors has an implicit hydrogen count of zero $(2 - 2)$.

For a selected atom, implicit hydrogen count equals subvalence minus one. This subtraction accounts for the extra valence implied by the atom's membership in the delocalization subgraph. Consider a selected carbon atom in benzene with a `hydrogens` attribute of `Implicit`. Subvalence equals two $(4 - 2)$, so implicit hydrogen count equals one $(2-1)$. Likewise, the subvalence of a selected nitrogen atom in pyridine equals one $(3 - 2)$ so implicit hydrogen count equals zero $(1 - 1)$. Stated differently, the calculation ensures that an atom's implicit hydrogen count is identical before and after deselection.

If the subvalence of a selected atom equals zero, then an implicit hydrogen count of zero is reported. The meaning of such an atomic state may seem suspect because an atom without a free valence can not perform the required promotion of an elided bond during deselection. As will be explained in detail later (Pruning), this situation can arise for reasons of convenience or tradition. Returning zero avoids miscalculation of the implicit hydrogen count.

An atom's implicit hydrogen count may or may not correlate with chemical intuition or experimental data. Consider the phosphorous-bearing atom of hypophosphorous acid ($HOP(O)H_2$). We might expect the implicit hydrogen count to equal the experimentally-determined hydrogen count (2). However, the subvalence for the phosphorous atom is found to be three $(2 + 1)$. The implicit hydrogen count is therefore zero $(3 - 3)$ rather than the expected two. To represent this atom and other like it, virtual hydrogens must be used.

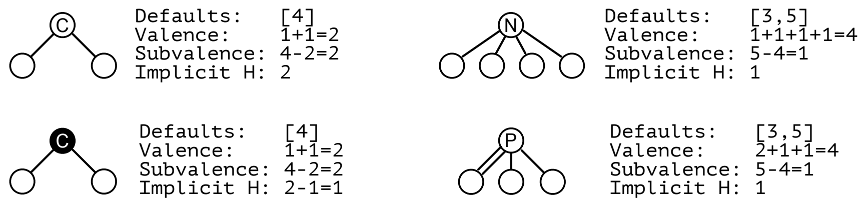


Figure 10: Implicit hydrogen count. The number of hydrogens is computed algorithmically. For unselected atoms (top left, top right, bottom right), implicit hydrogen count equals subvalence. For selected atoms (bottom left), implicit hydrogen count equals subvalence minus one. The hydrogen count for an atom whose element lacks at least one default valence can not be determined algorithmically.

Conformation

Restricted rotation about a bond leads to conformational isomers ("conformers"). When the bond in question is a double bond, two conformers result. These can be distinguished through *partial parity bonds* (PPBs). As the name implies, a PPB encodes some of the parity information characterizing a conformationally restricted double bond. Reconstruction of the full parity requires the double bond itself and at least one flanking PPB at each terminal.

A PPB is designated by setting its `direction` attribute to a non-default variant (`Up` or `Down`). The names of these variants refer to a two-dimensional geometrical model in which a parent atom and its child are placed along a vertical axis. If the child lies above the parent, the `Up` variant is used. If the child lies below the parent, the `Down` variant is used.

The parity of a double bond is determined by the relative direction of the PPBs neighboring it (Figure 11). First, the parent atom is identified. This atom has the lowest order in the assembly, meaning that it was added first to the molecular tree. The parent is placed on a vertical axis. Bonds from the parent atom are then iterated. If a PPB is found, its child atom is placed above the parent for a direction of `Up` or below the parent for a direction of `Down`. The child's bonds are then iterated in order. Should an additional PPB be found, its direction must match that of the bond between parent and child. Otherwise an error must be reported. An error must also be reported if more than two PPBs are present. If a double bond is found, its target, a grandchild, is placed to the right of the parent. The bonds of the grandchild are then iterated in order. The target of the first PPB, a great-grandchild, is placed as before, above the grandchild if the direction is `Up`, or below if the direction is `Down`. If a second PPB is present, it must have a direction opposite the first. Otherwise an error must be reported.

The placement of parent, child, grandchild, and great-grandchild will yield one of two parities (Figure 12). For convenience and to avoid confusion, these parities are given the labels *syn* (vertical lines both lie above or below the double

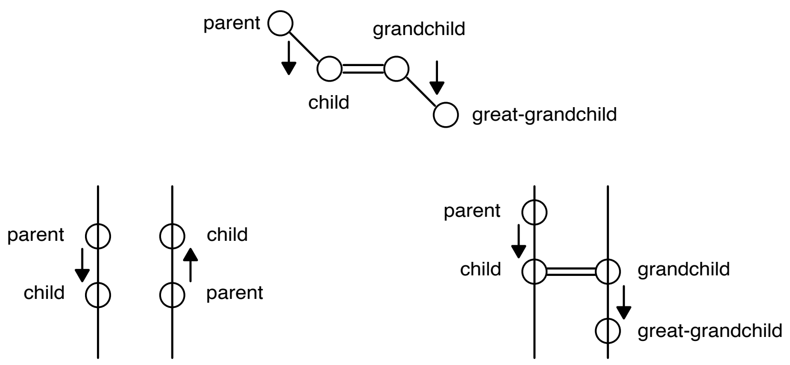


Figure 11: Partial parity bond (PPB). Double bond conformational parity is distributed over at least three bonds. A minimal assembly contains a double bond, two flanking PPBs (marked with upward- or downward-facing arrows), and associated nodes (top). To determine 2D orientation, parent and child are arranged on a vertical axis with parent above child for a bond direction of Down, and child above parent for a bond direction of Up (bottom-left). A complete diagram places parent, child, grandchild and great-grandchild (bottom right).

bond) and *anti* (vertical lines lie on opposite sides of the double bond).

To illustrate, consider the placement for (E)-2-butene. The parent atom, a methyl group, is placed on a vertical axis. Bonds are iterated and a PPB with a direction of `Down` is found. The child attached to the bond is therefore placed below the parent. Iterating the child's bonds reveals one bond of order two. The atom attached to it is placed to the right of the previous child. Bonds are again iterated, revealing a PPB whose direction is `Down`. The atom attached to this bond is then placed below its parent. The completed diagram reveals the *anti* conformation.

The same procedure works in reverse when encoding the `direction` attribute (Figure 13). 2-butene having the *syn* conformation is modeled using a diagram whose left- and right-hand sides point downward and upward, respectively. The parent atom, a primary carbon, is identified. It is bonded to one neighbor through a PPB. To match the diagram, a direction of `Down` is assigned. The child atom is attached to its own child through a double bond. Finally, the grandchild is attached to its own child through a PPB whose direction is consistent with the diagram. A direction of `Up` is therefore assigned.

Spreading conformational parity over more than one bond in this way means that several error states are possible (Figure 14):

- Overspecification. Two children are forced into the same direction along an axis.
- Underspecification. A required PPB is missing.

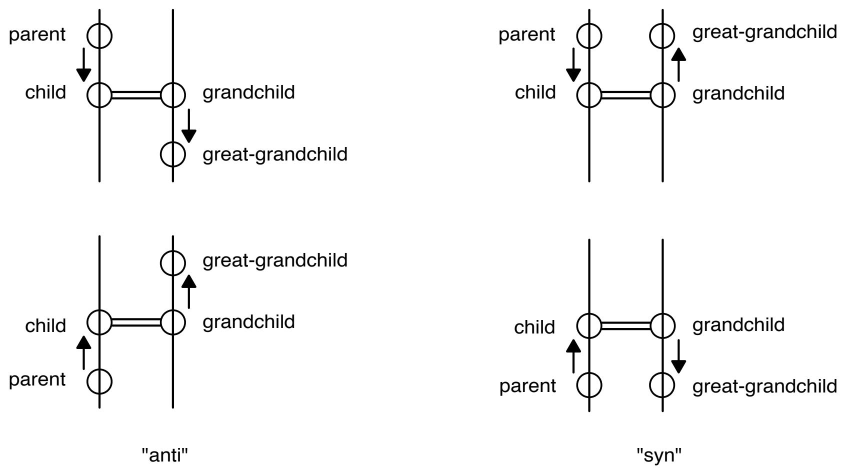


Figure 12: Partial Parity Bond Placement. Two unique modes of PPB placement are *syn* and *anti* (left and right, respectively). Within each mode is an equivalent pair of PPB pairs.

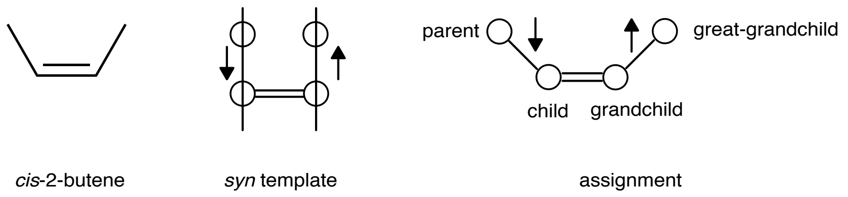


Figure 13: Assigning partial parity bonds to *cis*-2-butene.

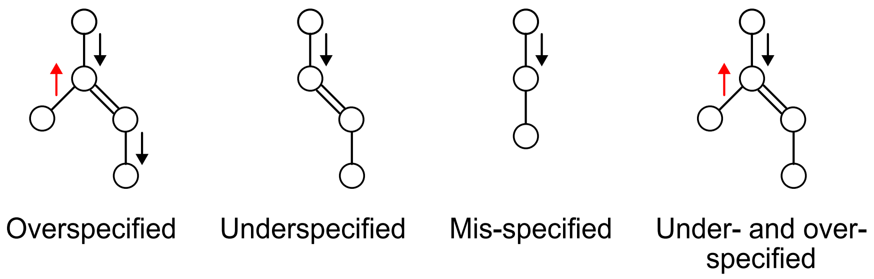


Figure 14: PPB error states. Because conformational descriptions are defined over three or more bonds, a variety of error states are possible.

