Chemlogic: A Logic Programming Computer Chemistry System

Nicholas Paun
<np@icebergsystems.ca>
Gr. 12, Mount Sentinel Secondary School

April 10, 2015

Abstract

Chemlogic is a logic program for performing stoichiometric calculations, converting chemical quantities between units, balancing and completing chemical equations and converting chemical formulas to and from chemical names. These features are implemented using a database of chemical information, a set of grammars, and a linear equation solver. Chemlogic can detect and provide guidance for resolving syntax and other errors. It is available as an Android App (designed as a study tool for chemistry students), a Web interface and a command-line domain-specific language.

Contents

List of Figures

1 Curricular Connections

1.1 Science 9

- Names and formulas of ionic compounds.
  (Including support for multivalent metals and polyatomic groups.)

1.2 Science 10

- Names and formulas of binary covalent compounds.
- Names and formulas of acids.
- Naming conventions for some simple organic compounds.
  (Including support for alkanes and alcohols.)
- Balancing of chemical equations in word or symbolic form.
- Identification of reaction types.
  (Including support for neutralization, double replacement, single replacement, synthesis, decomposition and combustion reactions.)
- Determining whether a reaction will take place using the reactivity series.
- Completion of common chemical reactions.
  (Including support for neutralization, double replacement and single replacement reactions).

1.3 Chemistry 11

- Names and formulas of alkenes and ionic compound hydrates.
- Unit conversion for chemical quantities.
(Including support for mass, moles, volume of gas, volume of solution and concentration).
- Stoichiometric calculations.
- Limiting reactant analysis and calculations involving the stoichiometry of excess quantities.

Not yet implemented:

- Structural formulas and naming conventions for organic chemistry; structures of polyelectronic atoms

2 Design and Algorithms

Chemlogic implements two major features: Balancing symbolic or word equations and converting chemical names to formulas and vice versa. These operations are done using two interesting algorithms, which will be discussed below.

2.1 Balancing Chemical Equations with Systems of Linear Equations

2.1.1 Balancing by Inspection

Chemical equations are most commonly balanced **by inspection**. [3] In this process, a single element is balanced over the entire equation. Then, more elements are balanced and the resulting coefficients corrected until the entire equation is balanced.

This process is not very useful for implementing a program because it is not systematic: steps may be performed differently or in different orders, depending on the equation to be balanced. Even a certain equation, such as the example could be balanced in a different way.

2.1.2 Balancing by Trial and Error

Another unsystematic process is to randomly select coefficients until the equation is balanced. This process is also unsystematic, but easy to programatically implement. The complexity of such a method increases exponentially; for an equation with 4 terms and a maximum coefficient of 10, balancing the equation may take, at worst, $10^4 = 10,000$ guesses.

If we expect that equations will generally have few terms and low coefficients, the massive processing power of modern computers will likely make the inefficiency of such a method unnoticeable to user. In fact, this method may

Algorithm 1 An example of balancing by inspection

$$
\begin{aligned}
C_2H_6 + O_2 &\rightarrow CO_2 + H_2O && (1)\\
C_2H_6 + O_2 &\rightarrow CO_2 + 3H_2O && (2)\\
C_2H_6 + O_2 &\rightarrow 2CO_2 + 3H_2O && (3)\\
C_2H_6 + {}^7\!/_2O_2 &\rightarrow 2CO_2 + 3H_2O && (4)\\
2C_2H_6 + 7O_2 &\rightarrow 4CO_2 + 6H_2O && (5)
\end{aligned}
$$

(1) The combustion of ethane, (2) There are 6 hydrogens on the left-hand side, so there must be 6 on the right (3 groups of 2), (3) There are 2 carbons on the left, so there must be 2 on the right, (4) Now there are 7 oxygens on the right (2 groups of 2 = 4 from CO_2 and 3 from H_2O), but oxygens come only in groups of 2, 5Each coefficient must be multiplied by 2 to clear the fraction.

Algorithm 2 The representation of a chemical equation as a system of linear equations

$$
\begin{aligned}
H_2 + O_2 &\rightarrow H_2O && (6)\\
aH_2 + bO_2 &\rightarrow cH_2O && (7)
\end{aligned}
$$

$$
\begin{cases}
2a &= 2c\\
2b &= c\\
a &= 1
\end{cases} \qquad (8)
$$

even perform better on small problems than more complex solvers, with fixed costs to create tables and costs per equation evaluated.

Besides the likelihood of the program never terminating when faced with a complex equation, this method is also unsatisfying because it is simply an application of brute-force, instead of finding a logical solution.

2.1.3 Balancing using Systems of Linear Equations

An elegantly logical method is to convert a chemical equation into a system of linear equations [5], with one equation representing each element, the number of each element representing a coefficient to an unknown variable, representing the chemical equation coefficient for a substance. To make the system solvable, the first unknown is arbitrarily set to 1.

The resulting values can then be multiplied by the greatest common denominator of the values to remove any fractions.

The process of converting a chemical equation to a system of linear equations, which is performed by a parser, can be made even simpler to implement by

Algorithm 3 Systematic representation of chemical equation (6)

$$\begin{cases} 2a + 0b - 2c & = 0 \\ 0a + 2b - 1c & = 0 \\ a & = 1 \end{cases} \tag{9}$$

$$\left\{ \begin{array}{ccc|c} 2 & 0 & -2 & 0 \\ 0 & 2 & -1 & 0 \end{array} \right\} \tag{10}$$

creating equations that are all in the same form (9) or by creating a matrix [2] (10):

2.1.4 Methods of Solving Systems of Linear Equations

When a system of linear equations is converted into a matrix, it can be solved by performing Gaussian elimination on the matrix. [2] The unknowns can then be found from the matrix.

