



Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Software Engineering and Automation Department

Semestrial Project
Topic: *Domain Specific Language for Chemistry*

Semester: IV
ECTS: 10
Group: FAF – 191
Team number: 1

Authors:
Andronovici Darinela
Corman Daniel
Mămăligă Dumitraș
Trofim Iuliana
Volcov Oleg

Supervisor:
Catruc Mariana

Contents

Introduction.....	4
1. Problem analysis.....	4
1.1 Problem formulation.....	5
1.2 Solution concept.....	5
2. Domain analysis.....	5
2.1 Impact of the problem over the domain of study.....	5
2.2 Target group.....	6
2.3 Customer validation.....	8
2.4 Existing solutions.....	10
3. DSL specifications.....	11
3.1 Semantic rules.....	11
3.2 Grammar rules.....	12
3.3 Method invocation.....	13
4. Implementation	14
4.1 Go language	14
4.2 Lexing	15
4.2.1 Lexical analysis	15
4.2.2 Tokens	15
4.2.3 Lexer	16
4.3 Parsing	18
4.3.1 Abstract Syntax Tree	18
4.3.2 Parsers	20
4.3.3 Pratt parser	22
4.3.4 Call expression	22
4.4 Chemical reactions	24
4.4.1 Types of chemical reactions	24
4.4.2 Balancing chemical reactions	25
4.4.3 Solving chemical reactions	33
4.4.4 Printing chemical reactions	36

4.5 Graphical User Interface	38
4.6 Examples	39
4.7 Links to all the repositories which store the project components	42
Conclucions	43
References	44

Introduction

When most people think of a programming language, they think of a general-purpose language: one capable of programming any application with relatively the same degree of expressiveness and efficiency. For many applications, however, there are more natural ways to express the solution to a problem than those afforded by general-purpose programming languages. As a result, researchers and practitioners, in recent years, have developed many different domain-specific languages, or DSL's, which are tailored to particular application domains.

With an appropriate DSL, one can develop complete application programs for a domain more quickly and effectively than with a general-purpose language. Ideally, a well-designed DSL captures precisely the semantics of an application domain, no more and no less.

Programs written in a DSL have the following advantages over those written in more conventional languages:

- they are more concise;
- they can be written more quickly;
- they are easier to maintain;
- they are easier to reason about;

These advantages are the same as those claimed for programs written in conventional high-level languages, so perhaps DSL's are just very high-level languages? In some sense, this is true. However, programs written in a DSL also have one other important characteristic:

- they can often be written by non-programmers;

More precisely, they can be written by non-programmers who are nevertheless experts in the domain for which the DSL was designed^[1]. This helps bridge the gap between developer and user, a potentially major hidden cost in software development. It also raises an important point about DSL design: a user immersed in a domain already knows the domain semantics. All the DSL designer needs to do is provide a notation to express that semantics.

1. Problem Analysis

The average citizen has a much better understanding of the non-scientific aspects of the modern world than of its scientific aspects. His knowledge has been obtained in part through his studies in school, and in part through his reading and personal experiences. The great mass of people have not had education extending beyond elementary school, and have accordingly received no instruction in science at all. The nature of science is such that it is difficult for a man to begin his study without help.

Because of the nature of chemistry, the average person knows less about chemistry than about other sciences. He knows about many physical phenomena through his observation—he knows that objects fall toward the earth, that hot bodies emit light and heat, that an electric spark may pass between two conductors at different potentials, that a body continues to move in a straight line unless acted on by some force and so on. With this knowledge from personal experience, he is able to understand simple explanations of discoveries in the field of physics, and appreciate their significance to him and the world as a whole^[2].

His personal experiences with chemical phenomena may, however, be very limited—be limited perhaps to the observation of the chemical reactions that occur in combustion. Chemical phenomena, involving the conversion of substances into other substances, which may have entirely different

properties, are so surprising in their nature that it is difficult for a man to develop an appreciation of them without instruction.

Chemistry is a very important and widely used domain in modern society. Due to its importance, the chemistry course was introduced in middle school and high school program.

1.1 Problem formulation

Knowing that chemistry is such an important domain and that it is very hard to observe and appreciate we started analyzing it. We focused on analyzing the chemistry that is taught in schools and the learning process of pupils. We concluded that pupils meet many problems during chemistry courses.

Their lack of experience leads to the appearance of many mechanical mistakes, especially in solving a chemical equation. Pupils must equal the coefficients of the substances that are entering in reaction, which most of the time, is a challenging process. Therefore, this inspired us to develop a Domain Specific Language.

1.2 Solution concept

Our solution concept is to design and develop a Domain Specific Language for chemistry that will help balance equations, solve them and represent them graphically. This DSL will make the learning process of chemistry easier for pupils, teachers or any other chemistry-interested person.

There are several ways in which we can implement a DSL. If treated as a conventional language, conventional techniques could be used:

- Building a conventional lexer and parser based on the BNF syntax.
- Performing various high-level analyses, transformations, and optimizations on the abstract syntax generated by the parser.
- Generating executable code for some host machine ^[1].

2. Domain analysis

In the following sections, we have described the information gathered from different available sources of information like domain experts, existing software and its documentation, articles and any other documents.

2.1 Impact of the problem over the domain of study

Chemistry is a very important and widely used domain in modern society. Some of the most important products that we are using in our life, things that helped us to overcome mass deaths because of epidemics and different kind of illnesses are medical drugs and medicines, developed in a domain called pharmaceuticals. Their production would have been impossible without chemistry. Apart from this, there are domains like technology, environment, agriculture and food security, which are essential for sustainable development. Thus, it is strange that we lack the tools to make the process easier to visualize and understand.

Well-designed domain-specific languages enable the easy expression of problems, the application of domain-specific optimizations, and dramatic improvements in productivity for their users. In this paper, we describe a compiler for polymer chemistry, and in particular rubber chemistry, that achieves all of these goals. The compiler allows the development of a system of ordinary differential equations describing a complex rubber reaction - a task that used to require months - to be done in days. The generated code, like many machines-generated codes, is more complex than human-written code and stresses commercial compilers to the point of failure. However, because of

knowledge of the form of ODEs generated, the compiler can perform specialized common sub-expression and other algebraic optimizations that simplifies the code sufficiently to allow it to be compiled (eliminating all but 6.9% of the operations in our largest program) and to provide five times faster performance on our largest benchmark codes [3].

2.2 Target group

Due to its various uses, the chemistry course was introduced in middle school and high school program. The middle school (gymnasium) chemistry program consists entirely of inorganic chemistry that includes all of the interaction between inorganic chemical elements, also the basic chemical elements that are not organic. Even though this compartment of chemistry comprises only 1% of the whole chemistry that we know today, still it is very important in industries like transport, communications and energy.

At the inorganic chemistry lessons, students learn how all of the known basic elements are interacting with each other generating composed substances. At this compartment, types of reactions are studied and as an example, there are reactions that generate heat and reactions that consume heat and so on. Also here are studied the basic chemical components such as oxides, acids, salts, bases and all the possible interactions between them. According to specific rules observed during experiments and written in the solubility table that shows what kind of substance we will obtain after the reaction (a sediment or a substance that is weakly soluble, partially soluble, or very soluble). Moreover, according to this table (that is presented in **Figure 2.1**), you can observe if a reaction can take place or not due to some specific particularities (most often because of incompatibility of these substances).

Even from the beginning, at this stage students encounter many problems during the educational process, because as they have a lack of experience they make many mechanical mistakes that are annoying. In addition, in order to solve a chemical equation, the students must equal the coefficients of the substances that are entering in reaction, which most of the time students have troubles. It appears that for the gymnasium students that are solving chemical equation is not that easy to equal the coefficients, so it is a big problem.

Due to all of the problems mentioned above, there is a need for a simple to use and subject-oriented DSL that will make the educational process easier and more pleasant than it was before.

Solubility Table Common Ionic Compounds													
	Group 1				Group 2				Transition Metals				
	NH ₄ ⁺	Li ⁺	Na ⁺	K ⁺	Mg ²⁺	Ca ²⁺	Ba ²⁺	Al ³⁺	Fe ³⁺	Cu ²⁺	Ag ⁺	Zn ²⁺	Pb ²⁺
F ⁻	sol	sol	sol	sol	insol	insol	sl sol	sol	sl sol	sol	sol	sol	insol
Cl ⁻	sol	sol	sol	sol	sol	sol	sol	sol	sol	sol	insol	sol	sol
Br ⁻	sol	sol	sol	sol	sol	sol	sol	sol	sol	sol	insol	sol	sl sol
I ⁻	sol	sol	sol	sol	sol	sol	sol	sol			insol	sol	insol
OH ⁻	sol	sol	sol	sol	insol	sl sol	sol	insol	insol	insol		insol	insol
S ²⁻	sol	sol	sol	sol		sl sol	sol		insol	insol	insol	insol	insol
SO ₄ ²⁻	sol	sol	sol	sol	sol	sl sol	insol	sol	sol	sol	sl sol	sol	insol
CO ₃ ²⁻	sol	sol	sol	sol	insol	insol	insol			sl sol	insol	insol	insol
NO ₃ ⁻	sol	sol	sol	sol	sol	sol	sol	sol	sol	sol	sol	sol	sol
PO ₄ ³⁻	sol	insol	sol	sol	insol	insol	insol	insol	insol	insol	insol	insol	insol
CrO ₄ ²⁻	sol	sol	sol	sol	sol	sol	insol		insol	insol	insol	insol	insol
CH ₃ CO ₂ ⁻	sol	sol	sol	sol	sol	sol	sol	sl sol	sol	sol	sol	sol	sol

sol — soluble >1g/100 mL
 sl sol — slightly soluble (0.1 to 1) g/100 mL
 insol — insoluble <0.1g/100 mL
 (blank) — not enough solubility data available to be determined

FLINN
 SCIENTIFIC
 "Your Safety Source for Science"
 © 2014 Flinn Scientific, Inc. All Rights Reserved.
 AP00001

Figure 2.1 - Solubility Table

In **Figure 2.2**, there are an example of some common chemical equations that students are usually having problems in equalizing the coefficients.

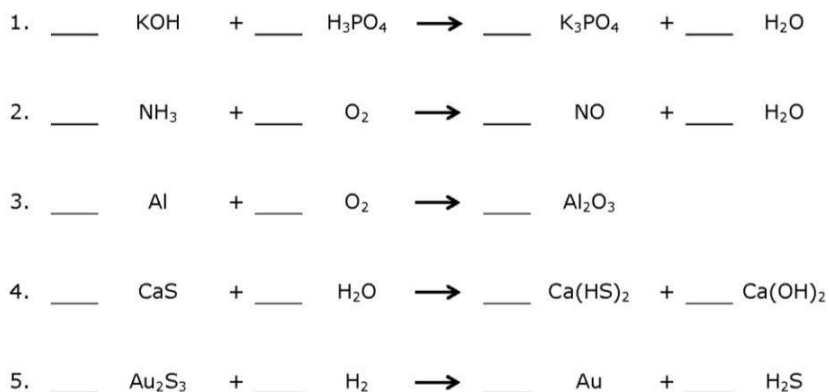


Figure 2.2 – Examples of chemical equations

After middle school, inorganic chemistry the educational process continues with the high school chemistry curriculum, which in this case represents the course of organic chemistry, which represents 99% of the entire chemistry that we are studying today. At this stage all, the students are learning the basics of organic chemistry and all classes of the organic components, for which carbon is the foundation for all substances. The organic substances deviate in carbohydrates, alcohols, saturated/unsaturated fats, proteins and their components.


