

TKP4555 Advanced Simulation

Marius Reed

December 1, 2017

Regular expression, Backus Naur and Compilers



Norwegian University of
Science and Technology

Department of chemical engineering

Contents

1	Introduction	1
2	Regular Expressions	1
2.1	Writing regular expression	1
2.1.1	Single character pattern	2
2.1.2	The dot - any character	2
2.1.3	Range of characters	2
2.1.4	Multipliers	3
2.1.5	Shortcut for character classes	3
2.1.6	Anchors	4
2.1.7	Word boundaries	4
2.1.8	Grouping	5
2.1.9	Alternation	5
2.2	Recap of whats been introduced in tabular form	5
2.3	Examples of application of regular expressions	6
2.3.1	Finding NTNU email addresses	6
2.3.2	Finding person number for people born in the 90's	6
2.3.3	Defining tokens for a parser	6
2.4	Regex in python	6
3	Backus-Naur Form	8
3.1	Rail road diagram (Syntax diagram)	9
4	Compiler	11
4.1	Lexical analysis (<i>lexer</i>)	12
4.1.1	Lexer in the compiler	12
4.2	Parsing	13
4.2.1	Example: Calculator	13
4.2.2	Parser in the compiler	15
4.3	Code generation	18
5	Closing Remarks	19
	References	20
	Appendix	i
A	Regex examples: python re module	i
B	LPEG: Compiler	v
C	Input to compiler	ix
D	Output from compiler: MATLAB code	xv

1 Introduction

This report is part of the TKP4555 Advanced Simulation course. The main objective of the project has been to translate system equations, which is the output from a modeling tool developed by Prof. Heinz Preisig, into Matlab code by the use of a *compiler* written in Lua. The report will cover some parts of formal language theory. First of all an introduction to regular expressions, which is a sequence of characters that define search *patterns*, will be given. Regular expressions is widely used in search algorithms for "find" or "find and replace" string operations. They are also used to define *tokens*, which is used in the *lexical analysis (lexer)* part of *compilers*. In addition to introducing how to write regular expression, the re module in python will be presented.

The second part of the report will cover some of the theory of compilers. This part includes how the compiler in Lua has been developed. *LPeg*, a pattern matching library in Lua, *lexical analysis (lexer)*, *Syntax analysis (parser)* and *code generation* will be presented.

2 Regular Expressions

A regular expression (often called regex or regexp) is a sequence of characters that defines a pattern. Such patterns are often used by search algorithms for "find" or "find and replace" string operations. Most programming languages supports the usage of regular expressions. Examples in this report will be presented using the re module in python, but similar modules exists in other languages such as Java and C++. In the first section the main focus will be on the regular expression syntax, whilst the usage of regex in python will be covered in section 2.4. There are many things that can be done with regular expression. Some of these applications are:

- **Search** for certain patterns inside a large text. An example could be to pull out all NTNU student mail addresses from a mailing list by matching "@stud.ntnu.no".
- **Validate** that a text consists of the demanded types of characters in for example a password.
- **Replace** part of a text. An example could be to correct every lower case letters in the beginning of a sentence to upper case letters.
- **Reformat** a text file such that an output file from one source can be fed as an input to another source.

All of these applications will be covered in this report, but before considering to use regular expressions for an application you need to know how to write regular expressions. Mastering the entire syntax of writing regular expression is comprehensive, but a lot can be achieved by knowing the basics and using cites such as <https://ryanstutorials.net/regular-expressions-tutorial/#learning> as support when writing more advanced expressions[5].

2.1 Writing regular expression

In this sections some parts of writing regex is presented. The section is structured on a step-by-step basis, meaning more and more complicated parts will be introduces. Each newly introduced term will be shown as a example. The example is presented inside a box where the regular expression is located in the top left, whilst the matches that will be made is highlighted in yellow.

2.1.1 Single character pattern

The most basic regular expression is the single character pattern. To find all *a*'s in a sentence a regular expression as simple as **a** is used.

a
A computer once beat me at chess, but it was no match for me at kick boxing. - *Emo Philips*

Notice that the first "A" will not be matched. The reason for this is that regular expressions are case sensitive. It is possible to make the regular expression to search in a case insensitive way by making it **[aA]**. An example on different searching methods in python and their corresponding outputs is given below as a motivation on how to use regular expressions. A more comprehensive introduction to regex in python is given in section 2.4.

```
import re #Importing the regex module in python
# Regular expression used to match all a's in a sentence.
a = re.compile('a') #Creates a var 'a' which is a
# regular expression of the letter 'a'
matches = a.findall('A computer once beat me at chess,' +
'but it was no match for me at kick boxing.') #Finds all matches
iterator = a.finditer('A computer once beat me at chess,' +
'but it was no match for me at kick boxing.') #Returns position of matches as an iterator
# Output
print(matches)
for match in iterator:
    print match.span()
```

```
output >> ['a', 'a', 'a', 'a', 'a']
(18, 19)
(24, 25)
(41, 42)
(48, 49)
(60, 61)
```

2.1.2 The dot - any character

The dot (.) is something called a metacharacter and such characters is used to make more interesting characters. The dot (.) matches any character, and can be used to find patterns where the first and third character is fixed, whilst the second character to be anything. An example of such a regex is **a.e**. This expression will match any string beginning on **a** followed by any character before an **e**.

a.e
Software is like entropy: It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases. - *Norman Augustine*

2.1.3 Range of characters

Another very useful part of regular expressions is **ranges**. Such ranges can be used when there is need to be more specific than using the dot. Ranges is written within brackets **[]** and any of the characters inside the brackets can be matched. The following regex, **er[ey]**, will match **e** followed by an **r** and then followed by either an **e** or **y**.

er[ey]

Th**ere** is only one problem with common sense; it's not v**ery** common. - *Milt Bryce*

The range can consist of any amount of characters. While it does not give much sense to put only one character inside the brackets, it is legal.

With ranges there are shortcuts for characters in a row in ASCII. **[0-9]** will match all integers between **0** and **9**, whilst **[0-9a-z]** will match any lower case characters as well.

2.1.4 Multipliers

In regular expression there exist different multipliers. Such multipliers make it possible to increase the number of times an item can occur. The multipliers are as follow:

- ***** : item occurs zero or more times.
- **+** : item occurs one or more times.
- **?** : item occurs zero or one time.
- **{a}** : item occurs exactly a times.
- **{a,b}** : item occurs between a and b times.
- **{a,}** : item occurs at least a times.

The effect that the multiplier has will be applied to the character directly to the left. The regex **et+** will match every instances of an **e** followed by one or more **t**'s.

et+

Get**ting** information off the Internet**et** is like taking a drink from a fire hydrant. - *Mitchell Kapor*

Changing the **+** to ***** would make the regular expression to match with all **e**'s in addition to the once highlighted above. When using multipliers it is important to bear in mind that the regex gets greedy if the dot (.) is used. It is greedy in the way that it will try to match as big parts of the text as possible. The regex **a.*e** will match the entire text from the first **a** until the last **e** in the text.

a.*e

The gre**ate**st enemy of knowledge is not ignorance, it is the illusion of knowledge. - *Stephen Hawking*

If the objective is to match only the parts starting from an **a** until the first instance of an **e**, this can be fixed including another multiplier, namely the **?**. **a.*?e** will match the following parts of the same sentence:

a.*?e

The gre**ate**st enemy of knowledge is not ignor**ance**, it is the illusion of knowledge. - *Stephen Hawking*

2.1.5 Shortcut for character classes

Some sets of characters are more used than others. Whilst it is possible to create these sets using ranges, some shortcuts has been made to refer to them. The following shortcuts exists in regex:

- `\s` : Matches anything which is considered whitespace.
- `\S` : Matches anything which is NOT considered whitespace.
- `\d` : Matches anything considered a digit. (Same as `[0-9]`)
- `\D` : Matches anything which is NOT considered a digit.
- `\w` : Matches anything which is considered a word character. (Same as `[A-Za-z0-9_]`)
- `\W` : Matches anything which is NOT considered a word character.

Note that the lower and upper case letters always has the opposite meanings. Combining these shortcuts with multipliers can make some interesting regular expression. The regex that will match every word in a text can be written as easy as `\w+`.