The parser in Chemlogic produces a structure that is then simply transformed into a matrix, but it is not solved directly, but is converted into a system of linear equations, to take advantage of the built-in linear equation solver (CLP(q)) provided by SWI-Prolog.

This allows for provided code to be re-used and makes programming simpler, at the cost of inefficient conversions between forms.

2.2 Using DCGs to Parse Chemical Names, Formulas and Equations

2.2.1 About DCGs

In any program that deals with user input, the input must be converted into a form that can be processed and interpreted. This process is called parsing.

A parser recognizes a formal grammar and can produce structures that can be manipulated programmatically from information contained in the inputs. The parsing facility provided in many logic programming languages, including Prolog, is an implementation of DCGs (Definite Clause Grammars).[4] This type of grammar maps a grammatical rule to a logic programming clause. They can be simply described as a simplified syntax for creating functions that process a grammar using lists.

In Prolog, DCGs are implemented using difference lists, an extremely efficient representation that has many useful properties.

Chemlogic uses grammatical rules to describe valid chemical names, formulas and equations. A simple example is shown in Figure 1 on the following page.

Figure 1: A very simple grammar (binary ionic compounds)

```
compound --> metal, " ",non_metal_base, ide.

metal --> "sodium".
metal --> "calcium".
metal --> "silver".

non_metal_base --> "chlor".
non_metal_base --> "ox".
non_metal_base --> "sulf".

ide --> "ide".
```

Figure 2: The internal form of the grammar from Figure 1

```
compound(Compound, Rest) :-
        metal(Compound, MetalRest),
    MetalRest=[' '|SpaceRest],
        non_metal_base(SpaceRest, NonMetalRest),
        ide(NonMetalRest, Rest).

metal([s, o, d, i, u, m|Rest], Rest).
non_metal_base([c, h, l, o, r|Rest], Rest).
ide([i, d, e|Rest], Rest).
```

2.2.2 Difference Lists and the Implementation of DCGs

Many programming languages have their own definition for a list. In Prolog, a list is a structure consisting of an arbitrary number of elements. Elements can be of any valid Prolog data type, most relevantly an atom (a single unit of characters) and nested sub-lists.

Difference lists consist of an instantiated part, the head and an uninstantiated part, the tail. The tail is always paired with the rest of the list. This can be a difficult concept to understand and the major use of difference lists is in DCGs, where it is not exposed to the programmer. See Figure 2 for an example of difference list usage.

In DCG clauses, the head is matched with the the input and any remaining data is returned as the tail, which is passed to the next clause, which performs the same matching until the entire input is parsed and the remaining tail is given back to the user (or rejected as a syntax error in Chemlogic). If any clause fails, Prolog will backtrack to find another clause that can satisfy the grammar. If none is found, then the parsing fails. To better explain the function of difference lists in practice, a sample parsing is shown in Figure 3 on the facing page.

Figure 3: An simplified trace of the grammar in Figure 1

```
 1  [trace]  ?- compound("sodium chloride",Rest).
 2           metal("sodium chloride", MetalRest)
 3  metal("sodium chloride", " chloride")
 4
 5           non_metal_base("chloride", NonMetalRest)
 6  non_metal_base("chloride", "ide")
 7           ide("ide", IdeRest)
 8  ide("ide", [])
 9  compound("sodium chloride", [])
10
11  yes.
```

1. Parse “`sodium chloride`” as a compound.
2. Parse “`sodium chloride`” as a metal leaving `MetalRest`.
3. “`sodium`” is consumed (the head), leaving “` chloride`” (the tail).

—

5. Parse “`chloride`” as a non-metal base name leaving `NonMetalRest`.
6. “`chlor`” is consumed (the head), leaving “`ide`” (the tail).
7. Parse “`ide`” as an `ide` terminal.
8. “`ide`” is consumed (the head), leaving the empty list, `[]` (the tail).
9. Therefore, “`sodium chloride`” is a compound with no remaining characters.
10. The result of the parsing.

2.2.3 Extra Arguments

The parsers that have been described so far can only state whether or not a given input conforms to a grammar or not (in Figure 3 on the previous page the answer was "`yes`" — it is valid). In order to make a program actually perform any tasks, some information must be recorded about the input that is parsed (Figure 4 on the facing page). This information is then in an internal representation which can be manipulated by the other parts of the program. Commonly, the internal representation is an Abstract Syntax Tree (implemented in Prolog using terms), but in Chemlogic a simple list is used. The AST provides the advantage of the various components of the input being labeled (e.g. `(metal(Na), non_metal(Cl))`), but lists are easier to manipulate.

2.2.4 Semicontext Notation

Another difficult concepts in DCGs is semicontext notation. It allows elements not originally present in the input being processed to be added to the parsing list. After the grammar rule is parsed, the pushback list is added to the front of the parsing list (see Figure 5 on page 12).

Semicontext notation can be used to implement lookahead, where a parser checks the next element in a list without processing it. This is used to make decisions about how to parse a certain element based on elements that occur after it. A lookahead is implemented by removing the item to be tested, testing it, then using the pushback list to put it back on the list of tokens to be parsed. Decisions can then be made based on the results of the test.

Figure 6 on page 12 demonstrates the use of a simple lookahead.

3 Implementation

3.1 Notes for the Equation Balancer

3.1.1 Building the Matrix

The equation balancer implements the linear equation balancing algorithm described in 2.1 on page 5 in multiple processes.

First, the parser is configured to produce structures (as described in 2.2 on page 7) containing the formulas involved in an equation, the elements involved in an equation and a structure grouping the formulas into reactants and products (equation structure).

The element structure is sorted and duplicates are removed to produce an element set.