Denumirea clasei	Formula generală	Zona reactivă/Grupa funcțională	Nomenclatură (sufix/prefix)
Alcani	R-H sau $\text{C}_n\text{H}_{2n+2}$	legături simple	- an
Cicloalcani	C_nH_{2n}	catenă ciclică	ciclo -
Alchene	C_nH_{2n}	o legătura dublă	- enă
Alcadiene	$\text{C}_n\text{H}_{2n-2}$	două legături duble	- dienă
Alchine	$\text{C}_n\text{H}_{2n-2}$	o legătura triplă	- ină
Arene	$\text{C}_n\text{H}_{2n-6}$ sau Ar-H	nucleu aromatic	-o (orto, 1,2), -m(meta, 1,3), p (para, 1,4) poziția substituenților urmat de benzen
Halogenoalcani	R-X	-X	halo-
Alcanoli (Alcooli)	R-OH sau $\text{C}_n\text{H}_{2n+1}\text{OH}$	-OH (hidroxil)	-ol
Polioli	R-(OH)_n		adăugarea sufixelor „diol”, „triol”, etc. la numele hidrocarburii
Fenoli	Ar-OH		
Alcoxialcani (Eteri)	R-O-R sau R-O-R'	-O- (oxi)	denumirile radiofuncționale se alcătuiesc prin citarea denumirii radicalilor R și R' în ordine alfabetică, apoi a termenului eter
Alcanali (Aldehyde)	R-CH=O sau 	-CH=O (carbonil)	-al
Alcanone (Cetone)	R-CO-R sau R-CO-R'	>C=O (carbonil)	-onă

Figure 2.3 – Organic substances classification

A small part of all of the functional groups of the organic components, which are in the school curricula, are presented in **Figure 2.3**.

During the course of organic chemistry, students are having the same problems in salvation of the chemical equations, problems, but there is one more. It will be much better if the students were able to visualize the chemical reaction while studying it.

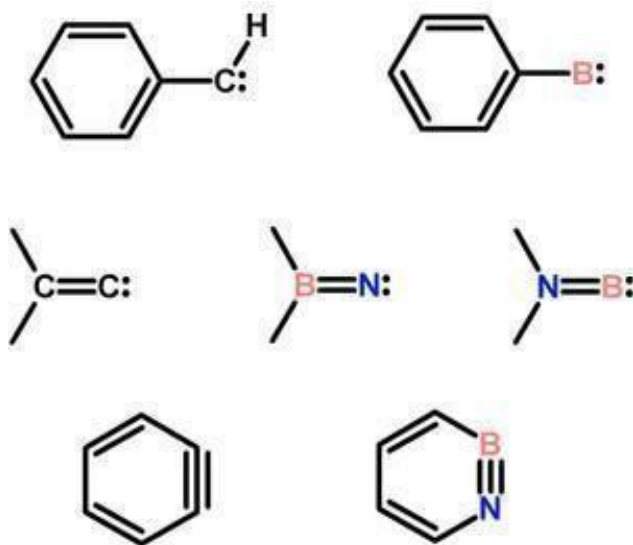


Figure 2.4 – Benzene cycle

In **Figure 2.4**, there is a representation of the benzene cycle, which represents the foundation of all the organic structures. The visualization of these structures will provide a better experience for students while studying chemistry and as a result an increasing interest for chemistry for the students which in long term perspective will provide an increasing number of the specialists in this important for the humanity domain.

2.3 Customer Validation

In order to prove the necessity of creating such a domain-specific language we conducted a survey. We have focused on our target groups – pupils and students, which is why the main respondents were from 15 to 18 years old and 19 to 24 years old.

Around 84% of the respondents told us that they enjoy/enjoyed chemistry classes during the school, but most of them encountered some problems during the classes. Most of them had difficulties remembering chemical properties (30%), solving chemical problems (28%) and solving chemical reactions (23%). Another 14% mentioned finding it hard to solve chemical equations.

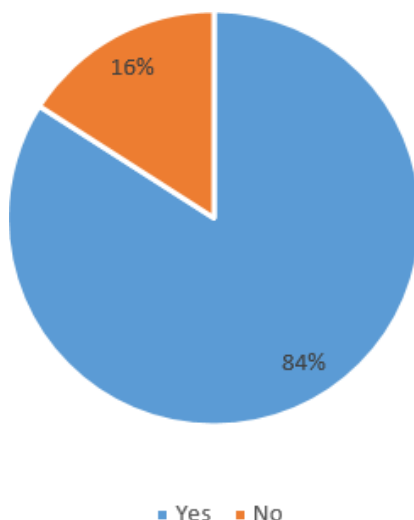


Figure 2.5 – *Did you enjoy chemistry in school?*

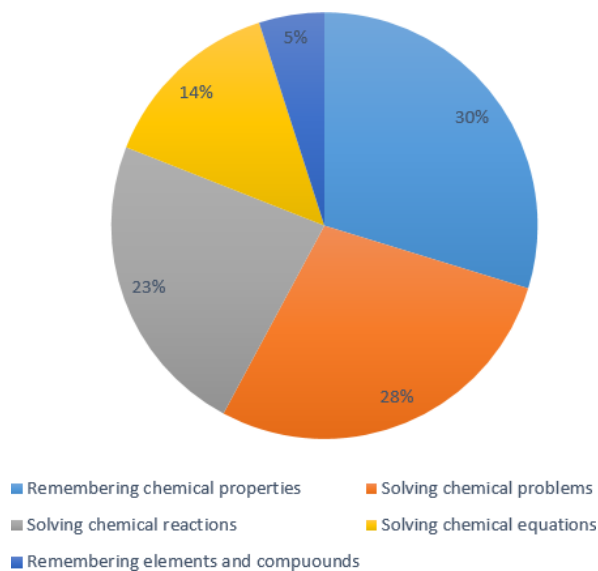


Figure 2.6 – *What did you find the most difficult during chemistry classes?*

We also asked whether people thought the chemical lessons were interactive enough and whether they find it useful the creation of a programming language that would help pupils and students solve chemical reactions and visualize them. 55% of respondents consider their chemical lessons interactive enough, and 83% of respondents think it would be useful to have a chemistry specific programming language.

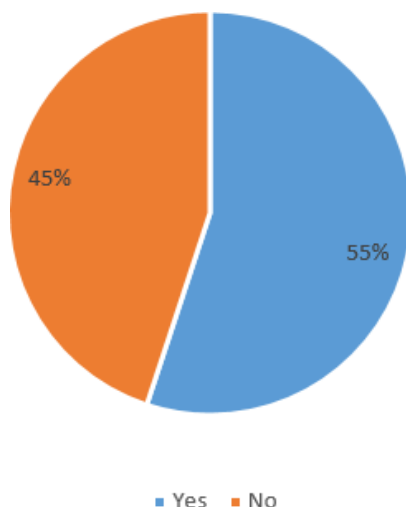


Figure 2.7 – *Do you think chemistry lessons were interactive enough?*

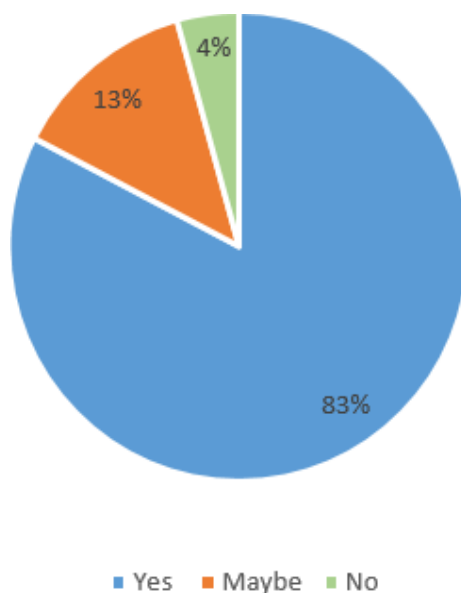


Figure 2.8 – *Would you find useful a programming language that would help pupils solve chemical reactions and visualize them?*

The results of the survey indicate the fact that pupils and students struggle to solve some of the aspects that are presented during chemistry classes and the fact that most of them would find it useful to have additional tools that would improve their understanding of the problems and concepts presented.

2.4 Existing solutions

Today one of the DSL that refers to chemistry is the Tensor Contraction Engine. The Tensor Contraction Engine (TCE) is a compiler for a domain-specific language that allows chemists to specify the computation in a high-level Mathematica-style language^[4]. It transforms tensor summation expressions to low-level code (C/Fortran) for specific hardware being mindful of memory availability, communication costs, loop fusion and ordering, etc. It is used primarily in computational

chemistry.

This DSL offers great computational advantages for the people that uses it. However, the critical problem related to this particular DSL is that it is not widely used, and for obvious reasons it will be definitely unsuitable for a casual user, for example, a student, because it has an academic realization and a person, which is not familiarized with the academic approach on chemistry, will have serious troubles using it. Therefore, there is still a necessity for a widely accessible and simple to use DSL that will help the students to learn chemistry more efficient, with which they will not have any troubles.

In conclusion, the project we are intending to realize is needed on the market and will have a great user base, since the potential target audience - students will have a great benefit from this DSL.

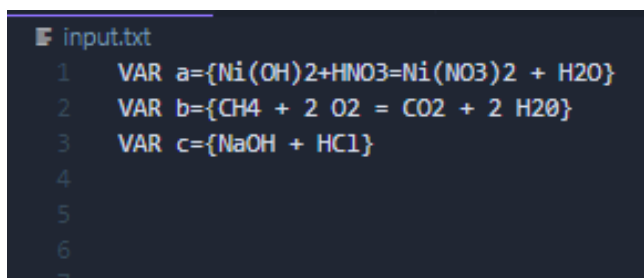
3. DSL specifications

In the following sections, we have described the specifications of our domain-specific language aimed for solving chemistry reactions. To be mentioned that in this DSL, string is the only one main data type, due to the fact that a chemical equation consists of numbers, letters and symbols.

The DSL has very simple scope rules:

- a variable used in future calculations must be declared first;
- variables may be declared at the beginning of the program/everywhere in the program;
- only predefined methods can be used;

The symbol for the assignment is “=” and it is used for declaring variables following the rule from the grammar of the domain specified language: `<varName> = { <reaction> }` or `<varName> = { <partialReaction> }`. An example of assignment can be seen in **Figure 3.1**.



```
input.txt
1  VAR a={Ni(OH)2+HNO3=Ni(NO3)2 + H2O}
2  VAR b={CH4 + 2 O2 = CO2 + 2 H2O}
3  VAR c={NaOH + HCl}
4
5
6
7
```

Figure 3.1 – Examples of assignments

3.1 Semantic rules

This set of rules place additional constraints on the set of valid programs besides the constraints implied by the grammar. The compiler will explicitly check each of these rules and as well as other languages, our DSL will use the console to communicate with the user, therefore all the errors will be specified in the console or by some error fields within the code where it occurred.

A program in the domain-specified language will consists of chemical equations. Each equation has at least one side, that if formed from molecules delimited by the equal sign. If the chemical reaction is balanced, in front of the molecule can be positioned a number (the coefficient). A molecule is a group of two or more elements; also, it represents a structure that contains information about elements from the periodic table followed by a number, which is optional. Respectively, an element has to consist of at least an upper case letter, that can be followed of a lower case letter, because in the periodic table, there are two types of symbols: uppercase letter or uppercase and lowercase letters.

3.2 Grammar rules

For a better understanding of the grammar are used special notations. (Table 3.1. Meta-notations)

Table 3.1 - Meta-notations

<foo>	means that foo is a nonterminal
foo	(in bold font) means that foo is a terminal i.e., a token or a part of a token
x?	means zero or one occurrence of x
x*	means zero or more occurrences of x
x ⁺	means one or more occurrences of x
	separates alternatives

The DSL design includes several stages. First of all, definition of the programming language grammar $L(G) = (S, P, V_N, V_T)$:

- S – is a start symbol;
- P – is a finite set of production of rules;
- V_N – is a finite set of non-terminal symbols;
- V_T – is a finite set of terminal symbols^[5].