`\w+`

Imagination is more important than knowledge. For knowledge is limited, whereas imagination embraces the entire world, stimulating progress, giving birth to evolution. - *Albert Einstein*

2.1.6 Anchors

There are two symbols which represents the beginning and end of the line. The `^` represents the beginning of the line, whilst `$` represents the end of the line. They are called anchors as they allow to anchor the pattern to a particular point on the line. Using these, it is essential knowing how beginning and end of lines are defined in regular expression. Normally one would say that the beginning of the line is the first character, but this is incorrect with respect to regular expressions. The beginning of the line is a zero width character just before the first character. The same applies to the end of line, but at the end of the last symbol. They are not visible, but they are still there. Matching the first word on a line could be done with the following regex, `^\w+`.

`^\w+`

Your most unhappy customers are your greatest source of learning. - *Bill Gates*

Matching the last word could be a little more complicated as there might be a ".", "?" or "!" at the end of the line. A regex that will match the last word in the text above would be `\w+[\.\!\\?]?$`. Note the `\` in front of `.`, `!` and `?`. This is necessary to make the regex recognize it as the characters instead of the dot, the negative lookahead symbol (not covered in this report) and the `?` multiplier.

`\w+[\.\!\\?]?$`

Your most unhappy customers are your greatest source of learning. - *Bill Gates*

2.1.7 Word boundaries

The anchors is useful to match the beginning and end of the line, but there is also some symbols that represents the end and beginning of a word, namely `\b`. To match every word beginning with `i`, the following regex can be used, `\bi\w+\b`

`\bi\w+\b`

The best way to get accurate information on Usenet is to post something wrong and wait for corrections. - *Matthew Austern*

2.1.8 Grouping

In regex it is possible to group several characters together putting them inside parentheses, "()". Doing so, it is possible to use multipliers to allow a sequence to repeat itself. An application for this can be matching IP addresses, which consist of 4 numbers between 0 and 255 separated by ".". The regex for an IP address is `(\d{1,3}\.){3}\d{1,3}`.

`(\d{1,3}\.){3}\d{1,3}`

This is an IP address `210.1.41.103`, but this is not `112.3332.124.1`

2.1.9 Alternation

Using alternation we are looking for something **or** something else. The **or** symbol in regex is `|`. The regex can be used to search for different names by for example `John|Michael|Paul`, or it can be used inside a group to match with a persons name with a given surname by `(John|Michael) Smith`.

`(John|Michael) Smith`

We want to match `John Smith` and `Michael Smith`, but not Paul Smith.

2.2 Recap of whats been introduced in tabular form

Item	Symbol	Example	Description
Single character		'a'	Matches a single character
The dot	.	a.b	Matches any character
Ranges	[]	[a-z]	Matches one character inside the range
Multipliers			
	*	a*	Item occurs zero or more times
	+	a+	Item occurs one or more times
	?	a?	Item occurs zero or one time
	{i}	b{i}	Item occurs exactly a times
	{i,j}	A{i,j}	Item occurs between i and j times
	{i,}	A{i,}	Item occurs at least i times
Shortcuts			
	\s	\s	Matches anything considered whitespace
	\S	\S	Matches anything NOT considered whitespace
	\d	\d	Matches anything considered a digit (same as [0-9])
	\D	\D	Matches anything NOT considered a digit
	\w	\w	Matches anything considered a word character
	\W	\W	Matches anything NOT considered a word character
Anchors			
	^	^\w+	Matches beginning of line
	\$	\w+\$	Matches end of line
Word boundary	\b	\b\w+\b	Matches beginning and end of line
Grouping	()	(ABC)+	Groups items such that multiplier affects the entire item
Alternation		dog cat	Works as an or operator

Table 1: Summary of the regex syntax introduced in this report.

2.3 Examples of application of regular expressions

2.3.1 Finding NTNU email addresses

An application of regular expressions can be used to find some specific email addresses in a long list of email addresses. NTNU email addresses can be found since the format of such emails is known. It will consist of the **username** consisting of only lower case letters followed by @ followed by either **stud.ntnu.no** or **ntnu.no**. The regex for this will be `[a-z]+@(stud\.|ntnu\.no)` or `[a-z]+@(stud\.ntnu\.no|ntnu\.no)`. The result of using this on a textfile consisting of several emails is presented below.

```
[a-z]+@(stud\.|ntnu\.no)
```

```
mariusre@stud.ntnu.no; mariusre@ntnu.no; marius.reed@lyse.net;john.smith@gmail.com
```

2.3.2 Finding person number for people born in the 90's

The person number in Norway consists of 11 digits where the last 2 digits represent what year a person is born. It is possible to find the person numbers for only the persons born in the 90's by using the following regex, `\d{9}9\d`. The regular expression demands 9 random digits followed by a 9 followed by a random digit to match, which would correspond to a person born in the 90's.

```
\d{9}9\d
```

```
Paul Smith 12345190192
```

```
John Smith 31290221172
```

```
Michael Smith 31231170599
```

2.3.3 Defining tokens for a parser

Regular expressions can be used to define tokens used by a parser. This will be covered in a detailed way in section 4.

2.4 Regex in python

The `re` module provides regular expression matching operators and both patterns and strings to be searched can be Unicode strings as well as 8-bit strings. In the module there are many functions that can be very useful. In this section the most basic functions will be covered. All the functions are described in <https://docs.python.org/2/library/re.html>.

re.compile(pattern, flags=0)

Compiles a regular expression pattern into a regular expression object. It is possible to use the functions in the module without using **compile()**, but this will save the resulting object for reuse.

```
a = re.compile(pattern)
result = a.match(string)
```

is equivalent to

```
result = re.match(pattern,string)
```


re.search(pattern,string,flags=0)

Searches for the first location where the pattern is matched in the string and returns a corresponding **MatchObject**. If there is no match, "None" is returned.

re.match(pattern,string,flags=0)

Does the same as **search()**, but it only searches in the beginning of the line. This means that it is possible to use the **search()** function instead of the **match()** by adding a **^** (beginning of line symbol) in front of the pattern.

```
re.match("a",string)
```

is equivalent with

```
re.search("^a",string)
```

re.split(pattern,string,maxsplit=0, flags = 0)

Splits the string by the occurrences of *pattern*. An example is splitting a string into a list consisting of all the words. This can be used to split a string into single words.

```
>>> result = re.split("\W+", "Hello ! How are, you doing?")
['Hello', 'How', 'are', 'you', 'doing', '']
```

re.findall(pattern,string,flags=0)

Returns all matches of pattern in string as a list of strings.

```
>>> result = re.findall("\W+", "Hello ! How are, you doing?")
[' ! ', ', ', ' ', '?']
```

re.finditer(pattern,string,flags=0)

Return an iterator yielding **MatchObject** instances over all non-overlapping matches for the RE *pattern* in *string*. The iterator consists of indexes corresponding to the position of where a match was found.

```
>>> string = 'abcdefghijklmnopqrstuvw'
>>> position = re.finditer(r'i',string)
>>> for pos in position:
    print pos.span()
    print(string[pos.span()[0]])
(8,9)
i
```

re.sub(pattern,repl,string,count=0,flags=0)

Returns a string where the occurrences of *pattern* in *string* is replaced by *repl*.

```
>>> result = re.sub(r'\b[iI]\w+\b', 'Replaced', 'It\'s hardware that makes a
↪ machine fast. It\'s software that makes a fast machine slow.')
```

Replaceds hardware that makes a machine fast. Replaceds software that makes a
↪ fast machine slow.

3 Backus-Naur Form

In computer science, Backus-Naur form (BNF) is a notation technique for context-free grammars¹. The Backus-Naur form is often used to describe the syntax of languages used in computing. A BNF specification is a set of derivation rules written as:

$$\langle \text{symbol} \rangle ::= _ _ \text{expression} _ _$$

Here $\langle \text{symbol} \rangle$ is a *nonterminal*, meaning that it is not one of the elementary symbols of the language defined. $_ _ \text{expression} _ _$ consists of one or more sequences of symbols, where more sequences are separated by a vertical bar "|". The entire $_ _ \text{expression} _ _$ represents all possible substitution for the $\langle \text{symbol} \rangle$ on the left. Symbols that never appear on a left side are *terminals*. For example a sentence is a nonterminal in a written language as it consists of multiple characters, which will be terminals. The "::=" means that the symbol on the left must be replaced with the expression on the right.