Then, the equation structure is tabulated: a lookup table is produced which records the number of occurrences of an element in a given formula. Polyatomic groups are flattened by this process (their subscript is distributed to their elements and the elements are added by the same process as is used when they

Figure 4: A new version of the parser from Figure 1, using extra arguments and its result

```
1  compound([Metal,...,NonMetal,...]) --> metal(Metal), "
        ",
2  non_metal_base(NonMetal), ide.
3
4  metal("Na") --> "sodium".
5  metal("Ca") --> "calcium".
6  metal("Ag") --> "silver".
7
8  non_metal_base("Cl") --> "chlor".
9  non_metal_base("O") --> "ox".
10 non_metal_base("S") --> "sulf".
11
12 ide --> "ide".
```

1. Extra arguments are passed to the parsing rules. The arguments are assembled into a list.

—

3—5. Values are provided for these arguments (the element symbol).

—

7—9. Values are also provided by this predicate.

—

11. This predicate does not record any information.

```
1  ?- compound(Compound,"sodium chloride",Rest).
2          Compound = [['N', a], ..., ['C', l], ...],
3          Rest = [] .
```

1. A new argument is added to the beginning of the query: the `Compound` structure.
2. The structure created for “`sodium chloride`” is `[Na,...,Cl,...]`.
3. There is nothing remaining after parsing — the input conforms to the grammar.

Figure 5: A trivial example and usage of semicontext notation

```
1   censor,
2           "secret metal"
3           --> "sodium".
```

1. Define a rule called `censor`.
2. Push “`secret metal`” to the front of the list, if...
3. “`sodium`” is parsed.

```
1   ?- censor("sodium",Rest).
2           Rest = "secret metal".
```

1. Call our completely useless predicate.
2. Notice that it did not parse anything — it just made “`secret metal`” the remaining unparsed list.
This predicate could perhaps be connected to another rule that processes compounds — it will receive the censored version.

Figure 6: A simple example and usage of a lookahead implemented with semicontext notation

```
1   is_vowel_drop,
2           [Tested]
3           --> [Tested],
4           {Tested = 'a'; Tested = 'o'}.
```

1. Define a rule called `is_vowel_drop`.
2. Push the content of `Tested` to the front of the list, if...
3. `Tested` is removed from the list, and...
4. `Tested` is `a` or `o`.

```
1   ?- is_vowel_drop("oxygen",NonMetal).
2   NonMetal = [o, x, y, g, e, n].
3
4   ?- is_vowel_drop("chlorine",NonMetal).
5   false.
```

1. Should we drop the vowel from our prefix if the non metal is oxygen?
2. Yes! Notice that “`oxygen`” has been left untampered and can be parsed by a non-metal rule.
—
4. Should we drop the vowel from our prefix if the non metal is chlorine?
5. No! Our rule will fail and a clause that doesn't drop the vowel will try to process this compound.

are on their own).[1]. The process of creating a lookup table is commonly used in chemical equation balancers — see [2] for a balancer that uses a roughly similar procedure.

To finally construct the matrix, each element is evaluated and its count in each formula is recorded.

3.1.2 Solving the Equations

To use Prolog's built-in equation solver, the matrix is converted back into an equation, a process made simple by Prolog's ability to treat code as data (metaprogramming, 3.4 on page 15).

The solver used in Chemlogic is CLP(q) (Constraint Logic Programming over Rational Numbers)[1], which is built-in to SWI-Prolog. This solver is used since the coefficients of a chemical equation are always rational numbers, but may sometimes not be integers before the fractions are cleared.[2]

CLP(q) allows input of equations in a simple notation and provides the solution to the unknowns by instantiating Prolog variables used in the equations. The results are always reduced to lowest terms using the provided predicate `bb_inf`.

3.1.3 Providing Results to the User

The solved coefficients are re-inserted into the structures, which are then provided as results to a parser. The parser returns the input that would create those structures (i.e. human-readable equations).

The program could be made more efficient by not re-parsing the structures to provide output, instead inserting the coefficients directly into the human-readable input by some method.

3.2 Notes for the Formula and Name Processing Code

3.2.1 Validation

Chemlogic uses a formula parser to convert formulas inputted by users to their internal representation in Chemlogic. The formulas are verified to ensure that they are in the correct form, but whether or not they represent valid compounds is not verified, to support balancing equations with types of compounds that are not known to the program. When parsing a name or converting a formula to a name, the input is checked to ensure that the compounds conform to the types known by Chemlogic.

[1]For example: NH_4NO_3 is flattened to `[N2, H4, O3]`

[2]A more common solver, built-in to nearly all Prolog systems, CLPfd (Constraint Logic Programming over Finite Domains) had annoying limitations regarding this situation.

3.2.2 DCG Translation

The chemical information database is not used directly. Instead, a file called `fact_dcg_translate` converts the facts to grammatical rules, providing base names, formulas, full names, symbols, etc. in whatever output format is selected (see 4.2.3 on page 19 for a brief discussion of output formats).

This process is cached when Chemlogic is compiled, so it is only re-run when the database is changed.

3.2.3 Element Structure

In order to create the element structures used by the balancer, the formula and name parsers record the elements contained in the input in a flat structure created by using additional difference lists as extra arguments to the DCG (see 2.2.3 on page 10 for a discussion of extra arguments).

3.3 Handling User Syntax and other Errors

3.3.1 About Error Handling in Chemlogic

The features and algorithms discussed so far deal only with cases when user input conforms to the grammars. If the user makes a mistake, the only information he is given is "`false.`"

In order to make Chemlogic truly useful, it must identify where in the input the problem has occurred and provide clear error guidance.

Clearly identifying a syntax error is a difficult problem: even many programming languages cannot provide clear guidance as to what is wrong. Implementing error handling took nearly as long as writing the rest of Chemlogic and the process of implementation uncovered a few bugs and poor design decisions in the parsers.