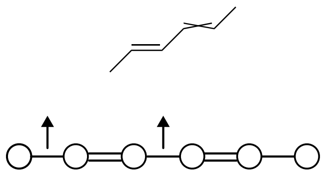


Figure 15: Error state exceptions. Conjugated dienes sharing a common PPB can lead to exceptional cases. The direction of the rightmost bond does not need to be set.

- Misspecification. A PPB is used between atoms, neither of which connects to a double bond.

An error must be reported should any of these states be encountered. Note that it is possible for a conformation to be both overspecified on one side of a double bond and underspecified on the other.

An exception to the underspecification rule applies in the case of conjugated polyenes (Figure 15). Here, a PPB may run from parent to child without one running from grandchild to great-grandchild. No error is reported in such a case and the conformation of the terminal double bond remains undefined.

A different kind of problem arises in the case of 2,4,6-octatriene, where the conformations of both outer double bonds are set but the conformation of the interior double bond is unknown (Figure 16) [41]. The use of PPBs along carbon-carbon bonds artifactually sets the inner double bond conformation. In this case the problem can be solved by reifying the hydrogens attached to the interior double bond, then directing PPBs along the resulting carbon-hydrogen bonds. This is, however, not a general solution. It will fail, for example, in the case of a nitrogen-nitrogen bond due to the lack of reifiable hydrogens.

The presence of a bridge node within a PPB complicates the assignment of direction (Figure 17). A bridge node has no children and therefore can never be a parent. At the same time, a bond always points from parent to child. This means that a bridged PPB yields a pair of bonds with opposite directions. The

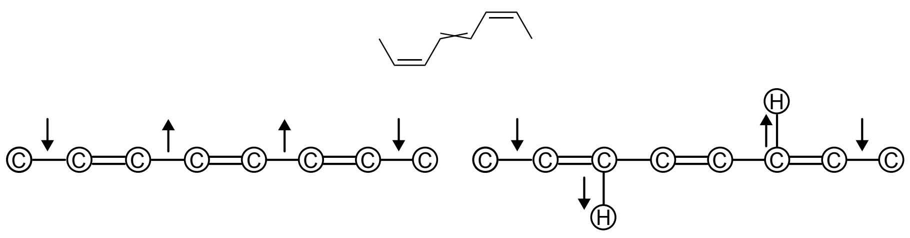


Figure 16: Artifactual bond conformation. Placing PPBs along carbon-carbon bonds sets the interior double bond's conformation artifactually to *anti* (left). This problem can be addressed by hydrogen reification followed by partial parity assignment to the newly-available carbon-hydrogen bonds (right).

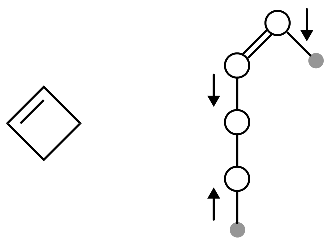


Figure 17: Bridged PPB Bond. A bridge across a PPB yields two bonds of opposite direction (right).

first (or "left") bond, has the direction that would result from a direct bond to an atomic child. The second (or "right") bond has the opposite direction.

Cyclic conjugated polyenes present special challenges to the use of PPBs, as exemplified by cyclooctatetraene (Figure 18). Although the (*syn*, *syn*, *syn*, *syn*) conformation is expressable using PPBs located along carbon-carbon axes only, the (*syn*, *syn*, *syn*, *anti*) conformation is not. This problem arises because the conformation for two different double bonds is set by the same PPB.

Configuration

Balsa limits configuration to the special case of an atom with exactly four substituents placed at the vertices of a tetrahedron. Here, "substituent" means an atomic neighbor or virtual hydrogen. Lone electron pairs are not substituents.

The configuration about an atom can be specified through its `parity` attribute. This attribute, together with a general convention, determine the relative three-dimensional positioning of substituents about a central atom (Figure 19). First, the parent of the central atom is identified. Then the frame of reference is shifted by sighting down the bond from the central atom's parent to the central atom. An equilateral triangle is then inscribed about the central atom. The bonds from the central atom are then iterated in order. If the parity

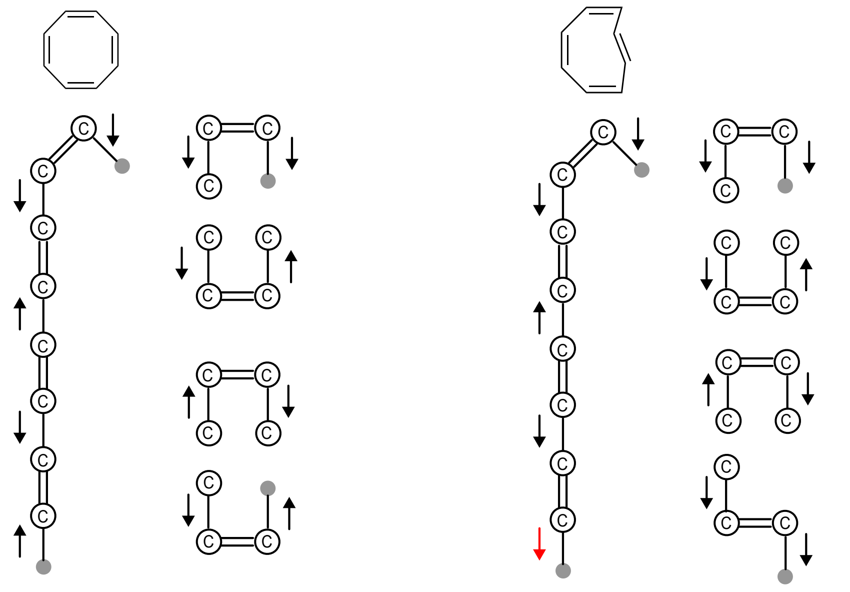


Figure 18: Cyclooctatetraene. The all-(*syn*) conformation is expressable (left), but the (*syn*,*syn*,*syn*,*anti*) conformation is not (right). Although the local conformations for the latter are all valid, the global conformation introduces a conflict, the direction marked in red.

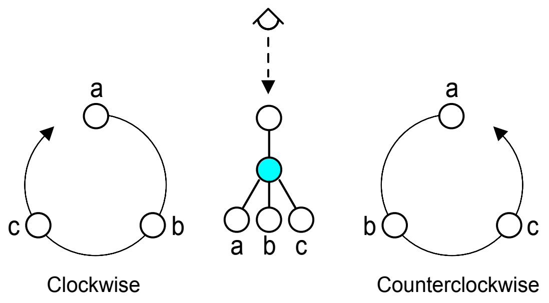


Figure 19: Atom parity. Four substituents surround a central atom with non-default atom parity (cyan). Sighting from the parent to the central atom, its children (a, b, and c) follow either a clockwise or counterclockwise pattern when iterated in order.

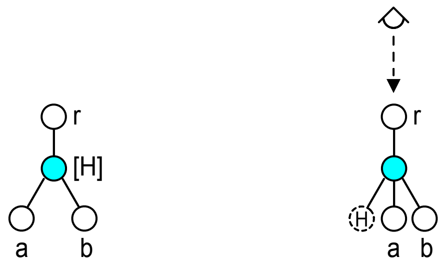


Figure 20: Atom parity with one virtual hydrogen. The central atom (cyan) has a virtual hydrogen (left). In this case, the virtual hydrogen assumes the role of root atom (right). The former root atom (r) then assumes the role of the first child, and analysis continues as before.

of the central atom equals `Clockwise`, children placed around the triangle in clockwise order. Otherwise the children are placed in counterclockwise order.

The convention is modified for atoms bearing one virtual hydrogen and three atomic substituents (Figure 20). The implied bond to the virtual hydrogen is considered to precede all other children.

One additional modification applies to a central atom without a parent. The frame of reference is shifted by sighting along the bond from the first child to the central atom. The remaining children are then placed around the triangle in either counterclockwise or clockwise order as before. If the central atom bears a virtual hydrogen, it is considered to be the first child.

A central atom whose parity attribute assumes a non-default value must have either four bonds if no virtual hydrogens are present, or three bonds if one virtual hydrogen is present. Some topologies might appear to be similar enough that the `parity` attribute can be used, but they all must be reported as errors (Figure 21). Examples include the central atom of allenes and other odd cumulenes, and the sulfur atom of sulfoxides where the lone pair is misconstrued

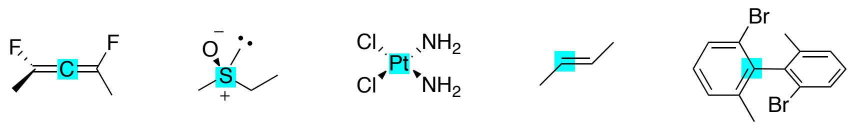


Figure 21: Invalid uses of atom parity. Only configuration about tetrasubstituted, tetrahedral atoms is supported. The marked atoms (cyan background) lack four substituents, tetrahedral geometry, or both. Assigning a non-default atom parity to any of them would be an error.

as a substituent.

Configurational descriptors may be assigned without regard to bond order. In other words, single, double, triple, and PPBs can all be present. The only requirement that must be satisfied by a central atom is an atomic neighbor count of four, or three when a virtual hydrogen is present.