The Backus-Naur is best explained by an example of a known "syntax". Such an example is a postal address in Norway. The BNF for a postal address is given below:

```
<postal-address> ::= <name-part> <street-address> <zip-part>
<name-part>      ::= <personal-part> <last-name> | <personal-part> <name-part>
<personal-part>  ::= <initial> "." | <first-name>
<street-address> ::= <street-name> <house-num> <opt-apt-num>
<zip-part>       ::= <ZIP-code> <town-name>
<opt-apt-num>    ::= <apt-num> | ""
```

Translated into English:

- A postal address consist of a name, followed by the street address and the zip code.
- The name part consists of personal part followed by the last name, or it can be a personal part followed by a name part. This illustrates the recursion in BNF and allows it for people to have more than one first names.
- The personal part consists of either the initial or the first name.
- The stress address consist of the street name, followed by house number and an optional apartment number.
- The zip part consists of a ZIP code followed by the name of the town.
- The optional apartment number consists of either the apartment number of empty string.

In the BNF above there are still missing some parts. For example what a name consists of (which would be characters in a sequence). For the postal address to be complete such things would have to be described by additional BNF rules.

In the next section lexers, parsers and compilers will be introduced. In the same chapter a parser is used to translate system equations with a specific syntax into matlab code. The BNF of the syntax of the equations is presented below:

¹A context-free grammar (CFG) is a type of formal grammar where a set of production rules (replacement) describe all possible strings in a given formal language. The replacement of the non-terminal symbol (left hand side of the production) is not affected of what is before or after the symbol.

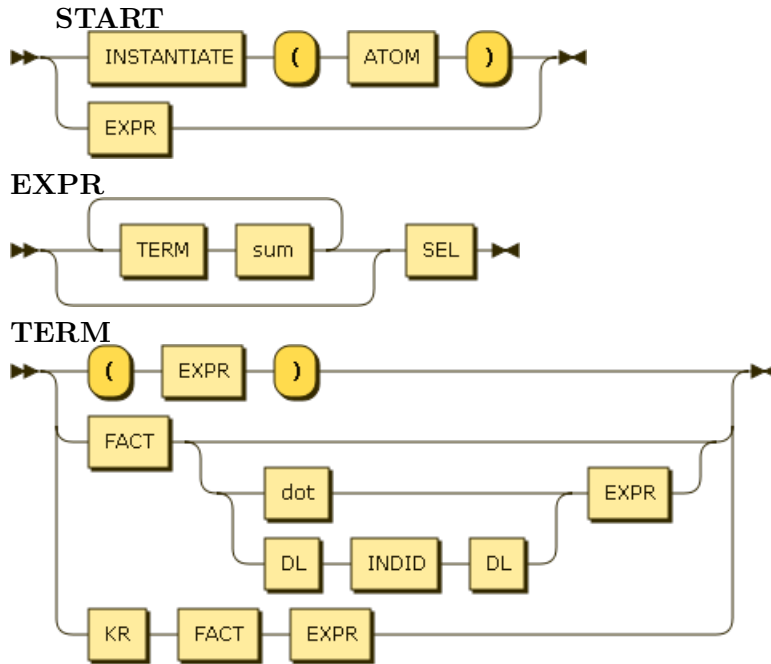
```

<START>      ::= <Instantiate> "(" <ATOM> ")" | <EXPR> | <SEL>
<EXPR>       ::= <TERM> | <EXPR> <sum> <TERM>
<TERM>       ::= <FACT> <dot> <EXPR> | <KR> <FACT> <EXPR> |
<FACT><DL> <INDID> <DL> <EXPR>
<FACT>       ::= <SATOM> <power> "" <SINTEGER> "" | <SATOM> <power> ""
<SATOM>      ::= <sum> <ATOM> | <ATOM>
<SINTEGER>   ::= <sum> <integer> | <integer>
<INDID>      ::= <VARID> "&" <VARID> | <VARID> <DL> <VARID> "&"
<VARID>      ::= <VARID> | <VARID>
<ATOM>       ::= <FUNC> | "(" <EXPR> ")" | <VARID>
<VARID>      ::= <VarID>
<FUNC>       ::= <Ufunc> "(" <EXPR> ")" | <Integral> "(" <EXPR> ":" <VARID>
<IN> "[" <VARID> "," <VARID> "]" | <Root> "(" <EXPR> ")" |
"Diff" "(" <EXPR> "," <EXPR> ")" | "diff" "(" <EXPR> "," <EXPR>
" )"
<SEL>        ::= <Selection> "(" <SEL> "," <INDID> "," <INDID> ")" | <ATOM>
<Ufunc>      ::= "inv" | "sqrt" | "exp" | "log" | "sin" | "cos" | "tan" | "asin" | "acos" |
"atan" | "sign" | "neg" | "abs" | "min" | "max"

```

3.1 Rail road diagram (Syntax diagram)

When the BNF of a syntax is available it is also possible to present it graphically using syntax diagrams (or railroad diagrams). The syntax diagrams are a graphical alternative to BNF. Below the syntax diagrams for the same syntax presented in Backus-Naur form above is shown.





4 Compiler

"A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses"[6]. Compilers are a type of translator that support digital devices, primarily computer. The FORTRAN team is generally credited for introducing the first complete compiler in 1957. This compiler was the first ever to produce assembly code comparable to the hand coded code from a high level language. The name compiler is mainly used about programs that translate high-level languages into low-level languages to create an executable program. However, there exist many different types of compilers. In this project a compiler which translate system equations, with it's own defined syntax, into Matlab code has been developed. The compiler translates the language expressed in BNF and rail road diagram in section 3. The compiler is written in Lua and the *lexer* and *parser* is created by the usage of LPeg, which is a pattern-matching library for Lua, based on Parsing Expression Grammars (PEGs)². The entire code for the compiler is presented in appendix B. An example of the source language (the language which is translated) is given i appendix C and the resulting translation into the target language is presented in appendix D. The compiler consists of different parts/operations including lexical analysis (*lexer*), parsing and code generation. These three parts/operations will be introduced in this section.

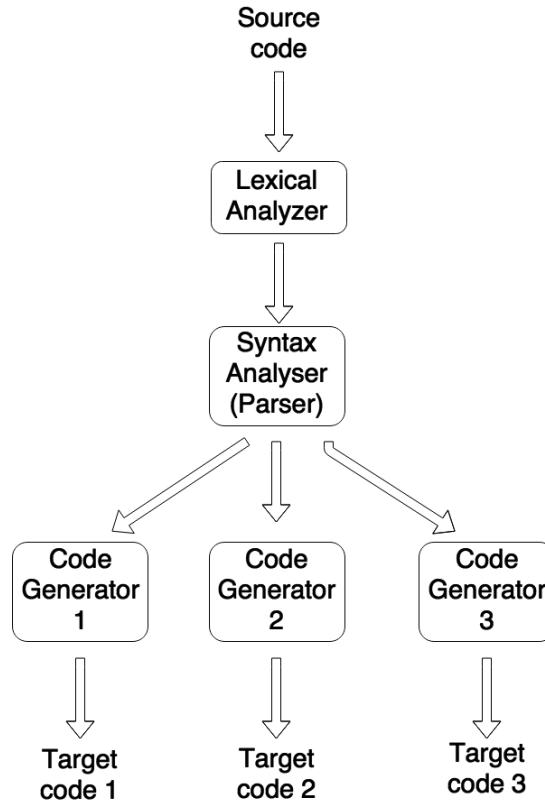


Figure 1: Diagram of a single-language multi-target compiler without any code optimization.

²Parsing Expression grammar is a type of analytical formal grammar. It is closely related to context free grammar, but unlike CFG PEGs cannot be ambiguous. This means that if one string parses, it has exactly one valid parse tree/meaning.