3.3.2 Raising a Syntax Error

Whenever a predicate that must succeed for a given input to be valid fails, a syntax error is raised. The error term contains a code name for the error that will be used to look up messages and the list of unparsed tokens remaining that can no longer be parsed because of this error. In Prolog, this is done by `throw`ing an exception. Exceptions stop the execution of the program and backtrack through the program until a handler can be found — the `catch`er.

The actual process of raising the syntax error is simplified with the `xx` operator (see 3.4 on the facing page).

3.3.3 Explaining the Syntax Error

The exception is then given to the error explaining library. It first attempts to localize the error by searching for the problematic part in the unparsed tokens. It does this by testing the character type of the first character, then implementing rules depending on the type (e.g. if the first character is a letter, read until a number or capital letter is reached, etc.) The erroneous part is recorded.

Messages are then looked up using the error code and the character type. Knowing what type of character has triggered the error can often pinpoint the problem and allow for different messages to be used when a logic error is suspected (e.g. bad letters in a compound name) or when a simple typing error is suspected (e.g. unusual symbols like ^ in a compound name).

A structure is created from the information determined by the error explainer. It is then re-thrown — the exception is allowed to pass all the way back to the predicate that called the parser. The calling predicate is an interface function (from either the command-line or the web interface). Using metaprogramming techniques, the exception is then given to a handler that knows how to render the error in that interface (e.g. `cli_error` knows how to highlight the erroneous part using ANSI color).

3.4 Simplification of Program Code using Metaprogramming

Many functional and logical programming languages, including Prolog have strong support for metaprogramming. Metaprogramming allows a program to write or manipulate parts of itself (or other programs). [10]

In the Fact—DCG translator (Figure 7), data is extracted from a fact and is then reassembled into a new term, which is asserted as a grammatical rule. This ability to manipulate code as data is a very useful feature. In this case, using metaprogramming saves the database builder from manually entering data in these awkward forms.

Figure 7: `fact_dcg_translate.pl` converting a group fact into a lot of rules

```
1  cl_poly_to_dcg(group([["N",1],["H",4]],"ammonium","n/a
       ")) :-
2          Clause =.. [_Functor,Sym,Name,Base],
3
4          dcg_translate_rule(group(ElemsL,ElemsR,Sym,
              Name) --> Name,FullRule),
5          assertz(FullRule),
6
7          dcg_translate_rule(group_base(ElemsL,ElemsR,
              Sym,Base) --> Base,BaseRule),
8          assertz(BaseRule),
```

1. An input clause (code)
2. The clause (code) is separated into data

—

4. A new clause (code) is translated from the data
5. The new code is loaded into the program

Chemlogic also makes use of the ability to call an arbitrary predicate at run-time (a common programming language feature) to avoid writing many different variants of the same predicate (Figure 8).

Figure 8: covalent.pl using call to implement multiple variants of the same code

```
1   systematic_covalent([[Sym1,Num1],[Sym2,Num2]]) -->
2           covalent_part(nonmetal,Sym1,sub_first,Num1),
               !,
3           " ",
4           covalent_part(nonmetal_ide,Sym2,sub_general,
               Num2) xx covalent_part_2.
5
6   covalent_part(SymGoal,Sym,NumGoal,Num) -->
7           {fwd_flag(Num,Hack)},
8           call(NumGoal,Num,Letter),
9           call(SymGoal,Sym,Matched,_),
10          (
11                  {double_vowel_test(Letter,Matched)};
12                  fwd_stop(Hack,vowel_required)
13          ).
```

—

2. Call a predicate telling it to use non-metal (without ide ending) and prefixes for the first element in a compound.

—

4. The same predicate now instructed to use non-metal with ide ending and to use normal prefix rules

—

7. Doing some processing. These lines of code don't have to be written n times.
8 and 9. Call the predicates instructed
10—13. Perform operations on the results of the predicates.

The most interesting use of metaprogramming is the ability to define new operators that extend the programming language with a simple syntax for a repetitive task.

In Chemlogic, two operators are defined:

- `xx`, the syntax error operator: it executes the predicate on its left-hand side and, if it fails, raises the given syntax error on its right-hand side. It is used in parsers to stop whenever a grammatical rule fails. (See Figure 9 on the next page).
- `handle`, the error operator: it executes the predicate on its left-hand side and, if a syntax error occurs it calls correct handler function for the interface, then returns the result on its right-hand side. It is used in interface code to show the correct error message

Figure 9: The xx syntax error operator: Definition and Use

```
1  :- op(990,yfx,xx).
2  :- meta_predicate xx(//,?,?,?).
3
4  Condition xx (SyntaxError,Flags) -->
5          {Condition = Module:_},
6          (
7                  Condition, !;
8                  syntax_stop(Module:SyntaxError,Flags)
9          ).
```

1. Declare xx as an operator
–
4. Define a predicate
5—9. The operation performed for the given predicate

```
1  formula(Fmt,Elems,ElemsR,Formula,FormulaR) -->
2          formula_part_first(Fmt,Elems,ElemsR0,Formula,
               FormulaR0) xx part_first,
3          (hydrate_part(Fmt,ElemsR0,ElemsR,FormulaR0,
               FormulaR), !).
```

—
2. Call formula_part_first, otherwise declare a part_first syntax error.

There are many potential places where more metaprogramming could be added to Chemlogic: the various extra arguments used in the DCGs could be simplified with special operators and a goal/term expansion function. This would massively simplify the parsers. Also, the Web interface has some code that performs the same steps, but with different predicates. These could be combined with the use of runtime calls.

4 Interfaces and Miscellaneous Code

4.1 Android Application — A new user interface

4.1.1 Summary

The Chemlogic App for Android was developed to allow users to use the program on a mobile device. Due to the popularity of mobile devices and the convience of mobile applications, this improvement greatly increases the number of users who would use the program and enables students to easily use Chemlogic as a study tool for chemistry classes.