$V_N = \{ \langle \text{commandStatementList} \rangle, \langle \text{commandStatement} \rangle, \langle \text{printCommand} \rangle, \langle \text{balanceCommand} \rangle, \langle \text{solveCommand} \rangle, \langle \text{showCommand} \rangle, \langle \text{varCommand} \rangle, \langle \text{declReaction} \rangle, \langle \text{declPartial} \rangle, \langle \text{varName} \rangle, \langle \text{EQUAL} \rangle, \langle \text{PLUS} \rangle, \langle \text{reaction} \rangle, \langle \text{partialReaction} \rangle, \langle \text{side} \rangle, \langle \text{molecule} \rangle, \langle \text{element} \rangle, \langle \text{DIGIT} \rangle, \langle \text{DIGITS} \rangle, \langle \text{LOWERCASE} \rangle, \langle \text{UPPERCASE} \rangle, \langle \text{number} \rangle \}$

$V_T = \{ \text{PRINT, BALANCE, SOLVE, SHOW, VAR, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \{, \}, (,), +, =} \}$

$S = \{ \langle \text{program} \rangle \}$

$P = \{ \langle \text{program} \rangle \rightarrow \langle \text{commandStatementList} \rangle$

$\langle \text{commandStatementList} \rangle \rightarrow \langle \text{commandStatement} \rangle^+$

$\langle \text{commandStatement} \rangle \rightarrow \langle \text{printCommand} \rangle \mid \langle \text{balanceCommand} \rangle \mid \langle \text{solveCommand} \rangle \mid \langle \text{showCommand} \rangle \mid \langle \text{varCommand} \rangle$

$\langle \text{printCommand} \rangle \rightarrow \text{PRINT} (\langle \text{varName} \rangle)$

$\langle \text{balanceCommand} \rangle \rightarrow \text{BALANCE} (\langle \text{varName} \rangle)$

$\langle \text{solveCommand} \rangle \rightarrow \text{SOLVE}(\langle \text{varName} \rangle)$

$\langle \text{showCommand} \rangle \rightarrow \text{SHOW}(\langle \text{varName} \rangle)$

$\langle \text{varCommand} \rangle \rightarrow \langle \text{declReaction} \rangle \mid \langle \text{declPartial} \rangle$

$\langle \text{declReaction} \rangle \rightarrow \text{VAR} \langle \text{varName} \rangle \langle \text{EQUAL} \rangle \{ \langle \text{reaction} \rangle \}$

$\langle \text{declPartial} \rangle \rightarrow \text{VAR} \langle \text{varName} \rangle \langle \text{EQUAL} \rangle \{ \langle \text{partialReaction} \rangle \}$

$\langle \text{varName} \rangle \rightarrow \langle \text{LOWERCASE} \rangle^+$

$\langle \text{reaction} \rangle \rightarrow \langle \text{side} \rangle \langle \text{EQUAL} \rangle \langle \text{side} \rangle$

$\langle \text{partialReaction} \rangle \rightarrow \langle \text{side} \rangle$

$\langle \text{side} \rangle \rightarrow \langle \text{side} \rangle \langle \text{PLUS} \rangle \langle \text{side} \rangle \mid \langle \text{number} \rangle? \langle \text{molecule} \rangle$

$\langle \text{molecule} \rangle \rightarrow (\langle \text{element} \rangle \langle \text{number} \rangle?)^+ \mid \langle \text{molecule} \rangle (\langle \text{molecule} \rangle) \langle \text{number} \rangle$

$\langle \text{element} \rangle \rightarrow \langle \text{UPPERCASE} \rangle \langle \text{LOWERCASE} \rangle?$

$\langle \text{number} \rangle \rightarrow \langle \text{DIGIT} \rangle \langle \text{DIGITS} \rangle^*$

<UPPERCASE> → A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
 <LOWERCASE> → a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
 <DIGIT> → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 <DIGITS> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 <EQUAL> → =
 <PLUS> → +

Table 3.2 - Example of code

Input	Output
VAR a = {Ni(OH)2+HNO3=Ni(NO3)2 + H2O} BALANCE(a)	Ni(OH) ₂ + 2 HNO ₃ → Ni(NO ₃) ₂ + 2 H ₂ O
VAR c={NaOH + HCl} SOLVE(c)	NaOH + HCl → NaCl + H ₂ O

The rule to start a language grammar is engage the parser first. Any rule in grammar can act as a start rule for the following parser. Start rules do not necessarily consume all of the input, because they consume only as much input as needed to match an alternative of the rule.

3.3 Method invocation

In order to verify if the grammar was written correctly and the invoked methods will show the needed result, ANTLR was used. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. From a grammar, ANTLR generates a parser that can build and walk parse tree [6]. In **Figure 3.2** and **Figure 3.3** is presented the graphical form of the parse tree for the inputs that were specified in Table 3.2, which has <program> as start symbol and then it derives in <commandStatementList> and so on.

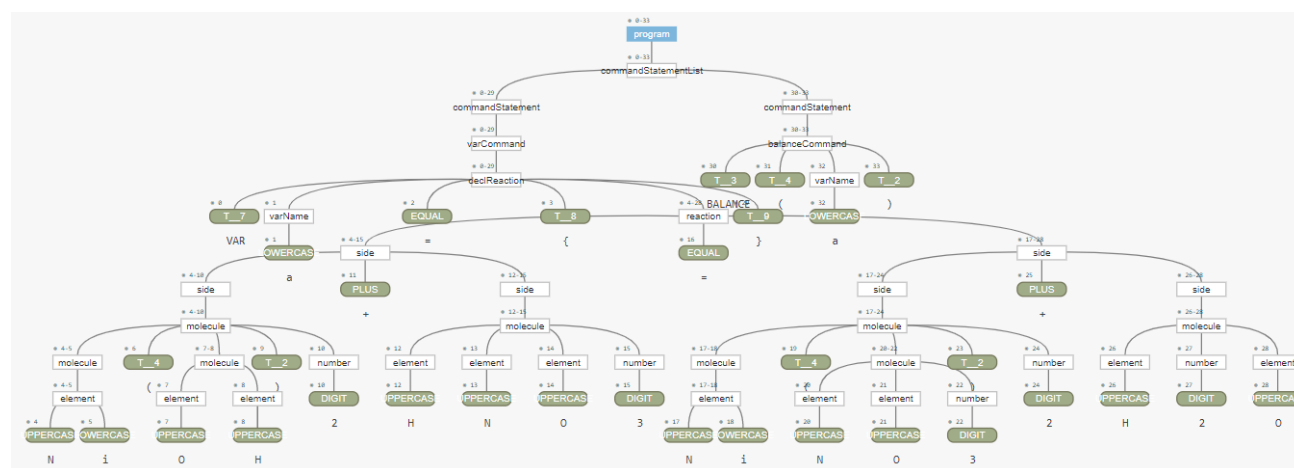


Figure 3.2 - The parse tree of the first input from Table 3.2

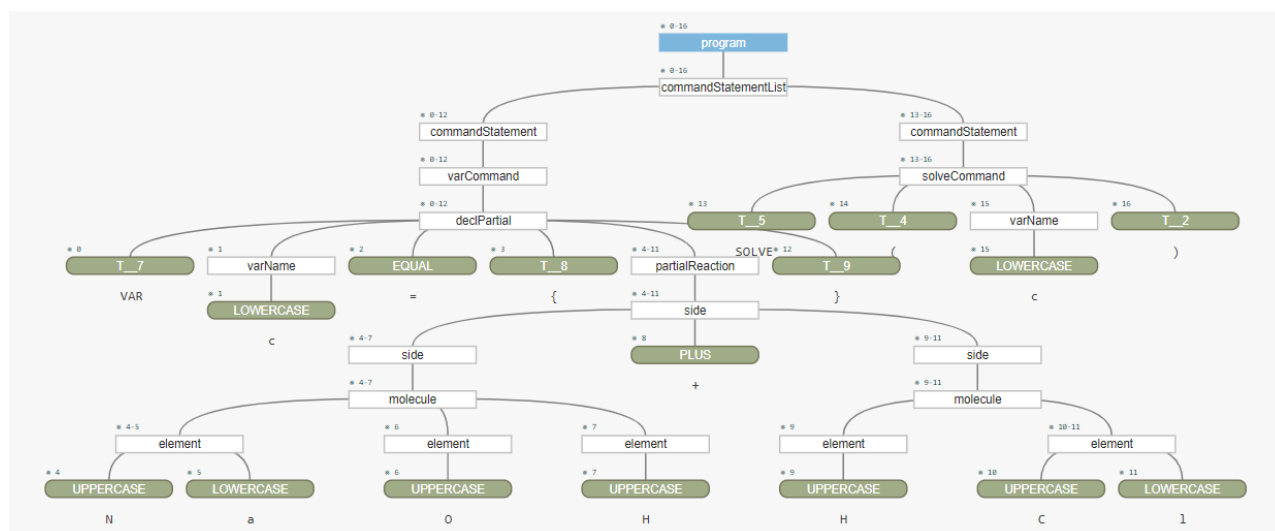


Figure 3.3 - The parse tree of the second input from Table 3.2

4. Implementation

Every interpreter is built to interpret a specific programming language. That is how to “implement” a programming language. Without a compiler or an interpreter a programming language is nothing more than an idea or a specification.

The interpreter we built, implements all features from previous chapter. It tokenize and parse source code in a REPL, building up an internal representation of the code called abstract syntax tree and then evaluate this tree.

It will have a few major parts:

1. the lexer;
2. the parser;
3. the Abstract Syntax Tree (AST);

4.1 Go language

Go, or *Golang*, is an open source programming language. It is statically typed and produces compiled machine code binaries. Developers say that Google's Go language is the C for the twenty-first century when it comes to syntax. However, this new programming language includes tooling that allows you to safely use memory, manage objects, collect garbage, and provide static (or strict) typing along with concurrency.^[7]

We have selected namely this programming language because of several reasons.

Go provides great tooling. With Go's universal formatting style thanks to *gofmt*, we were concentrated on our interpreter and did not worry about 3rd party libraries, tools and dependencies. We have not used any other tools in this project other than the ones provided by the Go programming language.

Go code maps closely to other more low-level languages, like C, C++ and Rust. Maybe the reason for this is Go itself, with its focus on simplicity, its stripped-down charm and lack of programming language constructs that are absent in other languages and hard to translate.

4.2 Lexing

In the following section are presented the lexical aspects of our project like: lexical analysis, tokens and obviously the lexer. All the information is supported with code snippets, for a better understanding.