4.1 Lexical analysis (*lexer*)

Lexical analysis breaks the source code text into a sequence of small pieces called lexical tokens[1]. The lexer defines which *building blocks* (tokens) the language is composed of. The set of token categories varies in different programming languages, but common token categories is keywords, separators and operators. The different tokens in the lexer is typically defined by a regular language, such as **regular expressions**. By defining these tokens the parsing can be done at the token level instead of the character level, meaning that the parser scans for tokens. For example a calculator program would look at an input such as "10 * (3 + 2)^3" and split it into the following tokens: 10, *, (, 3, +, 2, ^, 3. Each of these tokens is a meaningful symbol in the context of an arithmetic expression. The lexer in the calculator program will contain rules to tell it that the characters (,), ^, *, + mark the start of a new token, so meaningless tokens like "10*" or "(3" will not be generated.

4.1.1 Lexer in the compiler

As mentioned the compiler is written in Lua, and a package called LPeg is used. In LPeg it is possible to use pattern (called LPeg patterns) which are very similar to regular expressions[3]. The usage of these patterns are the same as for the regular expressions, but the syntax of defining them are a little different. It is also possible to write regex in LPeg using the re module. By doing so, the LPeg will use LPeg to parse the regex into LPeg patterns[4]. To be able to understand the content in the lexer some basic parts of the syntax in LPeg is needed to know. The parts of LPeg that is used in the compiler is presented in table 2.

LPEG	Description	re python
lpeg.P(<i>string</i>)	Creates a pattern that matches <i>string</i> literally	re.compile(<i>string</i>)
lpeg.S(<i>string</i>)	Creates a pattern that matches any character in <i>string</i>	re.compile([<i>string</i>])
lpeg.R("xy")	Creates a pattern that matches any character between x and y	re.compile([x-y])
patt1 * patt2	Matches patt1 followed by patt2	Not introduced
patt1 + patt2	Matches patt1 or patt2	patt1 patt2
patt^n	Matches at least n repetitions of patt	patt{n,}
patt^-n	Matches at most n repetitions of patt	patt{,n}

Table 2: The operations from LPeg that is used in the compiler presented in this report[4].

```
local L = require 'lpeg'

spc = L.S(" \t\n")^0
Ufunc      = spc * L.P("inv") + L.P("sqrt") + L.P("exp") + L.P("log") +
↳ L.P("sin") + L.P("cos") + L.P("tan") + L.P("asin") + L.P("acos") +
↳ L.P("atan") + L.P("sign") + L.P("neg") + L.P("abs") + L.P("min") +
↳ L.P("max") * spc
Root       = spc * L.P("root") * spc
Left       = spc * L.P("<") * spc
Right      = spc * L.P(">") * spc
IN         = spc * L.P("in") * spc
Integral   = spc * L.P("integral") * spc
Selection  = spc * L.P("select") * spc
Instantiate = spc * L.P("set") * spc
```

```

VarID = spc * L.R("az", "AZ", "__", "09")^0 * spc
sum = spc * L.S('+ -') * spc
power = spc * L.P("^") * spc
dot = spc * L.P(".") * spc
DL = spc * L.P("|") * spc
KR = spc * L.P("'") * spc
par = spc * L.P("(") + L.P(")") * spc
real = spc * L.C(
    L.P('-')^-1 *
    L.R('09')^0 *
    ( L.P('.') *
      L.R('09')^0
    )^-1 ) * spc /
tonumber
integer = L.R("09")^1

```

In the code snippet above the lexer is presented. In the entire content of the source language is defined as tokens. The language is made out of a total of 18 tokens, where each token can be identified by different strings. *spc* will match zero or more whitespaces (here defined as space, tab and newline), *Ufunc* will match zero or more whitespaces followed by either *inv*, *sqrt*, *exp*,... or *max* followed by a new sequence of zero or more whitespaces. *Ufunc* defines which mathematical functions that are available/defined in the source language. If for example "cosh" is part of the source language, a "L.P("cosh")" would have to be added in the *Ufunc* declaration for the parser to be able to capture it as a *Ufunc*. *VarID* determines which variable identifications that are allowed in the source could. In the source language, which this lexer represents, the variable name can be any combination of lower and upper letters, numbers as well as "_".

4.2 Parsing

Parsing syntax analysis or syntactic analysis is the process of analysing a string of symbols, either in natural language or in computer languages, conforming to the rules of a formal grammar. The term parsing comes from Latin, and means part (of speech). Parsing has a slightly different meaning in branches of linguistics and computer science, but the objective is more or less the same. A parser is used to understand the meaning of a sentence/word/string. The parser will parse the token sequence to identify the syntactic structure of the program. The parser is created by following the syntax of the source language (the language that is to be parsed) which is presented in backus naur form. The first objective of the parser is to check that the tokens form a allowable expression. This part is done with reference to the grammar in BNF which recursively defines components that can make up an expression and the order in which they must appear. The second objective is to analyse the expression just validated and take the appropriate action. In the case of a calculator the action is to evaluate the expression, whilst a compiler would generate some code. There are different types of parser, but both of the parsers presented in this report is based on **Recursive descent parsing**, which is a top down parsing technique[2].

4.2.1 Example: Calculator

The parser used to parse the system equations presented in C is quite complex, and is therefore not the best example to show the concept of a parser. As an example, a parser for a simple calculator that only knows the operations "+", "-", "/" and "*" is presented as an example. In

the code below the *lexer* consists of the tokens *spc* and *number*. It would also be possible to include the operators ("+-/*") in the lexer, but in this example they are just added directly in the parser. The parser in this example is the part starting with "expr = lpeg.P{", and is based on recursion. The "EXPR" indicates that the first step in the recursion starts at EXPR. lpeg.V("TERM") indicates that "TERM" is a nonterminal, meaning that the grammar still does not exist when this function is evaluated. This is in line with how non terminals are defined in BNF. lpeg.C is type of capture. A capture is a pattern that produces values according to what it matches. These patterns are the ones that are sent from the parser to the evaluator. The /eval at the end of the lines tells the parser where the captured patterns should be sent.

```
lpeg = require 'lpeg'

-- Evaluator
function eval(num1, operator, num2)
    if operator == '+' then
        return num1 + num2
    elseif operator == '-' then
        return num1 - num2
    elseif operator == '*' then
        return num1 * num2
    elseif operator == '/' then
        return num1 / num2
    else
        return num1
    end
end

-- Lexer
spc = lpeg.S(" \t\n")^0

number = spc * lpeg.C(
    lpeg.P('-')^-1 *
    lpeg.R('09')^0 *
    ( lpeg.P('.') *
      lpeg.R('09')^0
    )^-1 ) * spc /
tonumber

-- Parser
expr = lpeg.P{
    "EXPR"; --Start recursion at EXPR
    EXPR = ( lpeg.V("TERM") * lpeg.C( lpeg.S('+-') ) * lpeg.V("EXPR") +
            lpeg.V("TERM") ) / eval,
    TERM = ( lpeg.V("FACT") * lpeg.C( lpeg.S('/*') ) * lpeg.V("TERM") +
            lpeg.V("FACT") ) / eval,
    FACT = ( spc * "(" * lpeg.V("EXPR") * ")" * spc +
            number ) / eval
}
```

To describe what the parser does, it is easiest to use an example. Lets take the example "1

+ 2 * (4 - 3)". The parser will start at EXPR and will check what "1" is. It will go through TERM -> FACT -> number. Since the next token it will meet is "+" the parser will recognize that the sequence will correspond to TERM + EXPR. How the parser goes through the string recursively is presented below.

```

EXPR1  = TERM1 + EXPR2
TERM1   = FACT1
FACT1   = number1 = 1 (captured)
EXPR2   = TERM2
TERM2   = FACT2 * TERM3
FACT2   = number2 = 2 (captured)
TERM3   = FACT3
FACT3   = EXPR3
EXPR3   = TERM4 - EXPR4
TERM4   = FACT4
FACT4   = number3 = 5 (captured)
EXPR4   = TERM5
TERM5   = FACT5
FACT5   = number4 = 3 (captured)