It is sometimes useful to manipulate a configuration in a way that preserves the relative three-dimensional positioning of neighbor atoms. This process is called *transformation*. Five operations suffice to transform any configuration into any other (Figure 22):

- Virtualize. Replaces an atomic first hydrogen child with a virtual hydrogen.
- Reify. Replaces a virtual hydrogen with an atomic hydrogen first child.
- Swap Children. Exchanges any two children while simultaneously toggling the configurational descriptor.
- Slide Down. Sets the parent node as the first child. Disabled if virtual hydrogen is present.
- Slide Up. Sets the parent of a parentless node as the first child. Disabled if virtual hydrogen is present.

Not all tetracoordinate atoms will be *stereocenters*. A stereocenter is an atom whose "ligand permutation produces stereoisomers," as defined by Mislow and Siegel [42]. The presence of four bonds around a central atom is therefore a necessary, but insufficient condition to form a stereocenter. Molecular topology can result in permutation that does not yield stereoisomers. The assignment of non-default parities to such atoms should therefore be avoided.

Special handling is required for *undefined stereocenters*. A stereocenter is undefined if it lacks a parity. Undefined stereocenters indicate that no information about a central atom's configuration is known. Either a single configuration or a configurational mixture is present. This interpretation is consistent with the one used by Molfile format [33].

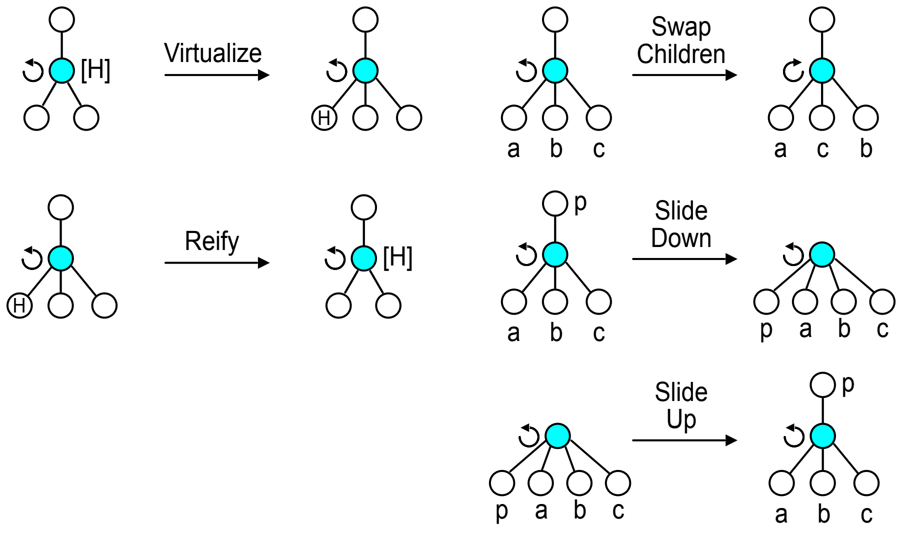


Figure 22: Configurational Transformations.

Syntax

A Balsa string is a sequence of zero or more characters belonging to the US-ASCII set (Figure 9). The internal structure of a string reflects a depth-first traversal of the corresponding molecular graph. As such, the syntax supports all of the features of a molecular tree, including branches, cycles, and disconnected components.

Grammar

Strings conform to an *LL(1) grammar* [43]. An LL(1) grammar is a context-free grammar whose strings can be parsed one character at a time from left to right with at most one character of lookahead. Additionally, LL(1) grammars expand the leftmost non-terminal first. These features make LL(1) grammars such as the one used by Balsa a good fit for manually-written recursive descent parsers. LL(1) grammars can also be used as a basis for auto-generated parsers through packages such as Bison [44] and ANTLR [45]. The full grammar for Balsa strings is available as a text file in this paper's Supporting Material.

Balsa's formal grammar is presented as a series of *production rules* ("productions"). A production defines a transformation allowed under the grammar. These transformations collectively define the set of valid Balsa strings. Productions can be used in the forward direction, when writing a string, or in the reverse direction, when reading a string.

A production is composed of two kinds of elements: *terminals* and *non-terminals*. A terminal is a character literal (e.g., "A"). A non-terminal is a reference to another production. This reference occurs through a name, which

Table 9: Balsa's US-ASCII Derived Character Set.

Code	Character	Code	Character	Code	Character
35	#	67	C	92	\
37	%	68	D	93	]
40	(	69	E	97	a
41	)	70	F	98	b
42	*	71	G	99	c
43	+	72	H	100	d
45	-	73	I	101	e
46	.	75	K	102	f
47	/	76	L	103	g
48	0	77	M	104	h
49	1	78	N	105	i
50	2	79	O	107	k
51	3	80	P	108	l
52	4	82	R	109	m
53	5	83	S	110	n
54	6	84	T	111	o
55	7	85	U	112	p
56	8	86	V	114	r
57	9	87	W	115	s
61	=	88	X	116	t
64	@	89	Y	117	u
65	A	90	Z	121	y
66	B	91	[		

appears to the left of a separator (`::=`) in a production. To the right of a separator appear the allowed terminals and non-terminals for the production.

Consider a hypothetical language composed of variable-length sequences of the lowercase letter “a”. Such a language could be cast as the following two productions:

```
text ::= a*
a    ::= "a"
```

The quantifier (`*`) indicates that a text in this language consists of a sequence of zero or more instances of the production `<a>`. This production in turn is defined as a lowercase letter (“a”). Therefore, valid texts in this language include the empty string, “a”, “aa”, and “aaaaaa” to name a few.

Atom

Atoms carry most of the information in a string. The `atom` non-terminal can be encoded using one of four productions.

```
atom ::= star | shortcut | selection | bracket
```

The first production, `star` (“star atom”), is comprised of one terminal value, the asterisk character (`*`). This “star atom” represents an atom in which every attribute assumes its default value.

```
star ::= "*"
```

The next atomic production, `shortcut`, is a non-terminal selected from the list: “B”; “C”; “N”; “O”; “P”; “S”; “F”; “Cl”; “Br”; and “I.” An atom encoded in this way (“shortcut atom”) receives the corresponding symbol as the value for the `element` attribute. All other attributes retain their default values, except `hydrogens`, which is set to `Implicit`.

```
shortcut ::= "B" "r"? | "C" "l"? | "N" | "O" | "P"
           | "S" | "F" | "I"
```

The third atomic production, `selection` (“selection”), is a non-terminal selected from the list: “b”; “c”; “n”; “o”; “p”; and “s.” An atom encoded in this way (“selected shortcut atom”) assigns the corresponding capitalized atom symbol to the `element` attribute, sets the `selected` attribute to `true`, and sets the `hydrogens` attribute to `Implicit`. All other atomic attributes retain their default values.

```
selection ::= "b" | "c" | "n" | "o" | "p" | "s"
```

The fourth and most complex atomic production is `bracket` (“bracket atom”). Bracket atom can set any atomic attribute. This atomic production must be used for any atom whose `parity`, `isotope`, or `charge` attributes assume non-default values, or whose `hydrogens` attributes equals anything other than `Implicit`. Attributes not set within the `bracket` production rule will leave the corresponding atomic values in their default states.

```
bracket ::= "[" isotope? symbol parity?
            virtual_hydrogen? charge? "]"
```

The value of a bracket atom's `isotope` attribute is determined by the optional `isotope` non-terminal. It consists of between one and three digits encoding the integers 1-999. Leading zeros (e.g., "007") are disallowed.

```
isotope ::= nonzero digit? digit?
```

The values of a bracket atom's `element` and `selected` attributes are determined by the `symbol` production ("symbol"). Three non-terminal variants are supported. These three variants influence the `element` and `selected` attributes in different ways. The `star` variant leaves both attributes as their default values. The `element` variant assigns the `element` attribute to the corresponding value while leaving the `selected` attribute as its default value. The `selection` non-terminal sets the `element` attribute to the corresponding element and the `selected` attribute to `true`. For example, the selection sequence "p" would assign the atomic `element` and `selected` attribute to `P` and `true`, respectively.

```
symbol ::= star | element | selection
```

Given the large number of variants within the `element` non-terminal, the following production only defines the first several. For a complete list of variants, see the full grammar in the Supporting Material.

```
element ::= "A" ( "c" | "g" | "l" | "m" | "r" | "s"
          | "t" | "u" )
          | "B" ( "a" | "e" | "h" | "i" | "k" | "r" )?
          /* etc. */
```

The `parity` attribute of a bracket atom is determined by the `parity` non-terminal. Allowed values are "@" and "@@," corresponding to the values `Counterclockwise` and `Clockwise`, respectively. A simple mnemonic correlates the counterclockwise swirl of the at sign with counterclockwise rotation.

```
parity ::= "@" "@"?
```

The `hydrogens` attribute of a bracket atom is controlled by the `virtual_hydrogen` non-terminal. It is comprised of the terminal "H" followed by an optional `nonzero` non-terminal. A digit appearing after the "H" terminal sets `hydrogens` to the corresponding virtual hydrogen count. If no digit is present, `hydrogens` is set to one. The virtual hydrogen count is set to zero by omitting the `virtual_hydrogen` non-terminal.

```
virtual_hydrogen ::= "H" nonzero?
```

The `charge` production sets the `charge` attribute of a bracket atom. This non-terminal begins with either the plus or minus terminals ("+" and "-" respectively) and ends with an optional `nonzero` non-terminal that sets the charge multiplicity. A plus terminal ("+") followed by anything other than a nonzero digit sets the `charge` attribute to one. Likewise, a minus terminal ("-") followed by any character other than a nonzero digit sets the `charge` attribute to minus one. Charge values of zero are obtained by omitting the `charge` production.

```
charge ::= ( "+" | "-" ) nonzero?
```

Values not defined within a bracket production leave the atomic attributes of the atom built from it in as their default values. For example, the bracket sequence "[C@H+]" lacks the `isotope` non-terminal so the value of the `isotope` attribute will remain `None`. Similarly, the bracket sequence "[13CH+]" lacks the `parity` non-terminal, so the corresponding `parity` attribute remains `None`.