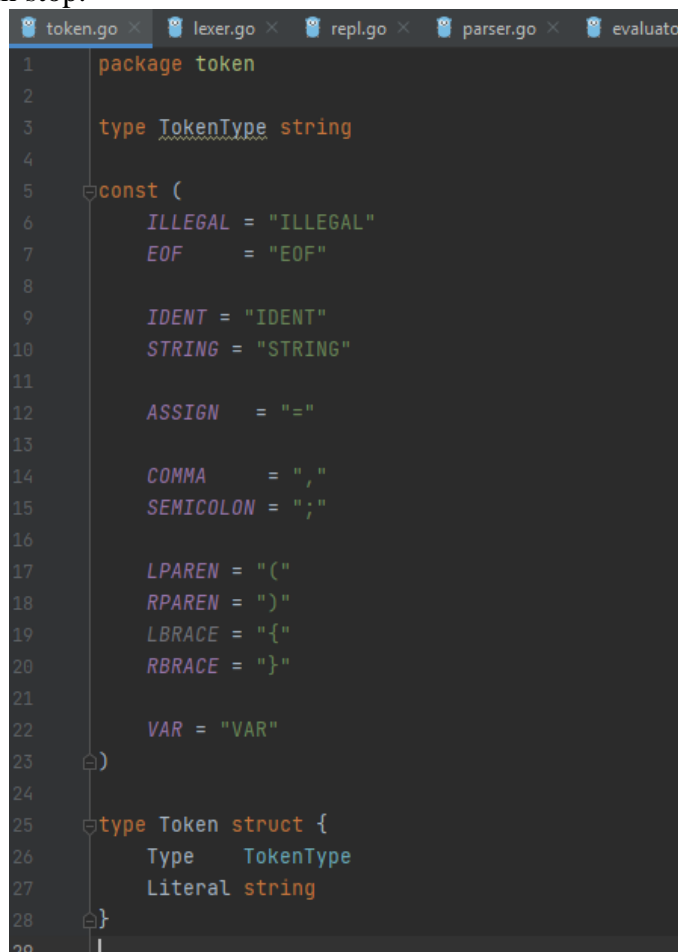
4.2.1 Lexical analysis

The first transformation, from source code to tokens, is called “lexical analysis”, or “lexing” for short. It’s done by a lexer. Tokens itself are small, easily categorizable data structures that are then fed to the parser, which does the second transformation and turns the tokens into an “Abstract Syntax Tree”.

4.2.2 Tokens

We defined the *TokenType* type to be a *string*. That allows us to use many different values as *TokenTypes*, which in turn allows us to distinguish between different types of tokens. Using *string* also has the advantage of being easy to debug without a lot of boilerplate and helper functions: we can just print a *string*.

As it can be seen in **Figure 4.1**, there are two special types: *ILLEGAL* and *EOF*. *ILLEGAL* signifies a token/character we don’t know about and *EOF* stands for “end of file”, which tells our parser later on that it can stop.



```
1 package token
2
3 type TokenType string
4
5 const (
6     ILLEGAL = "ILLEGAL"
7     EOF    = "EOF"
8
9     IDENT = "IDENT"
10    STRING = "STRING"
11
12    ASSIGN = "="
13
14    COMMA    = ","
15    SEMICOLON = ";"
16
17    LPAREN = "("
18    RPAREN = ")"
19    LBRACE = "{"
20    RBRACE = "}"
21
22    VAR = "VAR"
23 )
24
25 type Token struct {
26     Type    TokenType
27     Literal string
28 }
29
```

Figure 4.1 – Tokens

4.2.3 *Lexer*

Lexer takes source code as input and output the tokens that represent the source code. It goes through its input and outputs the next token it recognizes. It does not need to buffer or save tokens, since there is only one method which outputs the next token. That means, we initialize the lexer with our source code and then repeatedly go through the source code, token by token, character by character.

```
1  package lexer
2
3  import "chemistry/token"
4
5  type Lexer struct {
6      input      string
7      position   int
8      readPosition int
9      ch         byte
10 }
11
12 func New(input string) *Lexer {
13     l := &Lexer{input: input}
14     l.readChar()
15     return l
16 }
```

Figure 4.2 – *lexer struct and definition*

```
18 func (l *Lexer) NextToken() token.Token {
19     var tok token.Token
20
21     l.skipWhitespace()
22
23     switch l.ch {
24     case '=':
25         tok = newToken(token.ASSIGN, l.ch)
26     case ';':
27         tok = newToken(token.SEMICOLON, l.ch)
28     case ',':
29         tok = newToken(token.COMMA, l.ch)
30         tok = newToken(token.RBRACE, l.ch)
31     case '(':
32         tok = newToken(token.LPAREN, l.ch)
33     case ')':
34         tok = newToken(token.RPAREN, l.ch)
35     case '{':
36         tok.Type = token.STRING
37         tok.Literal = l.readString()
38     case 0:
39         tok.Literal = ""
40         tok.Type = token.EOF
41     default:
42         if isLetter(l.ch) {
43             tok.Literal = l.readIdentifier()
44             tok.Type = token.LookupIdent(tok.Literal)
45             return tok
46         } else {
47             tok = newToken(token.ILLEGAL, l.ch)
48         }
49     }
50
51     l.readChar()
52     return tok
53 }
```

Figure 4.3 – *Main lexer method for loop*


```

55 func (l *Lexer) skipWhitespace() {
56     for l.ch == ' ' || l.ch == '\t' || l.ch == '\n' || l.ch == '\r' {
57         l.readChar()
58     }
59 }
60
61 func (l *Lexer) readChar() {
62     if l.readPosition >= len(l.input) {
63         l.ch = 0
64     } else {
65         l.ch = l.input[l.readPosition]
66     }
67     l.position = l.readPosition
68     l.readPosition += 1
69 }
70
71 func (l *Lexer) peekChar() byte {
72     if l.readPosition >= len(l.input) {
73         return 0
74     } else {
75         return l.input[l.readPosition]
76     }
77 }
78
79 func (l *Lexer) readIdentifier() string {
80     position := l.position
81     for isLetter(l.ch) {
82         l.readChar()
83     }
84     return l.input[position:l.position]
85 }
86
87 func (l *Lexer) readNumber() string {
88     position := l.position
89     for isDigit(l.ch) {
90         l.readChar()
91     }
92     return l.input[position:l.position]
93 }

```

Figure 4.4 – *Lexer additional methods I*

```

95 func (l *Lexer) readString() string {
96     position := l.position + 1
97     for {
98         l.readChar()
99         if l.ch == '"' || l.ch == 0 {
100             break
101         }
102     }
103     return l.input[position:l.position]
104 }
105
106 func isLetter(ch byte) bool {
107     return 'a' <= ch && ch <= 'z' || 'A' <= ch && ch <= 'Z' || ch == '_'
108 }
109
110 func isDigit(ch byte) bool {
111     return '0' <= ch && ch <= '9'
112 }
113
114 func newToken(tokenType token.TokenType, ch byte) token.Token {
115     return token.Token{Type: tokenType, Literal: string(ch)}
116 }
117

```

Figure 4.5 – *Lexer additional methods II*

4.3 Parsing

In this section, it will be presented the parsing aspects of our chemistry DSL, including the Abstract Syntax Tree and parsers. Like in the previous section, we have presented code snippets for a better understanding on the implementation process.

4.3.1 Abstract Syntax Tree

We have three interfaces called *Node*, *Statement* and *Expression*. Every node in our AST has to implement the *Node* interface. The constructed AST consists solely of *Nodes* that are connected to each other – it is a tree, after all. Some of these nodes implement the *Statement* and some the *Expression* interface. They are not strictly necessary but they help us by guiding the *Go* compiler and possibly causing it to throw errors when we use a *Statement* where an *Expression* should have been used.

```
type Node interface {
    TokenLiteral() string
    String() string
}

type Statement interface {
    Node
    statementNode()
}

type Expression interface {
    Node
    expressionNode()
}
```

Figure 4.6 – AST construction

Program node is going to be the root node of every AST our parser produces. Every valid program is a series of statements. These statements are contained in the *Program*. *Statements*, which is just a slice of AST nodes that implement the *Statement* interface.

```
24 type Program struct {
25     Statements []Statement
26 }
27
28 func (p *Program) TokenLiteral() string {
29     if len(p.Statements) > 0 {
30         return p.Statements[0].TokenLiteral()
31     } else {
32         return ""
33     }
34 }
35
36 func (p *Program) String() string {
37     var out bytes.Buffer
38
39     for _, s := range p.Statements {
40         out.WriteString(s.String())
41     }
42
43     return out.String()
44 }
```

Figure 4.7 – Root node

VarStatement has the fields we need: *Name* to hold the identifier of the binding and *Value* for the expression that produces the value. The two methods *statementNode* and *TokenLiteral* satisfy the *Statement* and *Node* interfaces respectively.

```

45
46  type VarStatement struct {
47      Token token.Token
48      Name *Identifier
49      Value Expression
50  }
51
52  func (ls *VarStatement) statementNode() {}
53  func (ls *VarStatement) TokenLiteral() string { return ls.Token.Literal }
54  func (ls *VarStatement) String() string {
55      var out bytes.Buffer
56
57      out.WriteString(ls.TokenLiteral() + " ")
58      out.WriteString(ls.Name.String())
59      out.WriteString(" = ")
60
61      if ls.Value != nil {
62          out.WriteString(ls.Value.String())
63      }
64
65      out.WriteString(" ;")
66
67      return out.String()
68  }

```

Figure 4.8 – *Var statement content*

```

83
84  type Identifier struct {
85      Token token.Token
86      Value string
87  }
88
89  func (i *Identifier) expressionNode() {}
90  func (i *Identifier) TokenLiteral() string { return i.Token.Literal }
91  func (i *Identifier) String() string { return i.Value }
92
93  type PrefixExpression struct {
94      Token token.Token
95      Operator string
96      Right Expression
97  }
98

```

Figure 4.9 – *Identifier content*

4.3.2 Parsers

A parser is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process. The parser is often preceded by a separate lexical analyser, which creates tokens from the sequence of input characters.

Our parser is a recursive descent parser and in particular, it is a “top-down operator precedence” parser.

```
36 func New(l *lexer.Lexer) *Parser {
37     p := &Parser{
38         l:      l,
39         errors: []string{},
40     }
41
42     p.prefixParseFns = make(map[token.TokenType]prefixParseFn)
43     p.registerPrefix(token.IDENT, p.parseIdentifier)
44     p.registerPrefix(token.STRING, p.parseStringLiteral)
45
46     p.registerPrefix(token.LPAREN, p.parseGroupedExpression)
47
48     p.infixParseFns = make(map[token.TokenType]infixParseFn)
49
50     p.registerInfix(token.LPAREN, p.parseCallExpression)
51
52     p.nextToken()
53     p.nextToken()
54
55     return p
56 }
```

Figure 4.10 – Parser initialization

```
95
96 func (p *Parser) ParseProgram() *ast.Program {
97     program := &ast.Program{}
98     program.Statements = []ast.Statement{}
99
100    for !p.curTokenIs(token.EOF) {
101        stmt := p.parseStatement()
102        if stmt != nil {
103            program.Statements = append(program.Statements, stmt)
104        }
105        p.nextToken()
106    }
107
108    return program
109 }
```

Figure 4.11 – Public method to start parsing

```

10
11 func (p *Parser) parseStatement() ast.Statement {
12     switch p.curToken.Type {
13     case token.VAR:
14         return p.parseVarStatement()
15     default:
16         return p.parseExpressionStatement()
17     }
18 }
19
20 func (p *Parser) parseVarStatement() *ast.VarStatement {
21     stmt := &ast.VarStatement{Token: p.curToken}
22
23     if !p.expectPeek(token.IDENT) : nil
24
25     stmt.Name = &ast.Identifier{Token: p.curToken, Value: p.curToken.Literal}
26
27     if !p.expectPeek(token.ASSIGN) : nil
28
29     p.nextToken()
30
31     stmt.Value = p.parseExpression(LOWEST)
32
33     if p.peekTokenIs(token.SEMICOLON) {
34         p.nextToken()
35     }
36
37     return stmt
38 }
39
40
41
42

```

Figure 4.12 – Parse VAR statement

```

43
44 func (p *Parser) parseExpressionStatement() *ast.ExpressionStatement {
45     stmt := &ast.ExpressionStatement{Token: p.curToken}
46
47     stmt.Expression = p.parseExpression(LOWEST)
48
49     if p.peekTokenIs(token.SEMICOLON) {
50         p.nextToken()
51     }
52
53     return stmt
54 }
55
56 func (p *Parser) parseExpression(precedence int) ast.Expression {
57     prefix := p.prefixParseFns[p.curToken.Type]
58     if prefix == nil {
59         p.noPrefixParseFnError(p.curToken.Type)
60         return nil
61     }
62     leftExp := prefix()
63
64     for !p.peekTokenIs(token.SEMICOLON) && precedence < p.peekPrecedence() {
65         infix := p.infixParseFns[p.peekToken.Type]
66         if infix == nil : leftExp
67
68         p.nextToken()
69
70         leftExp = infix(leftExp)
71     }
72
73     return leftExp
74 }
75
76
77