```

Each time the parser goes through the different non terminals (EXPR, TERM and FACT) the captured tokens is sent to the evaluator, eval(num1, operator, num2), the parts which is deepest in the recursion will be evaluated first. By doing so everything inside parenthesis will be prioritized before multiplication and division, whilst addition and subtraction will have lowest priority. The order of evaluation is presented below. Notice how the non terminal to be parsed first is the first to be evaluated and the end expression is evaluated at the end.

```

FACT5   = eval(3, , ) = 3
TERM4   = eval(3, , ) = 3
EXPR4   = eval(3, , ) = 3
FACT4   = eval(5, , ) = 5
TERM4   = eval(5, , ) = 5
EXPR3   = eval(5,-,3) = 2
FACT3   = eval(2, , ) = 2
TERM3   = eval(2, , ) = 2
FACT2   = eval(2, , ) = 2
TERM2   = eval(2,*,2) = 4
EXPR2   = eval(4, , ) = 4
FACT1   = eval(1, , ) = 1
TERM1   = eval(1, , ) = 1
EXPR1   = eval(1,+,4) = 5

```

4.2.2 Parser in the compiler

As mentioned earlier, the parser for the system equation language is quite more complex than the parser for the calculator. Even though it is more complex, writing the parser is not that more difficult than for the calculator as long as the Backus Naur Form is available. The BNF, as explained before, defines how the syntax of the language is, and together with the tokens defined in the lexer determines every allowed sequence of characters in the language. Therefore

writing the parser is quite straightforward, assuming prior knowledge about the syntax of the language the parser is to be written. There are some things that has to be kept in mind. For example, in the lexer a VarID can be any combination of letters, `_` and numbers, so by the definitions in the lexer "root", "sin", "set", "integral", and so on, can be recognized as both VarID and Root, Ufunc, Selection or Instantiate. In the system equation language these strings are reserved names. To make sure that such strings does not get parsed as a VarID, the order of the recursion in the parser is important. The recursion has to be in such an order that all the other tokens "gets" the possibility to get matched before VarID. Another thing to notice about this parser is that the captured tokens are sent to different functions depending on the sequence. In addition the captures is sent as a list instead of a single captures by the use of `L.Ct(patt)` which collects every capture in `patt` into a list.

```
myGrammar = L.P{
  "START";
  START = (
    L.Ct(Instantiate * "(" * L.V("ATOM") * ")")/instant
    + L.V("EXPR")
  ),
  EXPR = (
    L.Ct(L.V("TERM") * L.C(sum) * L.V("EXPR"))/add
    + L.V("SEL")
  ),
  TERM = (
    L.Ct("(" * L.V("EXPR") * ")") /group
    + L.Ct(L.V("FACT") * dot * L.V("EXPR"))/ExpandProduct
    + L.Ct(L.V("FACT") * KR * L.V("EXPR"))/KhatRiRao
    + L.Ct(L.V("FACT") * DL * L.V("INDID") * DL *
      ↪ L.V("EXPR"))/ReduceProduct
    + L.V("FACT")
  ),
  FACT = (
    L.Ct(L.V("SATOM") * power * "{" * L.V("SINTEGER") * "}")/Power
    + L.Ct(L.V("SATOM") * power * "{" * L.V("SATOM") * "}")/Power
    + L.V("SATOM")
  ),
  SATOM = (
    L.Ct(L.C(sum) * L.V("ATOM"))/UFunc
    + L.V("ATOM")
  ),
  SINTEGER = (
    L.Ct(L.C(sum) * L.C(integer))/SintegerFunc
    + L.C(integer)),
  INDID = (
    L.Ct(L.C(VarID) * L.C("&") * L.C(VarID))/IndidFunc
    + L.Ct(L.C(VarID) * L.C(DL) * (L.C(VarID) * L.C("&") *
      ↪ L.C(VarID)))/IndidFuncDL
    + L.C(VarID)
  ),
  ATOM = (
    L.V("FUNC")
  )
}
```

```

        + L.V("VARID")
    ),
VARID = (
    L.C(VarID)
),
FUNC = (
    L.Ct(L.C(Ufunc) * "(" * L.V("EXPR") * ")")/UFunc
    + L.Ct(Integral * spc * "(" * spc * L.V("EXPR") * spc * "::" *
        ↪ spc * L.V("VARID") * IN * "[" * L.V("VARID") * "," *
        ↪ *L.V("VARID") * "]" * ")")/IntegralFunc
    + L.Ct(L.C(Root) * "(" * L.V("EXPR") * ")")/Implicit
    + L.Ct("Diff" * spc * "(" * L.V("EXPR") * "," * L.V("EXPR") *
        ↪ ")")/TotDifferential
    + L.Ct("diff" * spc * "(" * L.V("EXPR") * "," * L.V("EXPR") *
        ↪ ")")/ParDifferential
),
SEL = (
    L.Ct(Selection * "(" * L.V("SEL") * "," * L.V("INDID") * "," *
        ↪ L.V("INDID") * ")")/SelectionFunc
    + L.V("TERM")
    + L.V("ATOM")
)
}

```

Since the parser consists of several recursive levels there is many evaluation that are done to parse a string. To get a feeling about how the parser goes through the string a example is shown in the table below. The example shown is how the input string, "**half.(abs(Fm_vv) + (Fm_vv.D_v))**", is parsed and sent to the code generation function which will result in the output string; **ExpandProduct(half,(add(abs(Fm_vv),(ExpandProduct(Fm_vv,D_v))))).** The table presents what each part of the string is recognized as, and which function is called and with what the input to the functions are. The function evaluation will be executed in the opposite order of how it is presented in the table. This is due to recursion in the parser.

Expression	Function called
$\underbrace{\text{half} . (\text{abs}(\text{Fm_vv}) + (\text{Fm_vv.D_v}))}_{\text{half} \quad \text{dot} \quad \text{EXPR}}$	ExpandProduct(FACT, EXPR)
$\underbrace{\text{half}}_{\text{half}} \underbrace{.}_{\text{dot}} \underbrace{(\text{abs}(\text{Fm_vv}) + (\text{Fm_vv.D_v}))}_{\text{EXPR}}$	Just captured
$\underbrace{\text{abs}(\text{Fm_vv})}_{\text{TERM}} \underbrace{+}_{\text{sum}} \underbrace{(\text{Fm_vv.D_v})}_{\text{EXPR}}$	add(TERM, EXPR)
$\underbrace{\text{abs}}_{\text{Ufunc}} \underbrace{(\text{Fm_vv})}_{\text{EXPR}}$	abs(EXPR)
$\underbrace{\text{Fm_vv}}_{\text{Fm_vv}}$	Just captured
$\underbrace{\text{Fm_vv}}_{\text{FACT}} \underbrace{.}_{\text{dot}} \underbrace{\text{D_v}}_{\text{EXPR}}$	ExpandProduct(FACT, EXPR)
$\underbrace{\text{Fm_vv}}_{\text{Fm_vv}}$	Just captured
$\underbrace{\text{D_v}}_{\text{D_v}}$	Just captured
TOTAL	ExpandProduct(half, add(abs(Fm_vv),ExpandProduct(Fm_vv, D_v)))

4.3 Code generation

The code generation part of the compiler is where the new code is created. As mentioned, the tokens that are identified by the parser is sent to the function that corresponds to the sequence the tokens comes in. The form of the output language is determined by the output of these functions. The fact that the output it produced by functions outside of the parser opens up for multiple output languages. By reusing the same parser, and rewriting the code generation functions, it is simple to implement a second and third output language. All of the code generating functions is included in the entire compiler in appendix B. As an example, one of the functions used for code generation in the compiler is ExpandProduct(myList):

```
function ExpandProduct(myList)
    return "ExpandProduct".. myList[1] .. "," .. myList[2] .. "
end
```

This will return the output "ExpandProduct(A,B)", assuming myList[1] = A and myList[2] = B. The output language which is implemented in the compiler is Matlab, but if the output is needed in another language, or for some reason the function (in the target language) is implemented with another name, only corrections in the functions like "ExpandProduct(myList)" in Lua is needed. As of now the output code is not executable in Matlab as the functions are not implemented in Matlab. To make the resulting script to run either the functions has to be implemented, or for some of the functions the output can be altered such that built-in functions are used instead.

5 Closing Remarks

The area of formal language theory opens up for solving problems which is quite complex to solve in other ways. The application of regular expression and compilers/parsers within chemical engineering might not be that obvious, but it is applicable for any problem where "find and search" is needed. At the time Big Data is a hot topic, and regular expression can be applied to search and retrieve the requested data. In this report only a introduction to regular expressions is given, but many tutorials makes it possible to learn what is necessary to solve a given problem. To practice or play with regular expression there are several websites to be used, such as <https://regexr.com/>.