The existing code of Chemlogic, written in Prolog, was cross-compiled for the processors used in Android devices and is used unmodified, simplifying the development of new features.

The App consists of a user-interface, implemented in Java, using the Android APIs, and a system package consisting of the cross-compiled Chemlogic code, and its dependencies.

4.1.2 The Implementation of the User Interface and the System Package

The user interface renders the design of the app (textboxes, buttons, etc.), verifies the input to ensure that it does not contain invalid Prolog syntax and then communicates with Chemlogic through a UNIX pipe. Upon receiving the response, it renders the formatting of the result (e.g. subscripts, arrows and colored error messages) and displays it. The user interface also adds an extra row of keys to the keyboard which contains symbols commonly used in chemical equations and formulas.

The system package, which is approximately 4 MB in size, contains a copy of Chemlogic, compiled as a stand-alone application (meaning that it includes the Prolog interpreter in its binary), some required Linux libraries and an initialization script. When loaded, the App interface executes the initialization script, which starts Chemlogic by running the dynamic linker to locate and link Chemlogic with the provided libraries and then executes it. The cross-compilation process required to build the system package is extremely complicated, so it has been automated to reduce developer work and eliminate errors. Because the Android platform does not install the assets provided with the App, a procedure was implemented that verifies whether latest version of the system package is present and will extract the files and install them, if necessary.

Once Chemlogic is installed and started, the user interface writes a command to the pipe, when Chemlogic reads. Chemlogic writes its answer to the pipe, which the interface reads. This method could be replaced with bindings between Prolog and Java, which would simplify the development of the App, but would increase the complexity of the cross-compilation process.

4.1.3 Interprocess Communication in the App

The pipe model is, however, one of the most interesting parts of the UNIX model: everything is a file, everything is text, anything can read text and anything can write text. It allows many utilities to be connected to each other, each reading the previous one's output and writing to the next one. User input/output is, in fact, implemented using this model as well: the keyboard is piped to the input and the output is piped to the screen. This method of interprocess communication raises an interesting dilemma: How do the interface and Chemlogic know when the other end of the pipe finishes transmitting data (like a query or a response)?

Unlike a subroutine which automatically returns a variable to its caller, the two programs must be explicitly instructed to look for a specific pattern in order to know when to proceed.

When a user interacts with a command-line program, patterns are recognized by both the human user and the program itself: When the user enters a command, he presses ENTER. Now the system sees that the user has finished typing and it processes the input and displays its response. When the interactive prompt is displayed, the user sees that the system is ready to receive another query. The user interface and the Android App use these patterns, as well, allowing existing code to be used with minor modifications.

It must be noted that although the pipe model would seem to imply transmission and communication between computers (and often can), in this case the communication is entirely virtual — it is between multiple processes on the same computer. The mechanism is almost exactly the same from the point of view of the two programs that are communicating over the pipe.

4.2 Web Interface — A user-friendly way of using the program

4.2.1 Advantages of Web Interfaces

Command-line interfaces are often not user-friendly. Terminals also cannot display subscripts and special characters, making the output an approximation of the correct form of an equation or formula.

A web interface can provide a graphical interface to a program, making it simple to use select features and options, that is also available over the network.[3]

4.2.2 SWI-Prolog's Web stack

SWI-Prolog has a built-in Web stack, [6] providing a server that can run Prolog programs and send their output over HTTP. To avoid writing HTML manually and handling arguments, SWI-Prolog provides an excellent library that allows HTML to be written as Prolog terms and input to be validated. To keep code organized, a library that can insert menus, stylesheets and other header elements. These features make writing Web applications in Prolog even simpler than in many common languages used in Web programming.

The web libraries make excellent use of DCGs and metaprogramming in their implementations, making code easy to understand.

4.2.3 Output Formatting

In order to produce the correct formatting codes for each interface (HTML entities in this case), the handling of symbols is decoupled from the various parsers and is instead implemented in a common formatting library. This allows for readable and correct formatting in the Web interface.

[3]This allows an entire computer lab to use a single copy of Chemlogic installed on a server, for example.

If a copy of the `chemweb` (the web interface is running), users can access it by going to: `http://<ip-address>:8000/chemlogic/`

Figure 10: The entire code of the `chemcli` DSL

```
:- op(990,xfy,::).

InputType - Input :: -Result :-
        InputType - Input :: symbolic - Result, !.

InputType - AtomInput :: ResultType - Result :-
        atom_chars(AtomInput,Input),
        (
                InputType = name -> name_2_formula(
                   Input,StringRes) handle _;
                InputType = formula -> formula_2_name(
                   Input,StringRes) handle _;
                balance_equation(InputType,Input,
                   ResultType,StringRes) handle _
        ),
        atom_chars(Result,StringRes).
```

A programming language in 13 lines of code.

4.3 Command-line Interface — An extremely simple DSL

During the development of Chemlogic, it was often necessary to run various predicates in order to test the program. Calling Prolog predicates, as is, is not a user-friendly command-line interface, however. It requires the user to enter his query as a Prolog statement and gives back the answer as a list of characters.

Using metaprogramming techniques, like defining operators, calling at runtime and syntactic unification, it possible to build a very simple DSL (domain specific language) for querying Chemlogic (see Figure 10). Despite the fact that it only implements 4 simple features, with 2 rules, language statements can be composed into more complex programs. Prolog statements can also be used in the DSL and language statements can be combined with Prolog statements seamlessly.

The DSL offers an elegant way of balancing chemical equations and converting formulas to names and vice versa by offering a simplified syntax and automatic conversions to and from Prolog strings (see Figure 11 on the next page).