Sequence

Children are added to an atom through the `sequence` non-terminal ("sequence"). A sequence starts with a required `atom` non-terminal. If an allowed non-terminal does not follow, the corresponding atom will have no children. Allowed non-terminals will be one of: `union`; `branch`; or `gap`.

```
sequence ::= atom ( union | branch | gap )*
```

The `union` non-terminal consists of an optional `bond` non-terminal followed by a mandatory non-terminal selected from `bridge` or `sequence`. If either of these latter non-terminals are detected but `bond` is not, the resulting bond is elided.

```
union ::= bond? ( bridge | sequence )
```

The `bond` non-terminal supports five variants ("-," "=," "#," "/," and "\"). The first three ("-," "=," and "#") set the `order` attribute of a bond to one, two, or three, respectively. The last two terminals ("/" and "\") set the `order` attribute to 1 while also setting the `direction` attribute to `Up` or `Down`, respectively.

```
bond ::= "-" | "=" | "#" | "/" | "\"
```

A sequence can contain a union, which in turn can contain a sequence. This is an example of recursion, albeit transitive. Although left-recursion is disallowed in LL(1) grammars, right recursion of the kind in `sequence` is allowed. Right recursion also occurs within the `union` and `branch` non-terminals.

The `bridge` non-terminal ("bridge") can take two forms. A nonzero non-terminal can be used, enabling single-digit bridge indexes in the range one through nine, inclusive. Double-digit bridge indexes, supporting the values ten through 99 inclusive, are available by prepending the percent character (`%`). For the latter variant, leading zeros are disallowed, meaning that the sequence "%07" must generate an error.

```
bridge ::= nonzero | "%" nonzero digit
```

As noted previously, the purpose of a bridge is to create a bond without creating a connection within the molecular tree (see: Molecular Tree). A bridge will most commonly be used to encode a ring-closure bond, but can be used anywhere within a string. The only requirement is that a bridge must be paired with another bridge having the same index. To prevent overflow, a bridge index may be reused.

An alternative to `union` within a sequence is the `branch` non-terminal ("branch"). Like union, branch joins a parent and child node. Wrapped by opening and closing parenthesis terminals ("(" and ")" respectively), branch encodes a sequence that may or may not be bonded to its parent. Bonding occurs if the `bond` non-terminal is included. Alternatively, the sequence will not be bonded if the `dot` non-terminal (".") appears. If neither `bond` nor `gap` are present, parent and child are connected through an elided bond.

```
branch ::= "(" ( dot | bond )? sequence ")"
```

The third option for adding atoms within a sequence is the `gap` non-terminal ("gap"). A gap consists of a `dot` production followed by a `sequence` production. The `gap` non-terminal serves the same purpose as it does within a branch: to enable connection within a molecular tree without creating a bonding relationship.

```
gap ::= dot sequence
```

Having defined sequence, it's now possible to define a string as an optional sequence. In other words, a Balsa string is either empty or contains a sequence. A string without a sequence encodes a molecular tree having zero nodes and zero edges.

```
String ::= sequence?
```

Implementation

Although Balsa is a simple language at its core, a number of factors can complicate implementation. Some points may be apparent from the description of syntax and semantics, but others may not be visible at such close range. As an aid to software quality, the following section addresses common issues that arise during the development of software to read and/or write Balsa strings.

Reading Strings

The goal of a Balsa reader is to transform a string input into a data structure output consistent with the string's content. The output data structure can take many forms. For example, a reader can merely validate a string by returning a boolean type. A more sophisticated reader can return a molecular graph capturing all atom and bond attributes and connectivity relationships.

Balsa strings are read one character at a time starting at the leftmost character and finishing at the rightmost character. The first character sets an initial

Table 10: Stack and hub for reading the string “C(N*)O”.

Step	Character	Action	Hub	Stack
1	“C”	create atom	C	
2	“(”	push hub to stack	C	C
3	“N”	create, connect atom	N	C
4	“*”	create, connect atom	*	C
5	“)”	pop atom, update hub	C	
6	“O”	create, connect atom	O	

reader state, and each subsequent character causes a state transition. The cumulative application of these state transitions yields the data structure to be returned.

Readers that capture atom-atom connectivity will typically maintain a reference to a *hub*. A hub is an atom to which the next node will be connected. On reading the first complete `atom` non-terminal, the corresponding atom is constructed and set as the hub. Subsequently processing a complete `union`, `branch`, or `gap` non-terminal triggers three changes: (1) a child node is constructed; (2) the child is connected to the current hub, unless a `dot` non-terminal intervenes; and (3) the hub is replaced with the child node if it is an atom.

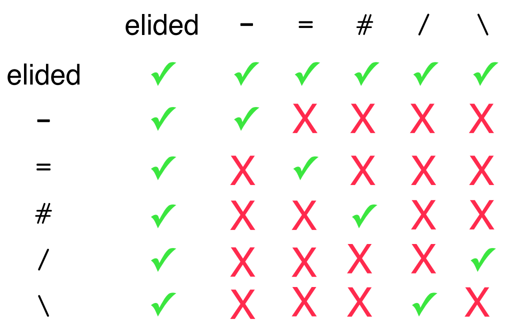
The presence of a branch adds some nuances over a union. The leading open parenthesis terminal (“(”) signifies that the current hub will later be re-exposed. This operation can be supported by a *stack* (Table 10). A stack is a data structure that allows items to be added individually (“pushed”) and removed (“popped”) in the reverse order of addition. At the start of a branch, the current hub is pushed to the stack. At the end of the branch, the stack is popped and its top value is assigned as the new hub.

Bridge bonds must be tested for *compatibility* (Figure 23), a task complicated by the presence of PPBs. A three-part test can be used:

1. An elided bond is compatible with any other bond.
2. A bond with a non-default direction is only compatible with an elided bond or a bond of opposite direction.
3. A bond with default direction is only compatible with an elided bond or a bond of identical order and default direction.

These tests can be implemented using a 100-element array. The first occurrence of a bridge causes the bridge bond’s attributes to be copied to the array at the bridge index. The second occurrence of a bridge causes the entry to be removed. The attributes stored in the entry are then compared with those of the current bridge bond. If the two sets of attributes are incompatible, an error is reported.

An elided bond is compatible with any other bond (Rule 1). Unless both bridge bonds are elided, however, the presence of compatible but unmatched



	elided	-	=	#	/	\
elided	✓	✓	✓	✓	✓	✓
-	✓	✓	X	X	X	X
=	✓	X	✓	X	X	X
#	✓	X	X	✓	X	X
/	✓	X	X	X	X	✓
\	✓	X	X	X	✓	X

Figure 23: Bridge bond compatibility. Two bonds with the types indicated along the row and column headings are either compatible (green check marks) or incompatible (red X).

bridge bonds could interfere with atom-centered computations. One solution is to set the attributes of the elided bridge bond explicitly. The array used for compatibility testing can be re-used to set the attributes of an unmatched bridge bond pair.

Readers must not assume that a dot (the period terminal, ".") implies the presence of disconnected components. This assumption is most likely to arise in the context of ad-hoc parsers using regular expressions, string matching, and the like. For example, the connected molecular graph for propane can be encoded using the string "C1C.C1".

A reader must assume that any input string can contain errors, and take appropriate steps to report them. The most useful errors will report a specific cause. Some will also report one or more cursor indexes. The most common mandatory errors are:

1. Invalid character (position). An unexpected character was encountered. A list of acceptable characters is helpful, but not required.

2. Unexpected end-of-line. Input ended unexpectedly.

3. Unbalanced bridge (position). A bridge with a given index appears an odd number of times.

4. Incompatible bridge bonds (position, position). The bonds to a pair of cuts are incompatible.

5. Delocalization subgraph lacks perfect matching. Before reporting this error, steps to remove unnecessarily selected atoms should be taken as described in the next section.

6. Partial parity bond not allowed (position). Neither terminal of a PPB possesses a double bond. Strings such as "C/C" and "C\C" contain isolated PPBs, which are invalid. A reader encountering such strings must report an error.

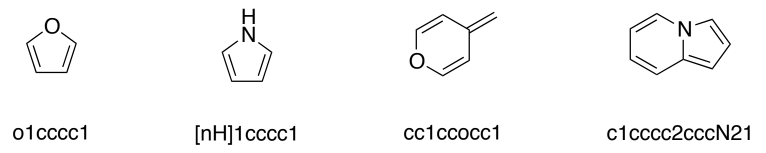


Figure 24: Gratuitous selection. Heteroatoms are selected needlessly because subvalence is zero in every case. Atoms selected gratuitously must be pruned before processing the DS.

A reader may also report optional errors, including:

1. Impossible isotope. A negative implied mass number results from the atom (e.g., "[2C]").
2. Impossible valence. The valence at an atom is impossibly high (e.g., "C(C)(C)(C)(C)C").
3. Impossible charge. An atom's charge gives it an apparent negative electron count (e.g., "[C+7]").