LPeg in Lua is a nice way to get familiar with parsers and compilers. Lua is a compact language which makes the transition to the syntax managable. In addition the LPeg is well documented, and the readability of the language is very good. This makes it easy to understand the example which are available online.

References

- [1] *Compiler Design - Lexical Analysis*. https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm. Accessed: 2017-12-01.
- [2] *Compiler Design - Types of Parsing*. https://www.tutorialspoint.com/compiler_design/compiler_design_types_of_parsing.htm. Accessed: 2017-12-01.
- [3] *LPEG and regular expressions*. <http://www.gammon.com.au/lpeg#lpeg>. Accessed: 2017-12-01.
- [4] *Parsing Expression Grammars For Lua*. <http://www.inf.puc-rio.br/~roberto/lpeg/>. Accessed: 2017-12-01.
- [5] *Regular expression! Learning*. <https://ryanstutorials.net/regular-expressions-tutorial/#learning>. Accessed: 2017-11-10.
- [6] *What is compiler?* <http://whatis.techtarget.com/definition/compiler>. Accessed: 2017-12-01.

A Regex examples: python re module

```
import re #Importing the regex module in python

# Regular expression used to match all a's in a sentence.
a = re.compile(r'a')
matches = a.findall('A computer once beat me at chess,' +
'but it was no match for me at kick boxing.') #Finds all matches
iterator = a.finditer('A computer once beat me at chess,' +
'but it was no match for me at kick boxing.') #Returns position of matches as an
↪ iterator
# Output
print(matches)
for match in iterator:
    print match.span()

# Dot
a = re.compile(r'a.e')
matches = a.findall('Software is like entropy: It is difficult to grasp,' +
'weighs nothing, and obeys the Second Law of Thermodynamics;' +
'i.e., it always increases.') #Finds all matches
iterator = a.finditer('Software is like entropy: It is difficult to grasp,' +
'weighs nothing, and obeys the Second Law of Thermodynamics;' +
'i.e., it always increases.') #Returns position of matches as an iterator
# Output
print(matches)
for match in iterator:
    print match.span()

# Ranges
a = re.compile(r'er[ey]')
matches = a.findall('There is only one problem with common sense;' +
'it\'s not very common.') #Finds all matches
iterator = a.finditer('There is only one problem with common sense;' +
'it\'s not very common.') #Returns position of matches as an iterator
# Output
print(matches)
for match in iterator:
    print match.span()

# Multipliers (*)
a = re.compile(r'et+')
matches = a.findall('Getting information off the Internet is like taking' +
'a drink from a fire hydrant.') #Finds all matches
iterator = a.finditer('Getting information off the Internet is like taking' +
'a drink from a fire hydrant.') #Returns position of matches as an iterator
# Output
```

```

print(matches)
for match in iterator:
    print match.span()

# Greedy when using dot and multiplier
a = re.compile(r'a.*e')
matches = a.findall('The greatest enemy of knowledge is not ignorance, it is the
↳ illusion of knowledge.') #Finds all matches
iterator = a.finditer('The greatest enemy of knowledge is not ignorance, it is
↳ the illusion of knowledge.') #Returns position of matches as an iterator
# Output
print(matches)
for match in iterator:
    print match.span()

# Fixed the greedyness of the regex
a = re.compile(r'a.*?e')
matches = a.findall('The greatest enemy of knowledge is not ignorance, '+
'it is the illusion of knowledge.') #Finds all matches
iterator = a.finditer('The greatest enemy of knowledge is not ignorance, '+
'it is the illusion of knowledge.') #Returns position of matches as an iterator
# Output
print(matches)
for match in iterator:
    print match.span()

#Matching every words
a = re.compile(r'\w+')
matches = a.findall('Imagination is more important than knowledge.' +
'For knowledge is limited, whereas imagination embraces the entire world,' +
'stimulating progress, giving birth to evolution.') #Finds all matches
iterator = a.finditer('Imagination is more important than knowledge.' +
'For knowledge is limited, whereas imagination embraces the entire world,' +
'stimulating progress, giving birth to evolution.') #Returns position of matches
↳ as an iterator
# Output
print(matches)
for match in iterator:
    print match.span()

#Matching word at beginning and end of line
a = re.compile(r'^\w+|\w+[\.\!?\]$')
matches = a.findall('Your most unhappy customers are your greatest' +
'source of learning.') #Finds all matches
iterator = a.finditer('Your most unhappy customers are your greatest' +
'source of learning.') #Returns position of matches as an iterator
# Output
print(matches)
for match in iterator:

```



```

    print match.span()

# Matching every word beginning on lower case i
a = re.compile(r'\bi\w+\b')
matches = a.findall('The best way to get accurate information on Usenet'+
'is to post something wrong and wait for corrections.') #Finds all matches
iterator = a.finditer('The best way to get accurate information on Usenet'+
'is to post something wrong and wait for corrections.')
#Returns position of matches as an iterator
# Output
print(matches)
for match in iterator:
    print match.span()

# Matching IP addresses
a = re.compile(r"(\d{1,3}\.){3}\d{1,3}")
matches = a.findall('This is an IP address 210.1.41.103,'+
'but this is not 112.3332.124.1') #Finds all matches
iterator = a.finditer('This is an IP address 210.1.41.103,'+
'but this is not 112.3332.124.1') #Returns position of matches as an iterator
# Output
print(matches)
for match in iterator:
    print match.span()

# Matching John and Michael Smith
a = re.compile(r"(?:John|Michael) Smith")
matches = a.findall('We want to match John Smith and Michael Smith,' +
'but not Paul Smith.') #Finds all matches
iterator = a.finditer('We want to match John Smith and Michael Smith,' +
'but not Paul Smith.') #Returns position of matches as an iterator
# Output
print(matches)
for match in iterator:
    print match.span()

# Matching NTNU email addresses
a = re.compile(r"[a-z]+@(?:stud\.)?ntnu\.no")
matches = a.findall('mariusre@stud.ntnu.no; mariusre@ntnu.no;'+
'marius.reed@lyse.net;john.smith@gmail.com') #Finds all matches
iterator = a.finditer('mariusre@stud.ntnu.no; mariusre@ntnu.no;'+
'marius.reed@lyse.net;john.smith@gmail.com') #Returns position of matches as an
↪ iterator
# Output
print(matches)
for match in iterator:
    print match.span()

```

```

# Matching person numbers for people born in the 90's
a = re.compile(r"\d{9}9\d")
matches = a.findall('    Paul Smith  12345190192 \n' +
    'John Smith  31290221172 \n' +
    'Michael Smith  31231170599') #Finds all matches
iterator = a.finditer('    Paul Smith  12345190192 \n' +
    'John Smith  31290221172 \n' +
    'Michael Smith  31231170599') #Returns position of matches as an iterator
# Output
print(matches)
for match in iterator:
    print match.span()