Because of its simplicity, the DSL is mostly a proof of concept and more syntax will need to be added (and the features to expose from Chemlogic) to make writing it possible to write interesting programs in this language.

Figure 11: Simple `chemcli` queries and an example of a program that can be written using it

```
1 ?- formula - 'CuCl2' :: -Name.
2 Name = 'copper(II) chloride'.
```

1. Input a formula, get default output.
2. The result, with human readable formatting.

```
1 ?- name - 'baking soda' :: -F, formula - F :: -
     CanonicalName.
2 F = 'NaHCO3',
3 CanonicalName = 'sodium hydrogen carbonate'.
```

1. This line finds the “canonical” name for a compound when given another name for it.
2. The name is converted to a formula.
3. ...and back into a now systematic name.

5 Discussion

5.1 Suitability of Prolog for application

Initially, Prolog was chosen because of its simple fact database and built-in support for parsers as DCGs and for linear equation solvers. Using a logic programming language would also reduce programmer work by describing the results, not the process and by offering metaprogramming features.

In practice, the simple fact database was no more convenient than in any other programming language because the facts (the chemical information database) had to be converted to DCG rules by a translator in order for them to be used efficiently and correctly in parsers.

DCG support in Prolog and similar languages makes it possible to easily write parser for all sorts of grammars. In most programming languages, the programmer must write the parser from scratch or use a contributed library. Chemlogic ended up using many advanced features of DCGs, including extra arguments for constructing parsing structures, using the underlying difference list representation for certain structures and using semicontext notation for some simple lookahead rules.

In some places, parsing rules have become complicated and hacky, but overall DCGs have made it simpler and more logical to construct grammars and extend them for many types of data a chemistry program uses.

As discussed in the section on the linear equation balancing algorithm (2.1.4 on page 7), using Prolog’s linear equation solver entails producing a system of linear equations, which is probably re-converted to a matrix in order to be solved. Using the library certainly simplified the code, because matrix operations are difficult in many Programming languages and because implementing Gaussian elimination requires a lot of mathematical understanding.

Using a logic programming language made the code simpler and more elegant. While the idea of describing results instead of process is somewhat exaggerated, programming in Prolog avoids manual writing of loops, if statements and other imperative constructs. It can be rather amazing, when a situation can finally explained in Prolog, how few lines of code are necessary to do something. [4] Metaprogramming was not used heavily but it proved to make code simpler by allowing predicates to be generated from other predicates, by allowing predicates that call other predicates and by defining convenient operators. This last aspect proved to be the most interesting and was used as the basis of the extremely simple `chemcli` DSL (4.3 on page 20). The support of metaprogramming is more extensive in Prolog than in many languages and it was relatively simple to make use of this support.

5.2 Methods of Analyzing Performance in Prolog

SWI-Prolog, like many programming language implementations, offers a few tools for analyzing a program's performance. It includes `gxprof` (A graphical profiler), `time` (a command-line predicate offering a few simple statistics) and `statistics` (a command-line predicate offering very detailed runtime statistics).

`gxprof` primarily performs time efficiency analyses, which were often not useful for Chemlogic because the time taken by different implementations were often so small as to be incomparable. Therefore it was not used during the development of Chemlogic.

The most commonly used analysis function was `time`, which provides a few interesting statistics, including inferences and Lips (logical inferences per second?). I am unsure of exactly what the inference count measures or its connection to the actual performance of the algorithm. Inferences were a useful way of comparing algorithms and implementations because it gave consistent results between runs and on different computers.

Analysis was performed mostly by recording the performance of an predicate for arguments of various sizes to try to find the fixed costs (intercept) and cost per element (slope) and to ensure that there were no algorithms with exponential cost. Inefficient pieces of code were found using the debugger and new solutions were compared against old ones.

`statistics` may provide details that could be useful for future research, but on a small program like Chemlogic analyzing some of these statistics may be overly complicated.

It must be noted that the performance of algorithms was not formally analyzed (by creating mathematical definitions and determining the fastest grow-

[4]Perhaps this is why Prolog programmers name their variables A, X, Acc, H, T, R0, etc and do not document things in detail?

ing terms to find the worst-case performance, for example) primarily because no novel algorithms were introduced in this program, nor unusual adaptations.

Formal analysis of a specific parser may enable useful improvements to be made.

5.3 Organization and Structure of Code

The organization of code is quite clear overall. I have tried to follow proper naming, indention and structuring practices. Prolog allows for code to be separated into modules with private predicates used only inside a single module and public predicates that are useful to other modules. Modules were frequently used to organize Chemlogic.

5.3.1 Modularity of Interfaces and Error Handlers

Since Chemlogic needed to support a Web interface to its Prolog predicates which were used from the command-line, it was very important to allow for the de-coupling of formatting and interfaces from the code itself. The interfaces for Chemlogic often implement the bare-minimum required to make the program modular, but there are a few over-engineered parts that may prove useful when improving the detail of error messages.

The error handling system itself was an experiment, using Prolog exceptions, handlers and meta-predicates that call modules. The design of the error handling system is modular and relatively organized, but more experience with Prolog exceptions may allow for improvements to the design.

5.3.2 Some Confusing Points

There are few confusing interactions between pieces of code, with a few small cases between the ionic, oxyanion and acid functions. The lack of a tokenizer proved to make implementing error handling problematic in some places, as it was always necessary to determine whether or not a certain parser can safely give error messages or whether it must simply fail and give the next parser a chance to process the input. A tokenizer can quickly determine which parser to use and avoid problems like these.

Some inefficient tokenizers are used to highlight the erroneous component of user input. Since they are used only when a problem occurs, they are not much of a concern.

The only uses of semicontext notation in this program were some hacks to allow naming for pure substances to avoid triggering error messages. This is a good place to target for improvement if a tokenizer is introduced.