Pruning

As noted previously, a delocalization subgraph is invalid if it lacks a perfect matching. The one exception is when a selected atom and its associated bonds can be deleted from the delocalization subgraph through *pruning*. Pruning unsets the `selected` attribute of a selected atom, removing it and its edges from the delocalization subgraph, without corresponding promotion of any attached bonds.

An atom must be pruned if its subvalence equals zero. None of the bonds to such an atom can be promoted without altering the atom's `charge` attribute. Pruning the atom ensures the stability of its `charge` attribute, without interfering with bond promotion elsewhere. Viewed from another perspective, an atom with zero subvalence lacks unpaired electrons - at least within the narrow boundaries of the Balsa valence model. Such atoms can only form double bonds through changes to atomic charge.

An atom to be considered for pruning may have a non-zero charge. If so, subvalence is computed using the isoelectronic element's default valences. For example, a selected nitrogen atom with a charge of +1 would use the default valence for carbon. A selected phosphorous atom with a charge of -1 would use the default valences of sulfur. And so on. If no default valences are found in this way (e.g., "[c+2]"), a reader must report an error. Writers must not encode such atoms.

Pruning becomes necessary in cases of *gratuitous selection* (Figure 24). Gratuitous selection is the selection of an atom whose subvalence is zero and which therefore can never become part of a DS. This can happen when style, tradition, or convenience override symmetry concerns. Consider a string representing

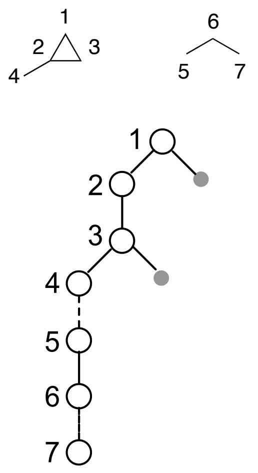


Figure 25: Depth-first traversal. Traversal of a disconnected, cyclic molecular graph (top) in depth-first order leads to a tree containing one bridge and one gap (bottom).

furan in which all atoms are selected (e.g., “c1ccco1”). Selecting any atom is unnecessary because furan does not exhibit DIME. This applies doubly to the the oxygen atom because it lacks an unpaired electron and so will never lead to DIME. It is nevertheless convenient to select the carbon atoms because all bonds can then be elided. The resulting representation (e.g., “c1cccO1”) leads to a delocalization subgraph with a perfect matching and hydrogen counts consistent with the original encoding.

Writers are encouraged, but not required, to avoid gratuitous atom selection. Readers, however, must always be prepared to prune.

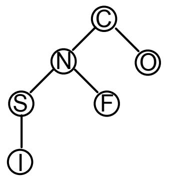
Writing Strings

Whereas a reader transforms a string into a data structure, the goal of a writer is the opposite: to transform a data structure into a valid string. The data structure will most often take the form of a molecular graph or tree, but other forms are possible. For practical reasons, the input data structure is likely to resemble the output from a reader. This enables strings to be encoded and decoded with minimal intermediate translation.

Regardless the input data structure’s form, it must be traversable in *depth-first* order (Figure 25). A depth-first traversal operates over a set of nodes associated in pairwise fashion by a set of connectivity relationships, which are typically edges. Traversal proceeds by successively replacing each node as the center of focus, or root. Each new root is chosen from the untraversed neighbors of the current root. Eventually all nodes are processed, ending the traversal.

A writer intercepts the depth-first traversal of an input data structure to

Table 11: Writing a string with the aid of a stack.



Step	Action	Stack
1	root C	"C"
2	open	"C","("
3	extend N	"C","(N"
4	open	"C","(N","("
5	extend S	"C","(N","(S"
6	extend I	"C","(N","(SI"
7	close	"C","(N(SI)"
8	extend F	"C","(N(SI)F"
9	close	"C(NSI)F)"
10	extend O	"C(NSI)F)O"

write an output string compatible with the formal grammar. There are no requirements around style. For example, it's equally valid to represent the carbon atom of methane using either implicit or virtual hydrogens (e.g., "C" or "[CH4]"). Single bonds may be elided or not, all other things being equal. Similarly, selection may or may not be used. Although an organization may seek to standardize certain styles of string output, a reader must process any string that is syntactically and semantically valid.

As described previously, the `branch` non-terminal encodes branches. A useful tool for encoding branches is a stack (Table 11). A writer begins by pushing the current branch onto the stack and extending it. When a new branch is encountered, it is pushed to the stack and extended. When the branch terminates, the current branch is popped and its contents are appended to the stack's new top item.

Writers should consider the non-negligible costs of atom selection. Algorithms for selection are likely to involve the perception of cycles, and so could exhibit superlinear time complexity. Often, a reader must perform a global deselection to arrive at a localized Lewis representation, which at the very least requires pruning and a maximal matching procedure. In other words, atom selection imposes two sets of costs: one on the writer and the other on every subsequent reader forever into the future.

Writers must ensure that all selected atoms can be deselected. Consider pyrrole, erroneously encoded with a selected nitrogen atom (e.g., "n1cccc1"). The nitrogen atom can not be pruned because of its non-zero subvalence (3 - 2). The DS therefore contains all five atoms and all five bonds. A perfect matching does not exist. A reader receiving such a string must report an error. To avoid this outcome, a writer can consume the subvalence by adding a virtual hydrogen (e.g., “[nH]1cccc1”). The subvalence of nitrogen in this case is zero, so the atom can be pruned. Doing so leaves a DS with four atoms, four edges, and a perfect matching. Readers will therefore consider the string valid. The larger question of gratuitous selection nevertheless remains.

Working with Molecular Graphs

Molecular trees offer some important benefits, but they are used less often than molecular graphs. Sooner or later, interconversions will be required. Readers will need to transform a tree into a graph, and writers will need to transform a graph into a tree. Although there are only three main differences between molecular trees and molecular graphs, these differences can lead to complex reconciliations during transformation. Understanding them is the key to lossless conversion.

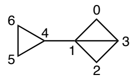
The first difference is the simplest to account for: atoms in a molecular tree have two kinds of edges: the one entering from an optional parent; and the zero or more exiting to children. A molecular graph, in contrast, maintains a single set of edges for an atom. Transformation of an atom in a tree to one in a graph must account for a possible inbound bond. The converse transformation must place the inbound bond.

The second difference is that molecular trees have gaps, but molecular graphs do not. Therefore, an edge may or may not represent a bonding relationship. Transformation from tree to graph must recognize the presence of gaps and place no corresponding bond. Gaps can occur at any outbound edge, even those within branches. Transformation from graph to tree must locate an appropriate pair of atoms to host the gap. A convenient pair would be the last atom of the first connected component and the first atom of the next connected component, but other choices are possible.

The third difference leads to the most complex reconciliation: molecular trees have bridges but molecular graphs do not. When transforming tree to graph, the two sides of a bridge bond must be identified, spliced, and added to the graph. When transforming graph to tree, a ring closure bond must be detected and split into a pair compatible bridge bonds. Compounding the problem in either sense of conversion is index reuse. Transformation from graph to tree will reuse gap indexes to avoid overflow. Transformation from tree to graph must account for the possibility that gap index reuse has occurred.

Transformation of graph to tree can be aided by a *pool* (Tabe 12). A pool issues a bridge index given an unordered pairing of atomic identifiers. The bridge index will not be re-issued until the pool receives the same pairing again. A pool is used during a depth-first traversal of a molecular graph, where the presence of a cycle is indicated by the discovery of an index that has already been traversed. On encountering a cycle, a bridge index is requested from the pool by submitting the source and target atomic indexes as an unordered pair. The bridge index is then used to construct the first half of the gap bond. Later,

Table 12: A pool manages bridge indexes during the transformation of a molecular graph.



Step	Edge	Cycle Found	Return Value	Action
1	(0,1)	No	None	
2	(1,2)	No	None	
3	(2,3)	No	None	
4	(3,0)	Yes	1	Reserve 1
5	(3,1)	Yes	2	Reserve 2
6	(1,3)	Yes	2	Release 1
7	(1,4)	No	None	
8	(4,5)	No	None	
9	(5,6)	No	None	
10	(6,4)	Yes	2	Reserve 2
11	(4,6)	Yes	2	Release 2
12	(0,3)	Yes	1	Release 1

the same bond will be traversed in the opposite direction. When it is, a gap index is requested from the pool by re-submitting a source and target index pairing. Doing so yields the bridge index, while simultaneously freeing it for later use.

Compatibility

For maximum compatibility with existing software, Balsa was designed as a *language subset* of SMILES. A language subset contains some of the syntax and semantics of its parent language, but adds none of its own. In principle this means that a feature present in Balsa should also be present in SMILES. Conversely, a feature present in SMILES may or may not also be present in Balsa.

Language subsets have ample precedent in computer science. One example is MISRA C [46], a subset of the C programming language that aims to eliminate "known undefined or otherwise dangerous behavior" [47]. Another example is the subset of JavaScript described by Crockford whose purpose is to "chip away at the features that are not beautiful until the language's true nature reveals itself." [48]. Language subsets arise for two main reasons. First, designing a language presents many opportunities to introduce errors small and large. Second, popular languages inevitably bring with them many users with diverse problems that demand new features. Some of these features cause future problems of their own. A language subset can improve both situations by eliminating the most problematic features.