```

Figure 4.13 – Parse default expressions

4.3.3 Pratt parser

A Pratt parser's main idea is the association of parsing functions (which Pratt calls “semantic code”) with token types. Whenever this token type is encountered, the parsing functions are called to parse the appropriate expression and return an AST node that represents it. Each token type can have up to two parsing functions associated with it, depending on whether the token is found in a prefix or an infix position.

```
20 type (  
21     prefixParseFn func() ast.Expression  
22     infixParseFn func(ast.Expression) ast.Expression  
23 )
```

Figure 4.14 – Pratt parser helper types

```
25 type Parser struct {  
26     l *lexer.Lexer  
27     errors []string  
28  
29     curToken token.Token  
30     peekToken token.Token  
31  
32     prefixParseFns map[token.TokenType]prefixParseFn  
33     infixParseFns map[token.TokenType]infixParseFn  
34 }
```

Figure 4.15 – Parser struct to contain data

```
269 func (p *Parser) registerPrefix(tokenType token.TokenType, fn prefixParseFn) {  
270     p.prefixParseFns[tokenType] = fn  
271 }  
272  
273 func (p *Parser) registerInfix(tokenType token.TokenType, fn infixParseFn) {  
274     p.infixParseFns[tokenType] = fn  
275 }
```

Figure 4.16 – Methods to register Prefix and Infix expressions

4.3.4 Call expression

Call expressions consist of an expression that results in a function when evaluated and a list of expressions that are the arguments to this function call.

```
132  
133 type CallExpression struct {  
134     Token token.Token  
135     Function Expression  
136     Arguments []Expression  
137 }
```

Figure 4.17 – CallExpression struct

```

139 func (ce *CallExpression) expressionNode() {}
140 func (ce *CallExpression) TokenLiteral() string { return ce.Token.Literal }
141 func (ce *CallExpression) String() string {
142     var out bytes.Buffer
143
144     args := []string{}
145     for _, a := range ce.Arguments {
146         args = append(args, a.String())
147     }
148
149     out.WriteString(ce.Function.String())
150     out.WriteString(" (")
151     out.WriteString(strings.Join(args, " "))
152     out.WriteString(")")
153
154     return out.String()
155 }

```

Figure 4.18 – *CallExpression methods*

4.4 Chemical Reaction

This chapter is more specific to the domain of study, because it contains theoretical information about chemical reactions, how to balance and solve them. In addition, there will be code snippets with explanation about the algorithms.

4.4.1 Types of chemical reactions

In the following section, we will present the definitions of some common types of chemical reactions. In order to make the topic more understandable, there will be some examples of each type.

The process characterized by a chemical change in which the starting materials, called *reactants*, are different from the products represents a *chemical reaction*. They tend to involve the motion of electrons, leading to the formation and breaking of chemical bonds. There are several different types of chemical reactions and more than one way of classifying them.^[8] Here are some common reaction types:

- **Direct Combination or Synthesis Reaction**

In a *synthesis reaction*, two or more chemical species combine to form a more complex product. It has the following form: $A + B \rightarrow AB$.

The combination of iron and sulfur to form iron (II) sulfide is an example of synthesis reaction:
 $8 \text{ Fe} + \text{S}_8 \rightarrow 8 \text{ FeS}$

- **Chemical Decomposition or Analysis Reaction**

In a *decomposition reaction*, a compound is broken into smaller chemical species. This type of reaction has the following form: $AB \rightarrow A + B$.

The electrolysis of water into oxygen and hydrogen gas is an example of a decomposition reaction: $2 \text{ H}_2\text{O} \rightarrow 2 \text{ H}_2\uparrow + \text{O}_2\uparrow$

- **Single Displacement or Substitution Reaction**

A substitution or *single displacement reaction* is characterized by one element being displaced from a compound by another element. The form is: $A + BC \rightarrow AC + B$.

An example of substitution reaction occurs when zinc combine with hydrochloric acid. The zinc replaces the hydrogen: $\text{Zn} + 2 \text{ HCl} \rightarrow \text{ZnCl}_2 + \text{H}_2$.

- **Metathesis or Double Displacement Reaction**

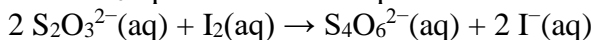
In a double displacement or *metathesis reaction*, two compounds exchange bonds or ions in order to form different compounds. It has the following form: $AB + CD \rightarrow AD + CB$.

An example of a double displacement reaction occurs between sodium chloride and silver nitrate to form sodium nitrate and silver chloride: $NaCl(aq) + AgNO_3(aq) \rightarrow NaNO_3 + AgCl(s)$.

- **Oxidation-Reduction or Redox Reaction**

In a *redox reaction*, the oxidation numbers of atoms are changed. Redox reactions may involve the transfer of electrons between chemical species.

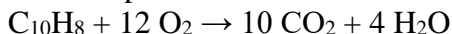
The reaction that occurs when in which I_2 is reduced to I^- and $S_2O_3^{2-}$ (thiosulfate anion) is oxidized to $S_4O_6^{2-}$ provides an example of a redox reaction:



- **Combustion**

A *combustion* reaction is a type of redox reaction in which a combustible material combines with an oxidizer to form oxidized products and generate heat (exothermic reaction). Usually, in a combustion reaction oxygen combines with another compound to form carbon dioxide and water.

An example of a combustion reaction is the burning of naphthalene:



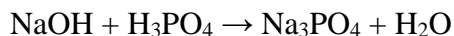
4.4.2 Balancing chemical reactions

A chemical equation is said to be balanced, provided that the number of atoms of each type on the left is same as the number of atoms of the corresponding type on the right. This leads to the concept of stoichiometry which is defined as the quantitative relationship between reactants and products in a chemical equation is. In other words, stoichiometry is the proportional relationship between two or more substances during a chemical reaction. The ratio of moles of reactants and products is given by the coefficients in a balanced chemical equation. It is through this that the amount of reactant needed to produce a given quantity of product, or how much of a product is formed from a given quantity of reactant is determined.^[9]

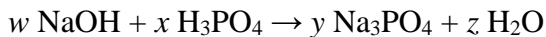
Linear algebra at present is of growing importance in engineering research, science, frameworks, electrical networks, traffic flow, economics, statistics, technologies, and many others. It forms a foundation of numeric methods and its main instrument is matrices that can hold enormous amounts of data in a form readily accessible by the computer. In this section will be presented how to construct a homogenous system of equations whose solution provides appropriate value to balance the atoms in the reactants using matrix algebra.

Below, we will show an example of how linear algebra method can be applied on chemical equation, in order to balance it using matrix algebra and Gaussian elimination method.

Let us take an example:



In order to balance this equation, we insert unknowns, multiplying the reactants and the products to get an equation of the form:



Next, we compare the number of sodium (Na), oxygen (O), hydrogen (H) and phosphor (P) atoms of the reactants with the number of the products and we obtain four linear equations:

$$Na: w = 3y$$

$$H: w + 3x = 2z$$

$$O: w + 4x = 4y + z$$

$$P: x = y$$

Now, we will rewrite these equations in standard form and we will be able to see that we have a homogeneous linear system in four unknowns, that is, w , x , y and z .

$$w + 0x - 3y + 0z = 0$$

$$w + 4x - 4y - z = 0$$

$$w + 3x + 0y - 2z = 0$$

$$0w + x - y + 0z = 0$$

Alternatively;

$$w - 3y = 0$$

$$w + 4x - 4y - z = 0$$

$$w + 3x - 2z = 0$$

$$x - y = 0$$

Writing these equations in matrix form, we have the following augmented matrix:

$$\left(\begin{array}{cccc|c} 1 & 0 & -3 & 0 & 0 \\ 1 & 4 & -4 & -1 & 0 \\ 1 & 3 & 0 & -2 & 0 \\ 0 & 1 & -1 & 0 & 0 \end{array}\right)$$

After applying the Gaussian elimination, we obtain:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1/3 & 0 \\ 0 & 0 & 1 & -1/3 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

$$w = z$$

$$x = 1/3 z$$

$$y = 1/3 z$$

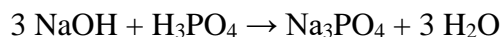
Hence, since z can be chosen arbitrary and we are dealing with atoms, it is convenient to choose values so that all the unknowns are positive integers. One of such choice is $z = 3$, which yields to the following:

$$w = 3$$

$$x = 1$$

$$y = 1$$

In this case, our balanced equation is:



Now, let us see how the implementation of the algorithm described above looks like. The parser of our Domain Specific Language extracts the chemical equation as a string, therefore the function `balance()` takes as parameter a string and return also a string – the balanced equation.

First, we have to split the equation in two sides and then to separate the substances. This was done by using `strings.Split()` function from Golang and it can be seen in **Figure 4.19**.

```

124 func balance(equation string) string{
125     equation = strings.ReplaceAll(equation, old: " ", new: "")
126     sides := strings.Split(equation, sep: "=")
127     leftSide := strings.Split(sides[0], sep: "+")
128     rightSide := strings.Split(sides[1], sep: "+")

```

Figure 4.19 – *Splitting the equation*

Next, we will need two maps with string keys, that will contain the chemical element and the value will be of type int, in order to hold the number of atoms of the element. The another two maps with string keys and bool values, will contain the information about chemical element that was parsed.

```

130     var leftCompounds []map[string]int
131     var rightCompounds []map[string]int
132     s1 := make(map[string]bool)
133     s2 := make(map[string]bool)
134     for i := 0; i < len(leftSide); i++){
135         leftCompounds = append(leftCompounds, parse(leftSide[i]))
136         for k := range (parse(leftSide[i])){
137             s1[k] = true
138         }
139     }
140     for i := 0; i < len(rightSide); i++){
141         rightCompounds = append(rightCompounds, parse(rightSide[i]))
142         for k := range (parse(rightSide[i])){
143             s2[k] = true
144         }
145     }

```

Figure 4.20 – *Creating and assigning values to the dictionaries*

The function parse, takes as input a string, in our case, the chemical substance, for examples: H_3PO_4 and returns a map[string] int. In order to make it more understandable, for H_3PO_4 , it will give us the dictionary: { "H" : 3, "P" : 1, "O" : 4}. In **Figure 4.21** is shown the function that parses the substance and returns a dictionary of element and its number of atoms.

```

270 func parse(formula string) map[string]int {
271     var maps = make(map[string]int)
272     var multiple []int
273     str, count := "", ""
274     for i := len(formula) - 1; i >= 0; i-- {
275         char := formula[i]
276         if char >= 48 && char <= 57 {
277             count = string(char) + count
278         } else {
279             if char == 41 {
280                 atoi, _ := strconv.Atoi(count)
281                 if len(multiple) > 0 {
282                     atoi = atoi * multiple[len(multiple)-1]
283                 }
284                 multiple = append(multiple, atoi)
285                 count = ""
286             } else if char == 40 {
287                 multiple = multiple[:len(multiple)-1]
288             } else if char >= 97 && char <= 122 {
289                 str = string(char) + str
290             } else if char >= 65 && char <= 90 {
291                 str = string(char) + str
292                 nums := 1
293                 if count == "" {
294                     count = "1"
295                 }
296                 atoi, _ := strconv.Atoi(count)
297                 if len(multiple) > 0 {
298                     nums = multiple[len(multiple)-1] * atoi
299                 } else {
300                     nums = atoi
301                 }
302                 maps[str] += nums
303                 str = ""
304                 count = ""
305             }
306         }
307     }
308     return maps
309 }
310

```

Figure 4.21 – *parse function*

For a better and simpler working process, we will sort the elements in ascending alphabetical order from the left side of the equation, i.e., for the equation:

$\text{H}_3\text{PO}_4 + \text{NaOH} = \text{Na}_3\text{PO}_4 + \text{H}_2\text{O}$, we will have the lists of maps:
`leftCompounds=[{"H": 3, "P": 1, "O": 4}, {"Na": 1, "O": 1, "H": 1}]`
`rightCompounds=[{"Na": 3, "P": 1, "O": 4}, {"H": 2, "O": 1}]`
`elements = ["H", "P", "O", "Na"]` and after sorting, elements becomes `["H", "Na", "O", "P"]`

The next map, called *indexes*, stores the chemical element and its index. For example: `{"H": 0, "Na": 1, "O": 2, "P": 3}`.