5.3.3 A Note about Covalent Prefixes

Correctly handling the rule that the last vowel (if a or o) of an IUPAC prefix is dropped if the succeeding element starts with an a or o was quite difficult to

Figure 12: Examples of vowel dropping in IUPAC prefixes

Prefix	Element	Result
mono	oxygen	**mon**oxygen
mono	carbon	**mono**carbon
tetra	oxygen	**tetr**oxygen
tri	iodine	**tri**iodine

implement. Figure 12 shows some examples of the rule.

It seems to me that this is a case of a Context-Sensitive Grammar, where a certain token may or may not be permitted at a certain point given its preceding token.

Currently a repeated test and a metaprogramming call is used to determine the right prefix to use.

5.4 Tokenizers: Simpler and More Performant Code

Very recently, SWI-Prolog introduced a new string type. [8] The string type solves many problems that have traditionally required complicated workarounds in Prolog.

The type replaces the use of atoms for manipulated strings, or lists of character codes with a more efficient data structure. The most important (for Chemlogic) change is the introduction of a predicate (split_string/4) which allows for a string to be broken into parts delimited by a character and allows extraneous padding characters to be removed. The string type can still be operated on as a list of character codes when needed (as in DCGs).

The benefits to a program like Chemlogic from this change are quite large: it is now simple to implement a tokenization step for the various parsers, like most compilers and interpreters (You will recall that Chemlogic does not tokenize its strings before parsing them).

Currently, when processing a string like “carbon monoxide”, the program must go through many testing steps (is it an ionic compound, is carbon a metal, is it an acid, is carbon a polyatomic ion, and their many sub-tests etc.) But, if the string is tokenized into [“carbon”,”monoxide”], it is obvious it is not an ionic compound, because the first token is not a metal and its obvious that it is not an acid because the second part is not “acid”.

In the case of determining if it is an acid, tokenization makes implementing a lookahead parser simpler: one element lookahead instead of n character lookahead (because it is not yet known how long the first part of a string is in that case).

The tokenizers currently in use for error parsing may be able to be re-written in a simpler way using the new string type.

5.5 Ideas for Additional Features

Many ideas for new features are discussed from a programmatical perspective in the `TODO` file distributed with the program and as an appendix to this paper (Appendix C on page 28). Some of them are summarized here:

In order to handle more complex types of organic compounds, the program must be greatly extended to support manipulating structural formulas (when given a name as input), displaying diagrams of them and to support an input method for structural formulas. Since balancing and many other parts of the program operate using molecular or empirical formulas, a module can be written to "flatten" structural formulas, allowing for code to be re-used.

5.6 Further Research

5.6.1 Algorithms

As discussed in 2.1.2 on page 5 (Balancing by Trial and Error), the performance of the trial and error algorithm may be better than the system of linear equations algorithm for simple cases. It would be interesting to compare the algorithms using the inference testing method described in 5.2 on page 22. The performance of this algorithm will depend on the efficiency of a test for whether the equation is balanced or not. The ranges of complexity in which each algorithm performs better and the appropriate algorithm selected for a specific balancing.

If Chemlogic is re-implemented, with an initial tokenization step, the performance of the program (both using inference testing and wall-time measurements, if possible. [5]) before and after the change can be compared. The cleanliness and simplicity of the code, in practice, should also be compared. Such a change may also improve the quality of the error messages.

5.6.2 Performance Analysis of the Whole Program

As described in 5.2 on page 22, performance analysis was performed mostly on small components of a program. But since changes in one part may affect another part, it will be useful to make comparisons over an entire program for a given input in order to find out if a certain algorithm is truly better or if it simply off-loads costs elsewhere.

5.6.3 Balancing for Complex Redox Reactions

For some difficult redox reactions, the currently used algorithm will produce a solution that satisfies the system of linear equations, and therefore is balanced, but is not actually what occurs when the reaction takes place.

This is because the system of linear equations has more than one solution, and the solver always picks the simplest form.

[5]Currently the individual algorithms in Chemlogic do not take more than 0.005 s to run. The new algorithm may perform in similar time, or the time may be affected by other factors.

Figure 13: Error messages for a missing charge on a multivalent metal

(a) Current error message

The element you have entered is multivalent. You must provide the charge in the highlighted space.
NOTE: Use capital roman numerals in parentheses
e.g. (II)

(b) Improved error message exposing chemical information

The element you have entered is multivalent. You must provide the charge in the highlighted space.
Copper ions have the following charges: Cu^{+} — **copper(I) and** Cu^{2+} — **copper(II)**
NOTE: Use capital roman numerals in parentheses
e.g. (II)

To correctly handle these types of equations, it will be necessary to find under what conditions this problem occurs. It will also be necessary to determine how to distinguish between the various solutions and determine the correct one.

It will perhaps be necessary to write a new algorithm for balancing, based on oxidation numbers or perhaps on half-reactions.

This algorithm will not be necessary in all cases. It may be possible to determine when to use this algorithm. Otherwise, a user-selected option will be necessary (the "Complex Redox" tab).

5.6.4 Messages and User Communications

The messages in Chemlogic have been written to explain, primarily, the specific syntax error a user is making and how the syntax error relates to Chemistry. The messages have been reviewed and were simplified, corrected or clarified according to advice.

The messages are written at a high-level, discuss the Chemistry in abstract and often refer to the idea of unnecessary or incorrect characters. Testing the error messages produced by the program with actual users (Science students) will make the program more useful as a means of reviewing Chemistry. It may be necessary, perhaps, to adjust the messages to make them more clear, given what triggers them most, in practice. Perhaps it would be useful to refer the user to sections from a textbook.

Improvements to the error handling functionality could allow for further distinctions in the errors triggered, beyond error code and token type. A particularly useful, but tedious, change would be to use templates to insert specific information about the problem in a given input (see Figure 13 for an example).