Although similar considerations apply to SMILES, its case is different in one important way: SMILES was never publicly described with high precision. Overviews of the language have been published by both Weininger and Daylight. But the kind of technical documentation suitable for software development was never made public. Both the OpenSMILES and SMILES+ initiatives cite this shortfall as motivation.

Nevertheless, several SMILES software implementations have been released. Each one represents a unique set of decisions about how to reduce the published broad SMILES descriptions to a specification detailed enough for software development. Some implementations have themselves been reverse-engineered for clues. The Daylight implementation in particular is sometimes considered a de facto SMILES specification. Unfortunately, access to this implementation is restricted by commercial licensing, as is the case with several SMILES implementations. Moreover, reverse engineering is unlikely to address every question because of the vast search space defined by SMILES. Although some implementations are released under Open Source licenses, they are neither authoritative nor unanimous in their interpretations.

A publicly-available and precise subset of SMILES would offer a new way to address the problem. Such a subset should be interpretable by all existing software without modification. Moreover, most strings written by existing software would be interpretable by software based on the language subset. Because its description would be both public and precise, the language subset could serve as the basis for many implementations, with strong consensus around both syntax and semantics.

Unfortunately, the choice of the Balsa base language is fraught with difficulty on technical, historical, and cultural grounds. Simply put, the term "SMILES" has come to mean different things to different people. To some, SMILES is the language defined in public documentation. To others, SMILES is the Daylight implementation source code. Some may view SMILES as the collective definitions contained in both published documents and all software. To others, SMILES isn't even definable.

These problems can be avoided by considering as a base language not SMILES itself, but a *proto-language*. A proto-language is a hypothetical language from which a family of languages can be derived. A suitable SMILES proto-language would ideally contain all of the features consistent with the many ways in which the term "SMILES" is used today.

Accordingly, Balsa's base language is "ProtoSMILES," a SMILES proto-language. ProtoSMILES was identified through a process of elimination that started with every SMILES document ever published and every SMILES software implementation. The field was narrowed through two restrictions: (1) the source must be authoritative; and (2) the source must be publicly accessible. "Authoritative" in this context means that the source was created by either Weininger himself or Daylight, who alone can speak most authoritatively about what SMILES is. The requirement for public accessibility ensures that claims of compatibility can be tested. The following documents were identified as candidate ProtoSMILES sources:

- Weininger's report to the US Environmental Protection Agency ("The Report") [21].
- Weininger's original publication (herein "The Article") [1].
- Weininger's 2003 book chapter, revised in 2008 ("The Chapter") [22].
- An online manual maintained by Daylight ("The Manual") [23].
- The Daylight's toolkit, which implements a SMILES reader and writer ("The Toolkit") [7].

The Toolkit was eliminated for three reasons. First, its source code is not available, forcing an indirect reverse-engineering approach to language definition. Second, as a product of the company Weininger started, the Toolkit is likely to recapitulate the other sources. Third, as a commercial product without source code, the Toolkit may not always be available in the future.

The Report was eliminated mainly because the contents of this older document contradict at least two of the remaining sources (The Article and the Chapter). The Report contains no mention of bracket syntax, using markedly different notation instead. The Report also supports unusual characters including the exclamation mark (!). Unspecified "special states" for nitrogen-containing atoms are also supported by the Report.

The Manual was disregarded as a ProtoSMILES source because it merely restates content already present in The Paper and The Chapter. Removing this source simplifies the construction of ProtoSMILES without changing its definition. A secondary consideration was the possibility that updates to this web page could change the meaning of ProtoSMILES. However, web archives indicate that neither styling nor content of the Manual have changed in at least ten years.

Analysis of The Article and The Chapter revealed the former to contain all of the material in the latter. Moreover, The Chapter contains information not present in The Article. Therefore, the Paper was eliminated as a basis for ProtoSMILES.

This process of elimination left just The Chapter as the basis for ProtoSMILES. The remainder of this section compares and contrasts ProtoSMILES with Balsa. The main difficulty with this approach is that the work defining ProtoSMILES occasionally contains contradictions. These will be noted when relevant.

ProtoSMILES defines a fixed set of chemical element symbols that even at the time of publication was partially obsolete [22, p. 83]. The symbol for element 105 is given as Ha (Hahnium), despite IUPAC's resolution of the naming controversy of the element in favor of the symbol Db (Dubnium) years prior [49]. Balsa therefore disallows the symbol Ha. To maintain compatibility with ProtoSMILES and avoid future changes to Balsa's grammar, elements having atomic number greater than 105 are also excluded.

ProtoSMILES defines a recursive grammar for branching which, perhaps without the knowledge of the author, allows sequentially nested parentheses [22, p. 86]. Balsa disallows such constructs (e.g., "*((*))*" and "*(((*)))*").

ProtoSMILES supports reactions but Balsa does not. Specifically, the greater than symbol (>) is not a valid Balsa character. This restriction precludes strings such as "*>>*". Furthermore, Balsa lacks the "map" attribute, used for atom-atom mapping in reactions, and its accompanying syntax.

Balsa only supports two stereodescriptors, encoded with the terminals "@" and "@@." ProtoSMILES provides a recursive grammar for stereodescriptors that allows multi-symbol descriptors such as "@@@" and "@@@@" [22, p. 94]. ProtoSMILES also supports non-tetrahedral descriptors including "@AL1," "@AL2," "@1," and "@SP1." None of these are supported by Balsa. Nor does Balsa support the application of tetracoordinate stereodescriptors to odd cumulenes. These other forms of atomic configuration are relatively rare. Their inclusion in Balsa would add substantial cost for little gain.

ProtoSMILES supports "aromatic bonds" (":"), but Balsa does not. Despite clear inclusion within the syntax of ProtoSMILES, no precise guidance on using aromatic bonds is provided. Weininger notes that "Adjacent atoms without an intervening bond symbol are connected by a valence-dictated bond (typically a single or aromatic bond) '-' (single) and ':' (aromatic) bond symbols may always be omitted on input" [22, p. 85]. This passage implies that aromatic bonds serve no purpose. Accordingly, Balsa does not allow them and the syntax does not include the colon symbol (`:`).

ProtoSMILES sets no limits on the atoms that can be marked as selectable (i.e., "aromatic"), but Balsa does. As described previously, pruning the DS requires a method to determine whether or nor an atom is subvalent (see: Valence and Subvalence). Making this determination requires at least one default valence. The only reasonable way to allow selection of all atoms regardless of associated element would be to provide default valences for them. As a result, only those atoms associated with a subset of elements may be selected ("B", "C", "N", "O", "P", "S"). This list is a subset of the elements whose default valences are defined. Halogens were excluded because they rarely appear in rings. This restriction only marginally limits the capabilities of Balsa because atoms other than those supported are rarely involved in DIME.

It's unclear whether ProtoSMILES allows a virtual hydrogen count on hydrogen itself. Weininger notes that "explicit hydrogen specification is required" in the case of "hydrogens connected to other hydrogens, e.g. [H][H], molecular hydrogen" [22, p. 97]. This may reflect an erroneous belief that parsers would be incapable of processing the implied strings (e.g., "[HH]"). To be clear, Balsa supports virtual hydrogen counts on all atoms.

ProtoSMILES ascribes ambiguous meaning to gaps. As noted by Weininger, "... In terms of the valence model being represented, the dot literally represents a bond of formal order zero: the atoms on either side of the dot are explicitly not bonded to each other." [22, p. 88]. Other statements appear to contradict this statement. Balsa explicitly disallows the zero bond order interpretation.

ProtoSMILES assigns no explicit upper or lower bounds to numerical atomic attributes. These boundaries are nevertheless crucial for implementors, who must determine the data types necessary to efficiently prevent underflow and overflow. For this reason, Balsa sets both upper and lower bounds, respectively, on the following atomic attributes: isotope ($0 <$ value <1000); charge (-10, $<$value <10); and hydrogens, $0 \leq$ value <10). Although it might be argued that lower bounds on physical quantities such as isotope and hydrogen count should be implicit, the lack of precision forces implementors to improvise, which can lead to divergent behavior. ProtoSMILES places no upper bound on the charge notation (e.g., "++" and "--"): "…'+' meaning +2 …" [22, p. 94]. Due to their redundancy, Balsa disallows duplicated plus and minus characters (+ and −, respectively) within bracket atoms.

Balsa provides a few semantic clarifications not fully addressed in ProtoSMILES. A detailed algorithm for determining implicit hydrogen count is provided, together with the required valence table. Unlike ProtoSMILES, Balsa explicitly considers the case of computing implicit hydrogen counts on selected atoms. Balsa also provides detailed algorithms, absent in SMILES, for selection and deselection. These are based in graph theory rather than the ambiguous and overloaded chemical concept of "aromaticity" used in ProtoSMILES. Uniquely, Balsa introduces the concepts of "pruning" and "gratuitous selection." ProtoSMILES does not specify those bonds that can be promoted during deselection, but Balsa does. ProtoSMILES does not restrict partial parity bonds to those atoms also possessing at least one double bond, whereas Balsa does. Finally, Balsa syntax is based on a formal grammar and tooling rooted in decades of computer science research, whereas ProtoSMILES syntax is based for the most part on imprecise natural language descriptions and only a partial formal grammar.

Compatibility between Balsa and ProtoSMILES is summarized in Table 13. Barring elements of ProtoSMILES that are either underspecified or internally inconsistent (entries marked "maybe"), Balsa can be seen to be a subset of ProtoSMILES. Output from Balsa writers should therefore be readable by SMILES readers. However, output from SMILES writers may or may not be readable by Balsa readers. Balsa can therefore be characterized as forward compatible with ProtoSMILES [50].