Now, it is the time for linear algebra. In order to create the augmented matrix, we have to know first how many rows and columns will have the matrix, because it will vary for every equation.

```

155         numCols := len(leftCompounds) + len(rightCompounds)
156         numRows := len(elements)
157
158         arr := make([][]int64, numRows)
159         for i := 0; i < numRows; i++){
160             arr[i] = make([]int64, numCols)
161         }

```

Figure 4.22 – creating the augmented matrix

The following step is to fulfill the matrix with the number of atoms for every elements, taking in the consideration that each row stores the data about a specific element. For example, row 0 stores the number of atoms of hydrogen in every substance (column), but row 2 about oxygen.

```

Adding Left Compounds
1 3 0 0
1 0 0 0
1 4 0 0
0 1 0 0
Adding Right Compounds
1 3 0 -2
1 0 -3 0
1 4 -4 -1
0 1 -1 0

```

Figure 4.23 – The Augmented matrix

At this point, remained to solve this system of linear equations and because we use the matrix to save the coefficients from each equation, we will need to do Gauss elimination and to get 4 coefficients of our chemical reaction. For this operation exists a repository on GitHub called **gomath**, which includes the packages *rational* and *gaussian*.

```

import (
    "fmt"
    "github.com/alex-ant/gomath/gaussian-elimination"
    "github.com/alex-ant/gomath/rational"

```

Figure 4.24 – Importing rational and gaussian

Now, we will use the new data type Rational and we will convert our augmented matrix in the form from **Figure 4.25**.

```

192
193     nr := func(i int64) rational.Rational {
194         return rational.New(i, d: 1)
195     }
196
197     equations := [][]rational.Rational{}
198     for _, v := range arr{
199         eq := []rational.Rational{}
200         for _, x := range v{
201             eq = append(eq, nr(x))
202         }
203         eq = append(eq, nr(i: 0))
204         equations = append(equations, eq)
205     }

```

balance(equation string) string

Run: go build package-main.go x

<4 go setup calls>

```

{1 1} {3 1} {0 1} {-2 1} {0 1}
{1 1} {0 1} {-3 1} {0 1} {0 1}
{1 1} {4 1} {-4 1} {-1 1} {0 1}
{0 1} {1 1} {-1 1} {0 1} {0 1}

```

Figure 4.25 – Creating equation rational matrix

```

213
214     res, gausErr := gaussian.SolveGaussian(equations, printTriangularForm: false)
215     if gausErr != nil {
216         log.Fatal(gausErr)
217     }
218

```

balance(equation string) string

Run: go build package-main.go x

<4 go setup calls>

```

{0 1} {-1 1}
{0 1} {-1 3}
{0 1} {-1 3}
{0 0}

```

Figure 4.26 – Gaussian elimination

As it can be seen in **Figure 4.26**, we got the first three coefficients: -1, -1/3 and -1/3. {0 0} means that the coefficient can be chosen arbitrary, and the most convenient number is the least common multiple of the denominator.

```

37 // greatest common divisor (GCD) via Euclidean algorithm
38 func GCD(a, b int64) int64 {
39     for b != 0 {
40         t := b
41         b = a % b
42         a = t
43     }
44     return a
45 }
46
47 // find Least Common Multiple (LCM) via GCD
48 func LCM(a, b int64) int64 {
49     result := a * b / GCD(a, b)
50     return result
51 }

```

Figure 4.27 – *Least Common Multiple function*

```

223     var lcm int64
224     lcm = 1
225     for _, v := range res{
226         for _, x := range v{
227             if x.GetDenominator() != 0 {
228                 lcm = LCM(lcm, x.GetDenominator())
229             }
230         }
231     }
232     fmt.Println(lcm)

```

balance(equation string) string

Run: go build package-main.go x

<4 go setup calls>

3

Figure 4.28 – *Finding the lcm from denominators*

Taking in consideration that we have found the lcm, now we have to multiply the absolute value of each rational found before with it.

```

233     coeffs := make([]int64, 0)
234     for _, v := range res{
235         for _, x := range v{
236             x = x.MultiplyByNum(lcm)
237             if x.GetNumerator() != 0 {
238                 coeffs = append(coeffs, int64(math.Abs(float64(x.GetNumerator()))))
239             }
240         }
241     }
242     coeffs = append(coeffs, lcm)
243     fmt.Println(coeffs)
244

```

balance(equation string) string

Run: go build package-main.go x

<4 go setup calls>

[3 1 1 3]

Figure 4.29 – *Getting the coefficients*

The final step is to get all together: coefficients, substances, pluses and equal sign in a single string called *result*.

```

245     var result string
246     if (int(coeffs[0])) == 1{
247         result = " " + leftSide[0]
248     }else {
249         result = strconv.Itoa(int(coeffs[0])) + " " + leftSide[0]
250     }
251     for idx, v := range leftSide {
252         if idx == 0{
253             continue
254         }
255         if (int(coeffs[idx])) == 1 {
256             result += " + " + v
257         } else {
258             result += " + " + strconv.Itoa(int(coeffs[idx])) + " " + v
259         }
260     }
261     result += " → "
262
263     gases := []string{"NH3", "AsH3", "CH4", "C6H6", "N2", "CO", "CCl4", "F2
264     sediments := []string{"Mg(OH)2", "Al(OH)3", "Sn(OH)2", "Pb(OH)2", "Cr(O
265
266     if stringInSlice(rightSide[0], gases){
267         rightSide[0] += "[g]"
268     }
269     if stringInSlice(rightSide[0], sediments){
270         rightSide[0] += "[s]"
271     }
272     substanceCoeff := int(coeffs[len(leftSide)])
273     if substanceCoeff == 1{
274         result += rightSide[0]
275     }else{
276         result += strconv.Itoa(int(coeffs[len(leftSide)])) + " " + rightSide[0]}
277     for idx, v := range rightSide {
278         if idx == 0{
279             continue
280         }
281         if stringInSlice(v, gases){
282             v += "[g]"
283         }
284         if stringInSlice(v, sediments){
285             v += "[s]"
286         }
287         substanceCoefficient := int(coeffs[len(leftSide) + idx])
288         if substanceCoefficient == 1{
289             result += " + " + v
290         } else {
291             result += " + " + strconv.Itoa(substanceCoefficient) + " " + v
292         }
293     }
294     return result

```



  3 NaOH + H3PO4 → Na3PO4 + 3 H2O

Figure 4.30 – *Getting the result*

4.4.3 Solving chemical reactions

The solving algorithm, gets as parameter a string, which represents the left side of the chemical reaction and return the whole equation and even balanced. For solve() function, implemented in Python, we will need the solubility table which is displayed in **Figure 4.31** in csv format (see **Figure 4.32**) and a valence json (see **Figure 4.33**).

SOLUBILITATEA ACIZILOR, BAZELOR ȘI SĂRURILOR ÎN APĂ MASELE MOLECULARE RELATIVE ALE LOR																								
Cationul Anionul	H ⁺	Li ⁺	Na ⁺	K ⁺	NH ₄ ⁺	Mg ²⁺	Ca ²⁺	Str ²⁺	Ba ²⁺	Al ³⁺	Sn ²⁺	Pb ²⁺	Cr ³⁺	Mn ²⁺	Fe ²⁺	Fe ³⁺	Co ²⁺	Ni ²⁺	Cu ²⁺	Ag ⁺	Zn ²⁺	Cd ²⁺	Hg ²⁺	
OH ⁻	18	24	40	56	35	58	74	122	171	78	153	241	103	89	90	107	93	93	98	—	99	146	—	
F ⁻	20	26	42	58	37	62	78	107	175	84	157	275	109	93	94	113	97	97	102	127	103	150	231	
Cl ⁻	36,5	42,5	58,5	74,5	53,5	95	111	159	208	133,5	190	278,5	158,5	126	127	162,5	130	130	135	143,5	136	183	272	
Br ⁻	81	87	103	119	98	184	200	248	297	267	279	367	292	215	216	296	219	219	224	188	225	272	361	
I ⁻	128	134	150	166	145	278	294	342	391	408	373	461	433	309	310	—	313	313	—	235	319	366	455	
SO ₄ ²⁻	98	110	142	174	132	120	136	184	233	342	215	303	392	151	152	400	155	155	160	312	161	208	297	
SO ₃ ²⁻	82	94	126	158	116	104	120	168	217	—	199	287	—	135	136	—	139	139	—	296	145	192	—	
S ²⁻	34	46	78	110	68	—	72	120	—	—	151	239	—	87	88	208	91	91	96	248	97	144	233	
NO ₃ ⁻	63	69	85	101	80	148	164	212	261	213	243	331	238	179	180	242	183	183	188	170	189	236	325	
PO ₄ ³⁻	98	116	164	212	—	262	310	454	601	122	547	811	147	355	358	151	367	367	382	419	189	385	793	
CO ₃ ²⁻	62	74	106	138	94	84	100	148	197	—	179	267	—	115	116	—	119	119	—	276	125	172	261	
SiO ₃ ²⁻	78	90	122	154	—	100	116	164	213	282	195	283	—	—	132	—	135	—	—	—	—	188	—	

Substanță solubilă în apă

Substanță insolubilă

Substanță hidralizează complet ori nu există

Substanță puțin solubilă în apă

Figure 4.31 – *Solubility table*

```
solubility.csv
1  elements,H,Li,Na,K,NH4,Mg,Ca,Sr,Ba,Al,Sn,Pb,Cr,Mn,Fe,Fe_,Co,Ni,Cu,Ag,Zn,Cd,Hg
2  OH,S,S,S,S,S,I,P,P,S,P,P,P,P,P,P,P,P,P,-,P,P,-
3  F,S,S,S,S,S,P,I,I,P,P,S,P,S,S,S,S,S,S,S,S,S,I
4  Cl,S,S,S,S,S,S,S,S,S,S,S,P,S,S,S,S,S,S,S,I,S,S,P
5  Br,S,S,S,S,S,S,S,S,S,S,S,P,S,S,S,S,S,S,S,I,S,S,S
6  I,S,S,S,S,S,S,S,S,S,S,P,P,S,S,S,-,S,S,-,I,S,S,I
7  SO4,S,S,S,S,S,S,P,I,I,S,S,I,S,S,S,S,S,S,S,P,S,S,I
8  SO3,S,S,S,S,S,S,I,I,P,-,I,I,-,I,I,-,I,I,-,I,P,P,-
9  S,S,S,S,S,S,-,S,I,-,-,I,I,-,I,I,I,I,I,I,I,I,I,I
10 NO3,S,S,S,S,S,S,S,S,S,S,I,S,S,S,S,S,S,S,S,S,S,S
11 PO4,S,I,S,S,-,P,I,I,I,I,I,I,I,I,I,I,I,I,I,I,I,I
12 CO3,S,S,S,S,S,P,I,I,I,-,I,I,-,I,I,-,I,I,-,I,I,I,I
13 SiO3,I,S,S,S,-,I,I,I,I,I,I,I,-,-,I,-,I,-,-,-,I,-
```