```

B LPEG: Compiler

```
local L = require 'lpeg'

-- Lexer
spc = L.S(" \t\n")^0
Ufunc = spc * L.P("inv") + L.P("sqrt") + L.P("exp") + L.P("log") +
  ↳ L.P("sin")
  + L.P("cos") + L.P("tan") + L.P("asin") + L.P("acos")
  + L.P("atan") + L.P("sign") + L.P("neg") + L.P("abs")
  + L.P("min") + L.P("max") * spc
Root = spc * L.P("root") * spc
Left = spc * L.P("<") * spc
Right = spc * L.P(">") * spc
IN = spc * L.P("in") * spc
Integral = spc * L.P("integral") * spc
Selection = spc * L.P("select") * spc
Instantiate = spc * L.P("set") * spc
VarID = spc * L.R("az", "AZ", "__", "09")^0 * spc
sum = spc * L.S('+-') * spc
power = spc * L.P("^") * spc
dot = spc * L.P(".") * spc
DL = spc * L.P("|") * spc
KR = spc * L.P("'") * spc
par = spc * L.P("(") + L.P(")") * spc
real = spc * L.C(
  L.P('-')^-1 *
  L.R('09')^0 *
  ( L.P('.') *
    L.R('09')^0
  )^-1 ) * spc /
  tonumber
integer = L.R("09")^1

-- Code Generation
function instant(myList)
  return ("Instantiate(" .. myList[1] .. ")")
end

function add(myList)
  if myList[2]:match( "%s*(.)%s*$" ) == "+" then -- Remove whitespace
    return "add(" .. myList[1] .. "," .. myList[3] .. ")"
  else
    return "subtract(" .. myList[1] .. "," .. myList[3] .. ")"
  end
end

function ExpandProduct(myList)
```

```

        return "ExpandProduct".. myList[1] .. "," .. myList[2] .. ")"
    end

function KhatriRao(myList)
    return "KhatriRao".. myList[1] .. "," .. myList[2] .. ")"
end

function ReduceProduct(myList)
    return "reduceProduct".. myList[1] .. "," .. myList[3] .. "," ..
        ↪ myList[2].."")
end

function group(myList)
    -- Solving parenthesis
    return "(" .. myList[1] .. ")"
end

function Power(myList)
    return "Power (" .. "^" .. "," .. myList[1] .. "," .. myList[2] .. ")"
end

function UFunc(myList)
    return myList[1] .. "(" .. myList[2] .. ")"
end

function SintegerFunc(myList)
    return myList[1].. myList [2]
end

function IndidFunc(myList)
    return myList[1] .. " " .. myList[2] .. " " .. myList[3]
end

function IndidFuncDL(myList)
    return myList[1] .. myList[2] " " .. myList[3] .. " " .. myList[4]
end

function IntegralFunc(myList)
    return "Integral(" .. myList[1] .. "," .. myList[2] .. "," .. myList[3]
        ↪ .. "," .. myList[4] .. ")"
end

function Implicit(myList)
    return "Implicit".. myList[1] .. "," .. myList[2] .. ")"
end

function TotDifferential(myList)
    return "TotDifferential(" .. myList[1] .. "," .. myList[2] .. ")"
end

```

```

function ParDifferential(myList)
  return "ParDifferential(" .. myList[1] .. "," .. myList[2] .. ")"
end

function SelectionFunc(myList)
  return "Selection(".. myList[1] .. "," .. myList[2] .. "," .. myList[3]
    ↪ .. ")"
end

-- Parser
myGrammar = L.P{
  "START";
  START = (
    L.Ct(Instantiate * "(" * L.V("ATOM") * ")")/instant
    + L.V("EXPR")
  ),
  EXPR = (
    L.Ct(L.V("TERM") * L.C(sum) * L.V("EXPR"))/add
    + L.V("SEL")
  ),
  TERM = (
    L.Ct("(" * L.V("EXPR") * ")") /group
    + L.Ct(L.V("FACT") * dot * L.V("EXPR"))/ExpandProduct
    + L.Ct(L.V("FACT") * KR * L.V("EXPR"))/KhatRao
    + L.Ct(L.V("FACT") * DL * L.V("INDID") * DL *
      ↪ L.V("EXPR"))/ReduceProduct
    + L.V("FACT")
  ),
  FACT = (
    L.Ct(L.V("SATOM") * power * "{" * L.V("SINTEGER") * "}")/Power
    + L.Ct(L.V("SATOM") * power * "{" * L.V("SATOM") * "}")/Power
    + L.V("SATOM")
  ),
  SATOM = (
    L.Ct(L.C(sum) * L.V("ATOM"))/UFunc
    + L.V("ATOM")
  ),
  SINTEGER = (
    L.Ct(L.C(sum) * L.C(integer))/SintegerFunc
    + L.C(integer)),
  INDID = (
    L.Ct(L.C(VarID) * L.C("&") * L.C(VarID))/IndidFunc
    + L.Ct(L.C(VarID) * L.C(DL) * (L.C(VarID) * L.C("&") *
      ↪ L.C(VarID)))/IndidFuncDL
    + L.C(VarID)
  ),
  ATOM = (
    L.V("FUNC")

```

```

        + L.V("VARID")
    ),
    VARID = (
        L.C(VarID)
    ),
    FUNC = (
        L.Ct(L.C(Ufunc) * "(" * L.V("EXPR") * ")")/UFunc
        + L.Ct(Integral * spc * "(" * spc * L.V("EXPR") * spc * "::" *
            ↪ spc * L.V("VARID") * IN * "[" * L.V("VARID") * "," *
            ↪ *L.V("VARID") * "]" * ")")/IntegralFunc
        + L.Ct(L.C(Root) * "(" * L.V("EXPR") * ")")/Implicit
        + L.Ct("Diff" * spc * "(" * L.V("EXPR") * "," * L.V("EXPR") *
            ↪ ")")/TotDifferential
        + L.Ct("diff" * spc * "(" * L.V("EXPR") * "," * L.V("EXPR") *
            ↪ ")")/ParDifferential
    ),
    SEL = (
        L.Ct(Selection * "(" * L.V("SEL") * "," * L.V("INDID") * "," *
            ↪ L.V("INDID") * ")")/SelectionFunc
        + L.V("TERM")
        + L.V("ATOM")
    )
}

-- Some file handling
function repl(file)
    file = io.input("/Users/MariusReed/Min Sky/NTNU/Kjemi/5.Klasse/Advanced
        ↪ Simulation/mariusLua/equations.cfg")
    io.output("//Users/MariusReed/Min Sky/NTNU/Kjemi/5.Klasse/Advanced
        ↪ Simulation/parsedSystem.m")
    parser = myGrammar
    i = 1
    for line in file:lines() do
        if string.sub(line,1,3) == "lhs" then
            lhs = string.sub(line,7)
        elseif string.sub(line,1,3) == "rhs" then
            rhs = string.sub(line,7)
        end
        if i == 6 then
            equation = lhs .. " = " .. parser:match(rhs)
            io.write(equation, "\n")
            i = 1
        else
            i = i+1
        end
    end
    equation = lhs .. " = " .. parser:match(rhs)
    io.write(equation, "\n")
end

```

```
repl()
```

C Input to compiler

```
[0]
lhs = Pn_NA
incidence_list = ['Pn_A', 'Pn_N']
rhs = Pn_N|species|Pn_A
layer = ontology_physical

[1]
lhs = p
incidence_list = ['U', 'V']
rhs = -diff(U,V)
layer = ontology_physical

[2]
lhs = T
incidence_list = ['Sn', 'U']
rhs = diff(U,Sn)
layer = ontology_physical

[3]
lhs = mu
incidence_list = ['U', 'n']
rhs = diff(U,n)
layer = ontology_physical

[4]
lhs = Cv
incidence_list = ['T', 'U']
rhs = diff(U,T)
layer = ontology_physical

[5]
lhs = Cp
incidence_list = ['H', 'T']
rhs = diff(H,T)
layer = ontology_physical

[6]
lhs = Sn
incidence_list = ['H', 'T', 'mu', 'n']
rhs = inv(T).H-inv(T).(mu|species|n)
layer = ontology_physical

[7]
lhs = H
```

```

incidence_list = ['U', 'V', 'p']
rhs = U+p.V
layer = ontology_physical

[8]
lhs = lambda_ns
incidence_list = ['Pn_N', 'lambda_s']
rhs = lambda_s|species|Pn_N
layer = ontology_physical

[9]
lhs = m
incidence_list = ['lambda_ns', 'n']
rhs = lambda_ns |species| n
layer = ontology_physical

[10]
lhs = Fm_vv
incidence_list = ['Fm']
rhs = select(select(Fm,node,node_v),arc,arc_v)
layer = ontology_physical

[11]
lhs = Fm_dd
incidence_list = ['Fm']
rhs = select(select(Fm,node,node_d),arc,arc_d)
layer = ontology_physical

[12]
lhs = Fn
incidence_list = ['Fm', 'Pn_NA']
rhs = Fm : Pn_NA
layer = ontology_physical

[14]
lhs = Pn_Nc
incidence_list = ['Pn_N']
rhs = select(Pn_N,node,node_conversion)
layer = ontology_physical

[16]
lhs = Nsn
incidence_list = ['Ns', 'Pn_Nc']
rhs = Ns |species| Pn_Nc
layer = ontology_physical

[17]
lhs = p_ref
incidence_list = ['p']