Discussion

Balsa's main advantage as a molecular serialization format is high information density. The most common atom types can be represented with just one character. The worst case atom, using every available atomic attribute, requires 14 characters. The most common bond types can be encoded implicitly. In the worst case one character per bond is required.

These metrics compare favorably with alternatives. Consider the Molfile (V2000) format. 32 characters per atom are required in the best case, and 51 characters in the worst. All bonds, regardless of type, require between nine and 12 characters. Atomic charges and isotopes typically require additional

Table 13: Some differences between ProtoSMILES and Balsa.

Feature	ProtoSMILES[a]	Balsa
element symbol Ha	yes	no
multiple branching (e.g., "*((*))*")	yes	no
reactions using greater than symbol (>)	yes	no
atomic "map" attribute	yes	no
extended stereodescriptors (e.g., "@OH1")	yes	no
use of stereodescriptors on odd cumulene centers	yes	no
virtual hydrogen count on hydrogen (e.g., "[HH]")	maybe	yes
gaps are bonds of "formal order zero"	maybe	no
unbounded numerical atomic properties	yes	no
atoms associated with any element are selectable	yes	no
non-elided bonds may be promoted during selection	maybe	no
isolated partial parity bond	yes	no
repeated charge notation (e.g., "++" and "--")	yes	no
aromatic bond (":")	yes	no

[a]Values of "maybe" indicate underspecified and/or internally inconsistent features.

characters.

The information density of Balsa strings makes them attractive for several applications. In memory-constrained environments such as those found on handheld devices and networks, many more Balsa strings can be present than alternative encodings. In-memory structure search over large collections becomes feasible. Because a string often fits within one line on a terminal, Balsa can be used for data entry in manual interactive shells such as a real-eval-print loops (REPLs) or notebooks.

Another advantage is lossless interconversion with other serialization formats. The line notation InChI might appear to compete with Balsa in this sense. The authors of InChI have noted that InChI is not a solution to the molecular serialization problem but rather an identifier [51]. The reason presumably lies with the fact that InChI uses concepts such as hydrogen delocalization that are uncommon elsewhere. To date, no third-party reader or writer of InChI has been reported.

Despite its information density, Balsa can faithfully encode and decode most of what chemists would consider "organic molecules." This is evidenced by the widespread presence of Balsa strings within large, public-facing databases such as PubChem,[9] ChEMBL,[12] ChEBI,[10] and others.

Balsa's compact representation requires some tradeoffs around expressiveness. Multi-atom bonding is not supported beyond what's available through the delocalization subgraph. As a result, many bonding arrangements such as those found in organometallics and delocalization-stabilized ions can not be encoded. Zero-order bonds are not supported. Only four-coordinate, tetrahedral stereocenters can be encoded, which precludes many forms of chirality such as helical chirality, all forms of non-tetrahedral stereochemistry, and lone-pair tetrahedral centers. Conformational restrictions other than those found in isolated double bonds are not supported, precluding important molecules exhibiting atropisomerism for example. Formats such as CDXML and molfile support enhanced stereochemical features enabling the differentiation of various kinds of partial stereochemical information, but Balsa does not.

Broader expansion of Balsa's capabilities could be possible through *metaformats*. A metaformat embeds one or more Balsa strings within a surrounding serialization format. For example, two- or three-dimensional coordinates could be associated with each atom through a metaformat that includes a dictionary mapping implicit atomic index to coordinate. An implicit atomic index could in turn be based on node order. Collections of atoms or bonds could likewise be encoded to replicate the enhanced stereochemistry features of other formats. And so on. However, the utility of such extensions should be weighed against Balsa's main value proposition: high information density. An application attempting to use a verbose metaformat may benefit from adopting a better-suited format instead.

Setting aside the many technical and usability issues a metaformat would raise, versioning is likely to play an important role. Balsa itself lacks any mechanism to convey the concept of version. This stands in contrast to InChI, which not only encodes a version identifier, but has done so from its first release.

Adding a version identifier would, unfortunately, break compatibility with the large number of existing SMILES software. Metaformats offer an opportunity to address this limitation.

The most noteworthy feature of Balsa is its compatibility with SMILES. Contemporary SMILES implementations will be able to read and write Balsa strings — at least to a point. Incompatibilities will arise from three main sources: (1) those features of SMILES that Balsa has deliberately omitted; (2) those aspects of the SMILES language that are incompletely-specified, ambiguous, or self-contradictory; and (3) those features of SMILES that were deliberately disregarded by implementations.

The availability of a minimal yet highly functional, fully-specified core language offers many opportunities to improve data quality. One of the most important will be an open reference implementation, the design and source code of which are underway. A freely-available reference implementation in turn makes automated validation suites possible. These suites can improve data quality by reporting syntax and semantic differences among implementations, preferably before release. A reference implementation taken together with the guidelines for readers and writers in this paper should make it possible to write software that conforms to a very high level of precision, regardless of programming language or paradigm. Given verified implementations, performance optimizations can be considered. The existence of a core language specification should also aid standardization efforts, either for Balsa itself, or SMILES. Finally, the development of better line notations is only possible given a thorough understanding of the scope and limitations of existing options. Here, metaformats could offer a bridge from the present to the future.

Conclusion

This paper describes Balsa, a compact molecular serialization format designed to be forward compatible with SMILES. Constitutionally, Balsa can encode molecules conforming to the electron-pair bonding model. In the event of undesired symmetry artifacts due to delocalization, Balsa offers a mitigation based in graph theory. Conformational isomerism of alkenes is supported by partial parity bonds. The configurations of tetracoordinate, stereogenic atoms are encoded through the use of a parity enumeration and conventions around its use.

Balsa's syntax is described in detail through a formal grammar. This method concisely summarizes the complete set of strings that could be considered syntactically valid Balsa representations. The formal grammar was created for either direct use with an automated parser generator or as a blueprint for a hand-written recursive-descent parser.

The complete set of rules needed to interpret Balsa's syntax are described in detail. This is a crucial component of the language's definition because Balsa achieves its information density in part by favoring convention over encoding. As an aid to difficult cases, guidelines for readers and writers are included.

As a language subset, Balsa can be used with a wide range of SMILES software today. In this sense, Balsa may seem to offer nothing new or even of value. However, Balsa has been defined at a level of precision that SMILES never was. This difference makes it possible to use Balsa in unique ways. Open reference implementations and validation suites can now be developed and deployed. Families of extensions can be built, each one based on the same unambiguous foundation. Formal standardization becomes more feasible given detailed reference material on which to draw. Finally, it is only through the clear demarcation of boundaries that the frontier becomes visible.

Acknowledgement

I thank Andrew Dalke and John Mayfield for helpful conversations during the preparation of this manuscript.

References

[1] David Weininger. "SMILES, a Chemical Language and Information System. 1. Introduction to Methodology and Encoding Rules." In: *J. Chem. Inf. Model.* 28.1 (Feb. 1, 1988), pp. 31–36. ISSN: 1549-9596. DOI: 10.1021/ci00057a005.

[2] William J. Wiswesser. "107 Years of Line-Formula Notations (1861-1968)." In: *J. Chem. Doc.* 8.3 (Aug. 1968), pp. 146–150. ISSN: 0021-9576, 1541-5732. DOI: 10.1021/c160030a007.

[3] Noel M O'Boyle et al. "Open Babel: An Open Chemical Toolbox." In: *J. Cheminform.* 3.1 (Dec. 2011), p. 33. ISSN: 1758-2946. DOI: 10.1186/1758-2946-3-33.

[4] *RDKit.* URL: https://www.rdkit.org/ (visited on 03/17/2022).

[5] Christoph Steinbeck et al. "The Chemistry Development Kit (CDK): An Open-Source Java Library for Chemo- and Bioinformatics." In: *J. Chem. Inf. Comput. Sci.* 43.2 (Mar. 1, 2003), pp. 493–500. ISSN: 0095-2338. DOI: 10.1021/ci025584y.

[6] *JChem Base — Chemaxon Docs.* URL: https://docs.chemaxon.com/display/docs/jchem-base.md (visited on 03/17/2022).

[7] *Daylight Toolkit.* URL: https://www.daylight.com/products/toolkit.html (visited on 03/17/2022).

[8] *OEChem TK — OEChem Toolkit — Cheminformatics.* URL: https://www.eyesopen.com/oechem-tk (visited on 03/17/2022).

[9] Sunghwan Kim et al. "PubChem Substance and Compound Databases." In: *Nucleic Acids Res* 44.D1 (Jan. 4, 2016), pp. D1202–D1213. ISSN: 0305-1048, 1362-4962. DOI: 10.1093/nar/gkv951.

[10] K. Degtyarenko et al. "ChEBI: A Database and Ontology for Chemical Entities of Biological Interest." In: *Nucleic Acids Research* 36 (Database Dec. 23, 2007), pp. D344–D350. ISSN: 0305-1048, 1362-4962. DOI: 10.1093/nar/gkm791.

[11] John J. Irwin and Brian K. Shoichet. "ZINC - A Free Database of Commercially Available Compounds for Virtual Screening." In: *J. Chem. Inf. Model.* 45.1 (Jan. 1, 2005), pp. 177–182. ISSN: 1549-9596, 1549-960X. DOI: 10.1021/ci049714+.

[12] A. Gaulton et al. "ChEMBL: A Large-Scale Bioactivity Database for Drug Discovery." In: *Nucleic Acids Research* 40.D1 (Jan. 1, 2012), pp. D1100–D1107. ISSN: 0305-1048, 1362-4962. DOI: 10.1093/nar/gkr777.