Figure 4.32 – *solubility.csv*


```

1  {} valence.json > ...
2  {
3    "cations" : {
4      "H" : 1,
5      "Li" : 1,
6      "Na" : 1,
7      "K" : 1,
8      "NH4" : 1,
9      "Mg" : 2,
10     "Ca" : 2,
11     "Sr" : 2,
12     "Ba" : 2,
13     "Al" : 3,
14     "Sn" : 2,
15     "Pb" : 2,
16     "Cr" : 3,
17     "Mn" : 2,
18     "Fe" : 2,
19     "Fe_" : 3,
20     "Co" : 2,
21     "Ni" : 2,
22     "Cu" : 2,
23     "Ag" : 1,
24     "Zn" : 2,
25     "Cd" : 2,
26     "Hg" : 2
27   },
28   "anions" : {
29     "OH" : 1,
30     "F" : 1,
31     "Cl" : 1,
32     "Br" : 1,
33     "I" : 1,
34     "SO4" : 2,
35     "SO3" : 2,
36     "S" : 2,
37     "NO3" : 1,
38     "PO4" : 3,
39     "CO3" : 2,
40     "SiO3" : 2
41   },
42   "oxide_to_ions" : {
43     "CO2" : "CO3",
44     "SO2" : "S",
45     "SO3" : "S",
46     "NO2" : "NO3",
47     "P2O5" : "PO4",
48     "SiO2" : "SiO3"
49   }
50 }

```

Figure 4.33 – valence.json

In this algorithm we will use regular expressions as patterns for oxides, acids, bases and salts. Their definition, cations, anions and valence extraction can be seen in **Figure 4.34**.

```

132
133 # Defining the chemistry substances types regular expressions
134 OXIDE_PATTERN = '\w+\d*O(?:H)\d*'
135 ACID_PATTERN = 'H\d*.\+'
136 BASE_PATTERN = '\w+\d*(?:OH)\d*'
137 SALT_PATTERN = '[A-Z][a-z]?\d*(?:.*?)\d+'
138
139 # Importing the solubility table.
140 df = pd.read_csv('solubility.csv', index_col='elements')
141
142 # Getting the cations and the anions lists.
143 cations = list(df.columns)
144 anions = list(df.index)
145
146 # Loading the valences of every cation and anion.
147 valences = json.load(open('valence.json', 'r'))
148
149 # Extracting the reactants from the passed reaction.
150 reactants = [x.strip() for x in chem_reaction.split('+')]
151

```

Figure 4.34 – Defining the regular expressions

The next step is to extract ions from the reactants and this is done differently in dependence on substance's type. For example, if it is a salt, we separate the cations and anions easily; if it is an acid, we take the 'H' into array of ions and then the rest of the substance is the anion; if it is an oxide,

we have in the valence dict how it changes from oxide to ion (CO_2 turns into CO_3^{2-}); and if it is a base, we take the 'OH' and the rest of the string is also an ion.

```

152 # ----- IONS EXTRACTION -----
153
154 # Creating the empty list o ions.
155 ions = []
156
157 # Matching every substance with the types of substances to extract the ions differently in each cases.
158 for i in range(len(reactants)):
159     # Matching the SALT Pattern.
160     if bool(re.match(SALT_PATTERN, reactants[i])):
161         # Searching for the anion from the salt and saving it in the ions list.
162         for anion in anions:
163             if anion in reactants[i]:
164                 ions.append(anion)
165         # Searching for the cation from the salt and saving it in the ions list.
166         for cation in cations:
167             if cation in reactants[i]:
168                 ions.append(cation)
169     # Matching the ACID Pattern.
170     elif bool(re.match(ACID_PATTERN, reactants[i])) and 'OH' not in reactants[i]:
171         # Extracting the H+ ion and the free radical ion.
172         ions.append('H')
173         temp = reactants[i].replace('H', '')
174         temp = temp[1:] if temp[0].isnumeric() else temp
175         ions.append(temp)
176     # Matching the OXIDE Pattern.
177     elif bool(re.match(OXIDE_PATTERN, reactants[i])):
178         # Extracting the ion for of the oxide.
179         ions.append(valences['oxide_to_ions'][reactants[i]])
180     # Matching the BASE Pattern.
181     elif bool(re.match(BASE_PATTERN, reactants[i])):
182         # Extracting the OH- ion and the free radical ion.
183         ions.append('OH')
184         temp = reactants[i].replace('OH', '')
185         temp = temp.replace('()', '')
186         temp = temp[:-1] if temp[-1].isnumeric() else temp
187         ions.append(temp)

```

Figure 4.35 – Ions extraction

The final step is to get the result and for this we will need two matrices *result_table* which will store per row at least 2 elements, the initial substance and how it changed after the reaction took place and *result_state*, will take the information from solubility table and will consist of 'S', 'I', 'P' or 'GAS'. Then, we will get the new obtained results substances and the list of resulting states. From chemistry lessons, we all know that a chemical reaction took place if the products are either gas, partial soluble or insoluble substances; in other cases it is called reversible. We balance the obtained string and here it is, the result.

```

193 # Building in parallel 2 matrices, one for possible result and another for the physical states of the products.
194 result_table = []
195 result_state = []
196
197 # Constructions all the combinations with the cations and anions.
198 for cat in result_cations:
199     result_table.append({})
200     result_state.append({})
201     for anion in result_anions:
202         # Building the substance formulas using the ions valences.
203         if valences['cations'][cat] == valences['anions'][anion]:
204             result_table[-1].append(f"{cat}{anion}")
205         else:
206             result_table[-1].append(f"{cat}{valences['anions'][anion] if valences['anions'][anion] != 1 else ''}{anion if valences['cations'][cat] == 1 else ''}")
207         # Saving the result states.
208         if result_table[-1][-1] == 'H2CO3':
209             result_state[-1].append('GAS')
210         else:
211             result_state[-1].append(df.loc[anion, cat])
212
213 # Getting the new obtained results substances.
214 result = []
215 final_state = []
216 for i in range(len(result_table)):
217     for j in range(len(result_table[i])):
218         if result_table[i][j] not in reactants:
219             result.append(result_table[i][j])
220             final_state.append(result_state[i][j])
221
222 # Checking if in the products is present the carbonic acid.
223 # if it is present then it transforms into CO2 and H2O
224 for i in range(len(result)):
225     if 'H2CO3' == result[i]:
226         result[i] = 'CO2 + H2O'
227     if "HOH" == result[i] or "OHH" == result[i]:
228         result[i] = 'H2O'
229
230 result = list(set(result))
231 # Printing out the result.
232 res = chem_reaction + ' = ' + ' '.join(result)
233 res = balance(res)
234 return res

```

Figure 4.36 – Getting the resulting equation

4.4.4 Printing chemical reactions

Another function implemented in our DSL is print(). It is created in order to print the reactions after they were balanced or solved, or just after assigning the value to a variable.

```

61 func printEquation(equation string) (string, error) {
62     var sides []string
63     var leftSide []string
64     var rightSide []string
65
66     var result string
67     var err error = nil
68
69     exception.Block{
70         Try: func() {
71             equation = strings.ReplaceAll(equation, old: " ", new: "")
72             sides = strings.Split(equation, sep: "=")
73             leftSide = strings.Split(sides[0], sep: "+")
74             rightSide = strings.Split(sides[1], sep: "+")
75         },
76         Catch: func(e exception.Exception) {
77             err = errors.New(text: "unreadable equation")
78         },
79     }.Do()
80
81     if err != nil {
82         return result, err
83     }
84

```

Figure 4.37 – printEquation() function part1

```

85     var leftCompounds []map[string]int
86     var rightCompounds []map[string]int
87     s1 := make(map[string]bool)
88     s2 := make(map[string]bool)
89     for i := 0; i < len(leftSide); i++ {
90         leftCompounds = append(leftCompounds, parse(leftSide[i]))
91         for k := range parse(leftSide[i]) {
92             s1[k] = true
93         }
94     }
95     for i := 0; i < len(rightSide); i++ {
96         rightCompounds = append(rightCompounds, parse(rightSide[i]))
97         for k := range parse(rightSide[i]) {
98             s2[k] = true
99         }
100    }
101
102    result = leftSide[0]
103    for idx, v := range leftSide {
104        if idx == 0 {
105            continue
106        }
107        result += " + " + v
108    }
109    result += " = "

```

Figure 4.38 – *printEquation()* function part2

```

110
111    gases := []string{"NH3", "AsH3", "CH4", "C6H6", "N2", "CO", "CCl4", "F2",
112    sediments := []string{"Mg(OH)2", "Al(OH)3", "Sn(OH)2", "Pb(OH)2", "Cr(OH)
113    if stringInSlice(rightSide[0], gases) {
114        rightSide[0] += "[G]"
115    }
116    if stringInSlice(rightSide[0], sediments) {
117        rightSide[0] += "[S]"
118    }
119    result += rightSide[0]
120    for idx, v := range rightSide {
121        if idx == 0 {
122            continue
123        }
124        if stringInSlice(v, gases) {
125            v += "[G]"
126        }
127        if stringInSlice(v, sediments) {
128            v += "[S]"
129        }
130
131        result += " + " + v
132    }
133

```

Figure 4.39 – *printEquation()* function part3

4.5 Graphical User Interface

GUI is a Graphical Interface that is a visual representation of communication presented to the user for easy interaction with the machine. GUI means Graphical User Interface. It is the common user Interface that includes Graphical representation like buttons and icons, and communication can be performed by interacting with these icons rather than the usual text-based or command-based communication.

The GUI for our DSL is a dedicated code editor, where the user will be able to write his code, save it, run it and look at the output in the console. To develop our editor we used Python, for its flexibility and GUI building libraries which are easy to understand and use. The library we used it's called Tkinter.

Tkinter is a Python binding to the Tk GUI toolkit. It is the standard Python interface to the Tk GUI toolkit, and is Python's de facto standard GUI. Tkinter is included with standard Linux, Microsoft Windows and Mac OS X installs of Python. The name Tkinter comes from Tk interface. Tkinter was written by Fredrik Lundh. Tkinter is free software released under a Python license.

So, our GUI consists of 4 parts: a text area where the user will be able to write his code, a run button, an area which will serve as a console so he can see the output and a menu bar where he will be able to manipulate the file: open file, save, save as, exit.