Acknowledgments

I would like to thank the many people who gave advice and helped with the project.

I am particularly grateful for the valuable assistance provided by Dr. Peter Tchir, my Physics, Chemistry and, now, Computer Science teacher. His help and advice, especially with algorithms and his support for my Computer Science projects helped make this program possible.

References

[1] C. Holzbaur, *OEFAI clp(q,r) Manual Rev. 1.3.2*, (1995). 13

[2] Nayuki Minase, *Chemical equation balancer (JavaScript)*, 2013. 7, 13

[3] L. Sandner, *BC Science 10*, McGraw-Hill Ryerson, 2008. 5

[4] Markus Triska, *DCG Primer*. 7

[5] Mark E. Tuckerman, *Methods of balancing chemical equations*, (2011). 6

[6] Jan Wielemaker, Zhisheng Huang, and Lourens van der Meij, *SWI-Prolog and the Web*, Theory and Practice of Logic Programming **8** (2008), no. 3, 363–392. 19

[7] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager, *SWI-Prolog*, Theory and Practice of Logic Programming **12** (2012), no. 1-2, 67–96.

[8] Jan Wielemaker, et al, *The string type and its double quoted syntax*, 2014. 24

[9] Wikipedia, *Gaussian elimination*.

[10] ———, *Metaprogramming*. 15

A Obtaining Chemlogic

Chemlogic is available from http://icebergsystems.ca/chemlogic/. Chemlogic currently depends on SWI-Prolog (http://swi-prolog.org/) because it uses CLP(q), Qsave files and a few other features. See the `README` file distributed with the program for installation instructions. The installer can be used on any Unix-like system (Linux, BSD, Mac OS X) as is. The program itself will run on any operating system SWI-Prolog runs on (incl. Windows).

Chemlogic is an open-source program licensed under the GNU Affero GPL v.3 (http://www.gnu.org/licenses/agpl-3.0.html). Users of open-source programs are free to use, modify and distribute them even for profit. Copyleft licenses (like the one used for this program) also require that any changes to the program that are distributed must also be released as open-source software.

The AGPL requires that modified versions offered as a network service are also released as open-source software. If this requirement is inconvenient, please contact me.

B User Reference

B.1 Formulas

Formula CH4, Na2CO3.10H20, (NH4)2SO4

B.2 Chemical Equations

Symbolic CH4 + O2 –> CO2 + H2O

Word methane + oxygen –> carbon dioxide + water

B.3 Chemical Names

Retained Names *(preferred)* water, ammonia

Ionic copper(II) chloride, sodium carbonate decahydrate, calcium chloride

Acids hydrocyanic acid, acetic acid, sulfuric acid, hypochlorous acid

Covalent carbon monoxide, trichloride triiodide, dihydrogen monoxide

Pure Substances/Allotropes copper, silicon, trioxygen, hydrogen (= H_2)

Common Names *(accepted but not produced)* baking soda, chalk, hydrogen peroxide

C TODO

C.1 Chemistry features

- Support for structural formulas:
  - The covalent parser will have to be extended very much, to handle structures of the compounds it supports
  - The formula parser will need to have some sort of input and output representation for structural formulas
  - Each output format will have its own ways of rendering structural formulas. This will have to be extended.
  - A module will be needed to convert structural formulas to molecular formulas for balancing and other processes.

- More organic naming:
  - It will be useful to implement organic compound naming at least for Chemistry 11 to 12.
- Complex Redox reactions:
  - Sometimes, for a few very complex redox reactions, Chemlogic gives an answer that satisfies the system of linear equations (i.e. is balanced) but will not actually occur in real life.
  - A new balancing process, with a separate module should be implemented. Perhaps based on oxidation numbers or half-reactions
- Diagramming:
  - Show structural formulas of compounds
  - Bohr models, Lewis diagrams
  - Periodic tables?

C.2 Program features

- Extend the `chemcli` DSL to make it more useful.
  - Offer a way to query the chemical information database.
  - More constructs/operators.
  - A standard library?
  - This will all depend on the sorts of programs that someone will actually want to write
- Quiz program
  - Allow for questions to be generated with selectable options (the parser and perhaps major sub-parts)
  - Interactive and non-interactive usage depending on output format
  - Configurable marking (allow retry, show correct answer at end, etc.)
  - Possible to produce the same questions if passed the same seed. Some sort of intelligence, focusing on problem areas when giving questions.
- Expose the chemical information database.
- Better error messages for equation balancing errors
  - Test to ensure that all elements appear in both products and reactants
  - Perhaps explain why some charge shifts are unsatisfied
  - Explain which element makes the system unbalancable, if possible.

C.3 Organization and Structure

- Take advantage of SWI-Prolog's new string type. This makes tokenization, concatenation and many other things more efficient.
  - Chemlogic needs a simple tokenizer to break up chemical names and equations (especially)
  - At minimum, it should split on spaces and remove extraneous spaces
  - Potentially deal with character types?
  - Potentially deal with the insides of parentheses?
- Rewrite the current error tokenizers to use the new and better functions
- Tokenizers make parser much, much nicer:
  - It is now quick and simple to see if something is an acid without having to go through all of the tests
  - It can be easy to distinguish between ionic and covalent
- Make the oxyanion functions less messy and hacky. There must be a better way to tell the user what's wrong with the oxyanion names.
- The ugly hacks around pure substances can be removed with a better tokenizer
- Some things will need to be renamed and reorganized
- Use more meta-programming to remove boilerplate code from the web interface.

C.4 Bugs

- The program will get very upset if a substance is repeated:
  - e.g. H2O + H2O –> H2O
  - There is not much of a valid reason to enter this, but the program should handle this correctly
  - An error message explaining that this is junk is probably a good idea