[13] *Talk:CAS Registry Number - Wikipedia.* URL: https://en.wikipedia.org/wiki/Talk:CAS_Registry_Number (visited on 03/17/2022).

[14] Tiago Sousa et al. "Generative Deep Learning for Targeted Compound Design." In: *J. Chem. Inf. Model.* 61.11 (Nov. 22, 2021), pp. 5343–5361. ISSN: 1549-9596, 1549-960X. DOI: 10.1021/acs.jcim.0c01496.

[15] David Weininger, Arthur Weininger, and Joseph L. Weininger. “SMILES. 2. Algorithm for Generation of Unique SMILES Notation.” In: *J. Chem. Inf. Model.* 29.2 (May 1, 1989), pp. 97–101. ISSN: 1549-9596. DOI: 10.1021/ci00062a008.

[16] Noel M O’Boyle. “Towards a Universal SMILES Representation - A Standard Method to Generate Canonical SMILES Based on the InChI.” In: *J. Cheminform.* 4.1 (2012), p. 22. ISSN: 1758-2946. DOI: 10.1186/1758-2946-4-22.

[17] Robert M. Hanson. “Jmol SMILES and Jmol SMARTS: Specifications and Applications.” In: *J Cheminform* 8.1 (Dec. 2016), p. 50. ISSN: 1758-2946. DOI: 10.1186/s13321-016-0160-4.

[18] Axel Drefahl. “CurlySMILES: A Chemical Language to Customize and Annotate Encodings of Molecular and Nanodevice Structures.” In: *J Cheminform* 3.1 (Dec. 2011), p. 1. ISSN: 1758-2946. DOI: 10.1186/1758-2946-3-1.

[19] Tzyy-Shyang Lin et al. “BigSMILES: A Structurally-Based Line Notation for Describing Macromolecules.” In: *ACS Cent. Sci.* 5.9 (Sept. 25, 2019), pp. 1523–1531. ISSN: 2374-7943, 2374-7951. DOI: 10.1021/acscentsci.9b00476.

[20] *Chemaxon Extended SMILES and SMARTS - CXSMILES and CXSMARTS — Chemaxon Docs.* URL: https://docs.chemaxon.com/display/docs/chemaxon-extended-smiles-and-smarts-cxsmiles-and-cxsmarts.md (visited on 03/17/2022).

[21] Eric Anderson, Gilman D. Veith, and David Weininger. *SMILES: A Line Notation and Computerized Interpreter for Chemical Structures.* EPA/600/M-87/021. US Environmental Protection Agency, Aug. 1987. URL: https://nepis.epa.gov/Exe/ZyPURL.cgi?Dockey=2000CAUR.TXT.

[22] David Weininger. “SMILES-A Language for Molecules and Reactions.” In: *Handbook of Chemoinformatics.* Ed. by Johann Gasteiger. Weinheim, Germany: Wiley-VCH Verlag GmbH, 2003, pp. 80–102. ISBN: 978-3-527-61827-9 978-3-527-30680-0. DOI: 10.1002/9783527618279.ch5.

[23] *Daylight Theory Manual.* URL: https://www.daylight.com/dayhtml/doc/theory/ (visited on 03/17/2022).

[24] Johann Gasteiger et al. “Leaving Us with Fond Memories, Smiles, SMILES and, Alas, Tears: A Tribute to David Weininger, 1952–2016.” In: *J Comput Aided Mol Des* 32.2 (Feb. 2018), pp. 313–319. ISSN: 0920-654X, 1573-4951. DOI: 10.1007/s10822-018-0104-3.

[25] *OpenSMILES Home Page.* URL: http://opensmiles.org/ (visited on 03/17/2022).

[26] Anton den Joed. “Parsing and Conversion of SMILES-strings to Molecular Graphs.” Universiteit Leiden: Leiden University, 2010. URL: https://theses.liacs.nl/203.

[27] *IUPAC SMILES+ Specification.* URL: https://iupac.org/projects/project-details/?project_nr=2019-002-2-024 (visited on 03/17/2022).

[28] *IUPAC SMILES+.* URL: https://github.com/IUPAC/IUPAC_SMILES_plus (visited on 03/17/2022).

[29] Mark D. Wilkinson et al. "The FAIR Guiding Principles for Scientific Data Management and Stewardship." In: *Sci Data* 3.1 (Dec. 2016), p. 160018. ISSN: 2052-4463. DOI: 10.1038/sdata.2016.18.

[30] *Depict Archive Page.* URL: https://web.archive.org/web/20070113052758/https://www.daylight.com/daycgi/depict (visited on 03/17/2022).

[31] Peter Murray-Rust and Henry S Rzepa. "CML: Evolution and Design." In: *J Cheminform* 3.1 (Dec. 2011), p. 44. ISSN: 1758-2946. DOI: 10.1186/1758-2946-3-44.

[32] *CDX Format Specification: The CDXML Text-Based File Format.* URL: https://www.cambridgesoft.com/services/documentation/sdk/chemdraw/cdx/IntroCDXML.htm (visited on 03/17/2022).

[33] *CTfile Formats - Dassault Systèmes.* URL: https://discover.3ds.com/ctfile-documentation-request-form (visited on 03/17/2022).

[34] Alexandru T. Balaban. "Applications of Graph Theory in Chemistry." In: *J. Chem. Inf. Comput. Sci.* 25.3 (Aug. 1, 1985), pp. 334–343. ISSN: 0095-2338. DOI: 10.1021/ci00047a033.

[35] Alex M. Clark. "Accurate Specification of Molecular Structures: The Case for Zero-Order Bonds and Explicit Hydrogen Counting." In: *J. Chem. Inf. Model.* 51.12 (Dec. 27, 2011), pp. 3149–3157. ISSN: 1549-9596, 1549-960X. DOI: 10.1021/ci200488k.

[36] *Periodic Table of Elements - IUPAC — International Union of Pure and Applied Chemistry.* URL: https://iupac.org/what-we-do/periodic-table-of-elements/ (visited on 03/18/2022).

[37] Gilbert Lewis. "The Atom and the Molecule." In: *J. Am. Chem. Soc.* 38 (1916), p. 672. DOI: 10.1021/ja02261a002.

[38] Milan Randić. "Aromaticity Revisited." In: *Advances in Quantum Chemistry.* Vol. 77. Elsevier, 2018, pp. 167–199. ISBN: 978-0-12-813710-9. DOI: 10.1016/bs.aiq.2017.09.001.

[39] Robert C. Kerber. "If It's Resonance, What Is Resonating?" In: *J. Chem. Educ.* 83.2 (Feb. 2006), p. 223. ISSN: 0021-9584, 1938-1328. DOI: 10.1021/ed083p223.

[40] Jack Edmonds. "Paths, Trees, and Flowers." In: *Can. J. Math.* 17 (1965), pp. 449–467. DOI: 10.4153/CJM-1965-045-4.

[41] John Mayfield. *CXSMILES Gotchas – Part 1: Bond Indexes – NextMove Software*. NextMove Software Blog. URL: https://nextmovesoftware.com/blog/2022/04/08/cxsmiles-gotchas-part-1-bond-indexes/ (visited on 09/11/2022).

[42] Kurt Mislow and Jay Siegel. "Stereoisomerism and Local Chirality." In: *J. Am. Chem. Soc.* 106.11 (May 1984), pp. 3319–3328. ISSN: 0002-7863, 1520-5126. DOI: 10.1021/ja00323a043.

[43] Douglas Thain. *Introduction to Compilers and Language Design*. Second Edition. University of Notre Dame, 2020. ISBN: 9798655180260. URL: https://www3.nd.edu/~dthain/compilerbook/.

[44] *Bison, The YACC-compatible Parser Generator*. URL: http://dinosaur.compilertools.net/bison/index.html (visited on 08/21/2022).

[45] Terence Parr, Sam Harwell, and Kathleen Fisher. "Adaptive LL(*) Parsing: The Power of Dynamic Analysis." In: *ACM SIGPLAN Not.* 49.10 (Oct. 15, 2014), pp. 579–598. DOI: 10.1145/2714064.2660202.

[46] *MISRA Website*. URL: https://www.misra.org.uk/ (visited on 03/18/2022).

[47] Les Hatton. "Language Subsetting in an Industrial Context: A Comparison of MISRA C 1998 and MISRA C 2004." In: *Information and Software Technology* 49.5 (May 2007), pp. 475–482. ISSN: 09505849. DOI: 10.1016/j.infsof.2006.07.004.

[48] Douglas Crockford. *JavaScript: The Good Parts ; [Unearthing the Excellence in JavaScript]*. 1. ed. Unearthing the Excellence in JavaScript. Beijing Köln: O'Reilly, 2008. 153 pp. ISBN: 978-0-596-51774-8.

[49] A. M. Sageson. "Names and Symbols of Transfermium Elements (IUPAC Recommendations 1997)." In: *Pure Appl. Chem.* 69.12 (Jan. 1, 1997), pp. 2471–2474. ISSN: 1365-3075, 0033-4545. DOI: 10.1351/pac199769122471.

[50] *[Editorial Draft] Extending and Versioning Languages: Terminology*. URL: https://www.w3.org/2001/tag/doc/versioning (visited on 10/02/2022).

[51] Stephen R Heller et al. "InChI, the IUPAC International Chemical Identifier." In: *J Cheminform* 7.1 (Dec. 2015), p. 23. ISSN: 1758-2946. DOI: 10.1186/s13321-015-0068-4.