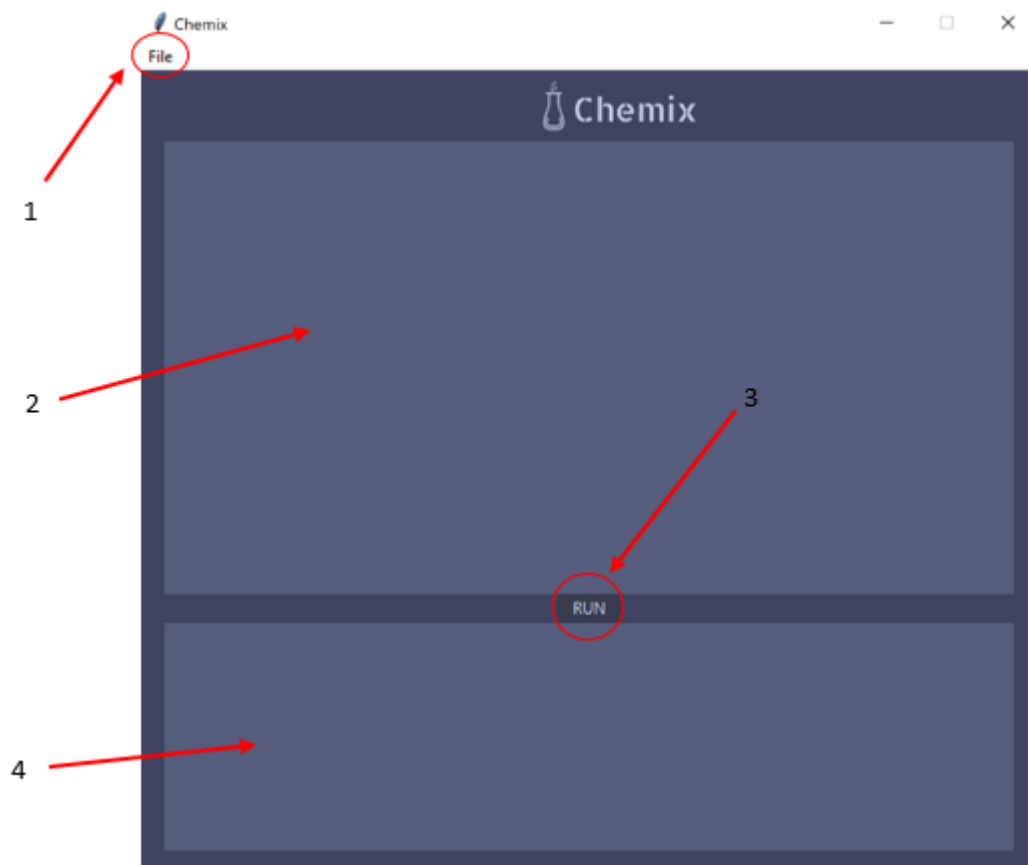


Figure 4.40 – Overview of our GUI

Menu consists of two main functions which are `open_file()` and `save_as()`. They are represented in **Figure 4.41**.

```

17 def set_file_path(path):
18     global file_path
19     file_path = path
20
21
22 def open_file():
23     path = askopenfilename(filetypes=[('Files', '*.txt')])
24     with open(path, 'r') as file:
25         code = file.read()
26         editor.delete('1.0', END)
27         editor.insert('1.0', code)
28         set_file_path(path)
29
30
31 def save_as():
32     if file_path == '':
33         path = asksaveasfilename(filetypes=[('Files', '*.txt')])
34     else:
35         path = file_path
36     with open(path, 'w') as file:
37         code = editor.get('1.0', END)
38         file.write(code)
39         set_file_path(path)

```

Figure 4.41 – *Menu*

The input and output areas are simple Text widgets from Tkinter where the user can write text and also you can insert and delete using class functions from Text widget. To make the connection of input and output we made a run button which on click runs the run() function which is presented in **Figure 4.42**.

```

42 def run():
43     if file_path == '':
44         code_output.delete('1.0', END)
45         code_output.insert('1.0', "Please Save your Code!")
46         return
47     code_output.delete('1.0', END)
48     process = subprocess.run([exec, file_path], capture_output=True, text=True)
49     if process.returncode == 0:
50         code_output.insert('1.0', process.stdout)
51     else:
52         code_output.insert('1.0', process.stderr, 'warning')

```

Figure 4.42 – *run() function*

In order to connect the parser, lexer and functionality from Go we used the subprocess module which is used to run new applications or programs through Python code by creating new processes. It also helps to obtain the input/output/error pipes as well as the exit codes of various commands.

We generated a .exe file which is sent as an argument to subprocess.run() function along with the input file saved by the user “file_path”. And if this process returns 0 then we insert the output in the console of our GUI else we insert the error message with another color.

4.6. Examples

Now, as we explained how everything was implemented, let us see how it works; for this, we have prepared some code examples and the output can be seen in figures below.



Figure 4.43 – *Example 1*



Figure 4.44 – *Example 2*

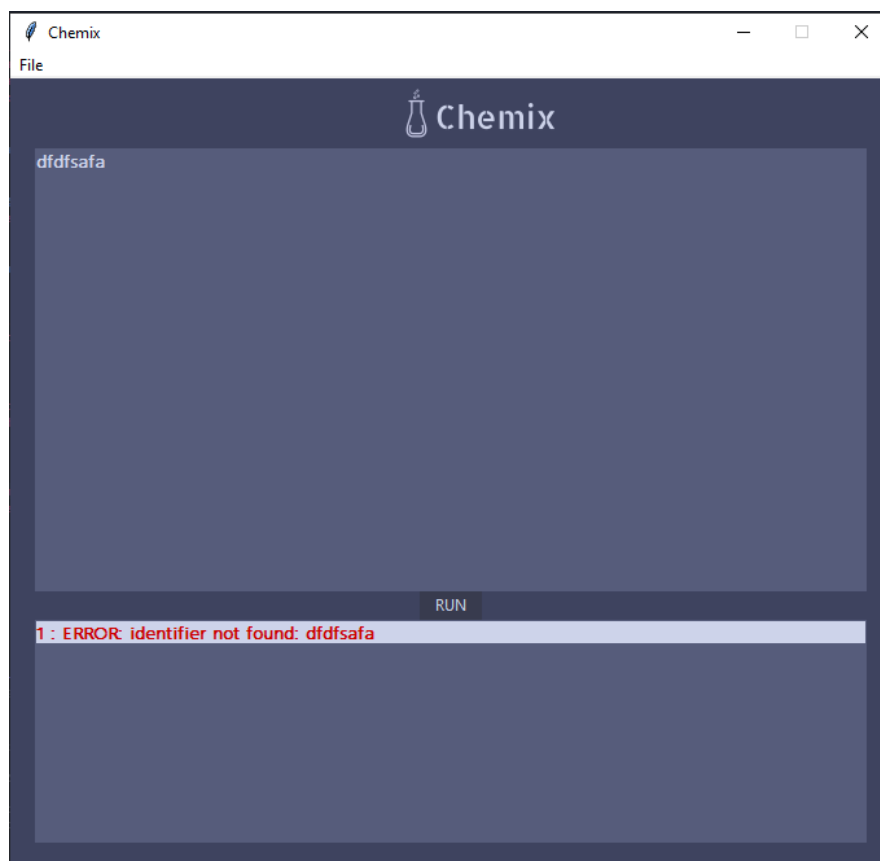


Figure 4.45 – *Example 3*

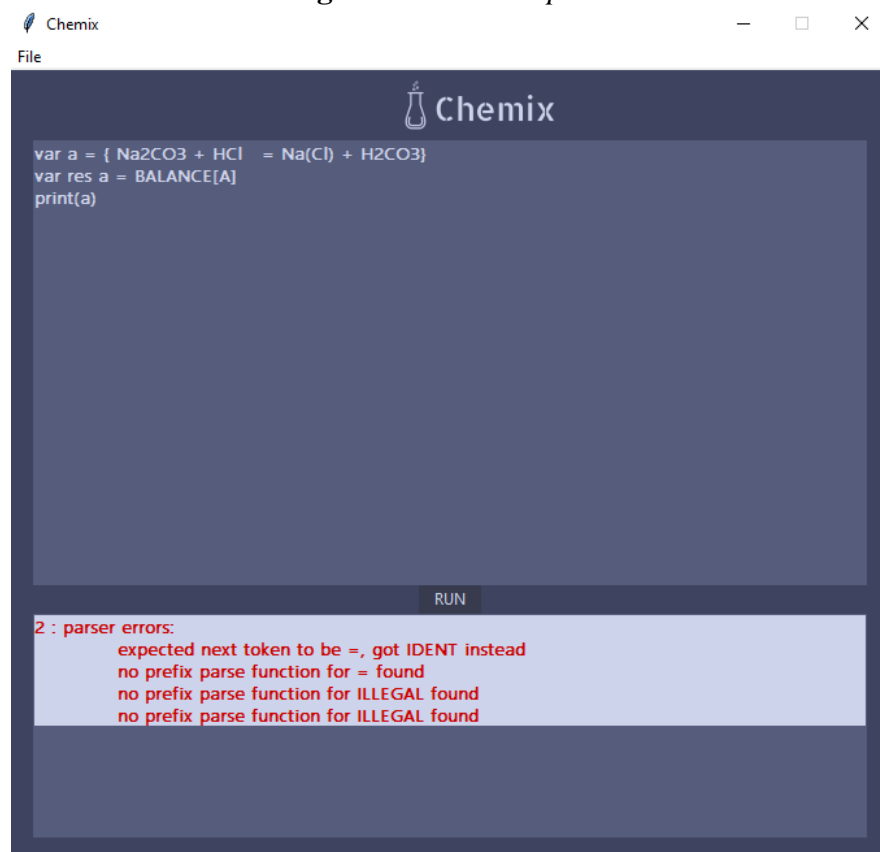


Figure 4.46 – *Example 4*



Figure 4.47 – *Example 5*

4.7 Links to all repositories which store project components

Below we have listed the links to all repositories which store the project components:

- <https://github.com/Dunitrashuk/Chemix>
- <https://github.com/vivk/pbl-chemistry-dsl>

Conclusion

This report included the presentation of the most important information concerning the goal of our Domain Specific Language for Chemistry, its design and functionalities.

Our starting point was the domain identification, thus after a brainstorming on problems that different people and gener face, we came up with several problems, but we have chosen the one that we, as students, also faced it in school, at chemistry lessons. From our point of view, creating a DSL that will help our target audience – pupils will be a great idea to solve some problems and also to support people in studying this object.

The phenomenon on which is based our project is the struggle of solving chemical equations. Therefore, in order to prove that this is actually a problem, we made a survey in which people were asked a couple of questions. After analyzing the answers, we understood that pupils would rather use a Domain Specific Language to solve or balance chemical equations, rather doing it by themselves, because of their lack of experience and the desire to not make mechanical mistakes

Before developing our DSL, we analysed the existing solution, but we found only one, which is not widely used and requires an academic level of knowledge. From this point, we understood that our main goal was to create a useful tool that would improve pupils' understanding of chemistry problems and concepts.

References

1. Paul Hudak, *Domain Specific Languages*, Department of Computer Science, Yale University, 1997 (link: <https://docplayer.net/369257-Domain-specific-languages.html>).
2. Linus Pauling, *The significance of chemistry* (link: <https://collections.nlm.nih.gov/ocr/nlm:nlmuid-101584639X111-doc>).
3. Jun Cao, Ayush Goyal, Samuel P. Midkiff, James M. Caruthers, *Caruthers An Optimizing Compiler for Parallel Chemistry Simulations*, 2007.
4. <http://www.csc.lsu.edu/~gb/TCE/> The Tensor Contraction Engine
5. Ranald Cloutson, *Non-Deterministic Finite Automata and Grammars* (link: <https://cs.anu.edu.au/courses/comp2600/2013/lectures/NFA.pdf>).
6. *What is ANTLR?* (link: <https://www.antlr.org/>).
7. Victor Osadchiy, *Why Use the Go Language for Your Project?* (link: [What Is Go Language and Why Use It for Your Project \(yalantis.com\)](http://yalantis.com/what-is-go-language-and-why-use-it-for-your-project/)).
8. Helmenstine, Anne Marie, Ph.D. *Types of Chemical Reactions*. ThoughtCo, Aug. 28, 2020, (link: [Types of Chemical Reactions \(With Examples\) \(thoughtco.com\)](https://www.thoughtco.com/types-of-chemical-reactions-with-examples-603486/)).
9. Cephas Iko-ojo Gabriel, *Balancing of Chemical Equations using Matrix Algebra*, Department of Mathematics, Faculty of Sciences, University of Science and Technology, Aleiro, Kebbi State, Nigeria, 2015