```



```

rhs = set(p)
layer = ontology_physical

[18]
lhs = T_ref
incidence_list = ['T']
rhs = set(T)
layer = ontology_physical

[19]
lhs = mu_ref
incidence_list = ['mu']
rhs = set(mu)
layer = ontology_physical

[20]
lhs = pi_T
incidence_list = ['T', 'T_ref']
rhs = inv(T_ref).T
layer = ontology_physical

[21]
lhs = pi_p
incidence_list = ['p', 'p_ref']
rhs = inv(p_ref).p
layer = ontology_physical

[22]
lhs = pi_mu
incidence_list = ['mu', 'mu_ref']
rhs = inv(mu_ref).mu
layer = ontology_physical

[23]
lhs = c
incidence_list = ['V', 'n']
rhs = inv(V):n
layer = ontology_physical

[24]
lhs = pi_pv
incidence_list = ['pi_p']
rhs = select(pi_p,node, node_v)
layer = ontology_physical

[25]
lhs = pi_mud
incidence_list = ['pi_mu']
rhs = select(pi_mu, node, node_d)

```

```

layer = ontology_physical

[26]
lhs = D_v
incidence_list = ['Fm_vv', 'pi_pv']
rhs = sign(Fm_vv | node_v | pi_pv)
layer = ontology_physical

[27]
lhs = B_v
incidence_list = ['B']
rhs = select(B, arc, arc_v)
layer = ontology_physical

[28]
lhs = B_d
incidence_list = ['B']
rhs = select(B, arc, arc_d)
layer = ontology_physical

[29]
lhs = fVp
incidence_list = ['B_v', 'Fm_vv', 'K_v', 'pi_pv']
rhs = K_v.B_v.(Fm_vv | node_v | pi_pv)
layer = ontology_physical

[30]
lhs = fVp
incidence_list = ['B_v', 'D_v', 'Fm_vv', 'K_v', 'pi_pv']
rhs = D_v.K_v.B_v.sqrt(abs(Fm_vv | node_v | pi_pv))
layer = ontology_physical

[31]
lhs = Nnrs
incidence_list = ['Nsn', 'Pc_rnr']
rhs = Pc_rnr | conversion | Nsn
layer = ontology_physical

[32]
lhs = Pn_vv
incidence_list = ['Pn_NA']
rhs = select(select(Pn_NA, node, node_v), arc, arc_v)
layer = ontology_physical

[33]
lhs = s_v
incidence_list = ['D_v', 'Fm_vv', 'half']
rhs = half.(abs(Fm_vv) + (Fm_vv.D_v))
layer = ontology_physical

```

```

[34]
lhs = cv
incidence_list = ['c']
rhs = select(c,node,node_v)
layer = ontology_physical

[35]
lhs = ca
incidence_list = ['Pn_vv', 'cv', 's_v']
rhs = cv|node_v & species|(s_v:Pn_vv)
layer = ontology_physical

[36]
lhs = fn_v
incidence_list = ['ca', 'fVp']
rhs = fVp:ca
layer = ontology_physical

[37]
lhs = Pn_dd
incidence_list = ['Pn_NA']
rhs = select(select(Pn_NA,node,node_d),arc,arc_d)
layer = ontology_physical

[38]
lhs = fn_d
incidence_list = ['B_d', 'Fm_dd', 'K_d', 'Pn_dd', 'pi_mud']
rhs = B_d:K_d.(Fm_dd:Pn_dd)|node_d & species|pi_mud
layer = ontology_physical

[39]
lhs = xi_ns
incidence_list = ['Pc_rnr', 'xi']
rhs = xi|conversion|Pc_rnr
layer = ontology_physical

[40]
lhs = tnr
incidence_list = ['Nnrs', 'xi_ns']
rhs = Nnrs|node_conversion & conversion|xi_ns
layer = ontology_physical

[41]
lhs = Fn_v
incidence_list = ['Fn']
rhs = select(Fn,arc,arc_v)
layer = ontology_physical

```

```

[42]
lhs = Fn_d
incidence_list = ['Fn']
rhs = select(Fn,arc,arc_d)
layer = ontology_physical

[43]
lhs = ndiff
incidence_list = ['Fn_d', 'Fn_v', 'fn_d', 'fn_v']
rhs = Fn_v|arc_v & species|fn_v+Fn_d|arc_d & species|fn_d
layer = ontology_physical

[44]
lhs = ndiff
incidence_list = ['Fn_d', 'Fn_v', 'Pnrn', 'fn_d', 'fn_v', 'tnr']
rhs = Fn_v|arc_v & species|fn_v+Fn_d|arc_d & species|fn_d+Pnrn|node_conversion &
  ↪ species|tnr
layer = ontology_physical

[45]
lhs = n
incidence_list = ['ndiff', 't', 't_0', 't_e']
rhs = integral(ndiff :: t in [t_0,t_e])
layer = ontology_physical

[46]
lhs = ntot
incidence_list = ['e_NS', 'n']
rhs = e_NS|species|n
layer = ontology_physical

[47]
lhs = pg
incidence_list = ['p']
rhs = set(p)
layer = ontology_physical_gas

[48]
lhs = pg
incidence_list = ['Rg', 'T', 'V', 'ntot']
rhs = ntot.T.Rg.inv(V)
layer = ontology_physical_gas

[49]
lhs = fu
incidence_list = ['Bx', 'u']
rhs = Bx.u
layer = ontology_control

```

```

[50]
lhs = fx
incidence_list = ['Ax', 'x']
rhs = Ax|node & signal|x
layer = ontology_control

[51]
lhs = xdiff
incidence_list = ['Nu', 'Nx', 'fu', 'fx']
rhs = Nx|arc & signal|fx + Nu|arc & signal|fu
layer = ontology_control

[52]
lhs = fn
incidence_list = ['n', 'ntot']
rhs = inv(ntot):n
layer = ontology_physical

[53]
lhs = pp
incidence_list = ['Rg', 'T', 'V', 'n']
rhs = T.Rg.inv(V):n
layer = ontology_physical_gas

```

D Output from compiler: MATLAB code

```

Pn_NA = reduceProduct(Pn_N,Pn_A,species)
p = -(ParDifferential(U,V))
T = ParDifferential(U,Sn)
mu = ParDifferential(U,n)
Cv = ParDifferential(U,T)
Cp = ParDifferential(H,T)
Sn = ExpandProduct(inv(T),subtract(H,ExpandProduct(inv(T),
↳ (reduceProduct(mu,n,species))))))
H = add(U,ExpandProduct(p,V))
lambda_ns = reduceProduct(lambda_s,Pn_N,species)
m = reduceProduct(lambda_ns ,n,species)
Fm_vv = Selection(Selection(Fm,node,node_v),arc,arc_v)
Fm_dd = Selection(Selection(Fm,node,node_d),arc,arc_d)
Fn = Fm
Pn_Nc = Selection(Pn_N,node,node_conversion)
Nsn = reduceProduct(Ns ,Pn_Nc,species)
p_ref = Instantiate(p)
T_ref = Instantiate(T)
mu_ref = Instantiate(mu)
pi_T = ExpandProduct(inv(T_ref),T)
pi_p = ExpandProduct(inv(p_ref),p)
pi_mu = ExpandProduct(inv(mu_ref),mu)
c = inv(V)

```

```

pi_pv = Selection(pi_p,node, node_v)
pi_mud = Selection(pi_mu, node, node_d)
D_v = sign(reduceProduct(Fm_vv ,pi_pv,node_v))
B_v = Selection(B, arc, arc_v)
B_d = Selection(B, arc, arc_d)
fVp = ExpandProduct(K_v,ExpandProduct(B_v,(reduceProduct(Fm_vv,pi_pv,node_v))))
fVp = ExpandProduct(D_v,ExpandProduct(K_v,ExpandProduct(B_v,sqrt(abs(
  ↪ reduceProduct(Fm_vv,pi_pv,node_v))))))
Nnrs = reduceProduct(Pc_rnr,Nsn,conversion)
Pn_vv = Selection(Selection(Pn_NA,node,node_v),arc,arc_v)
s_v = ExpandProduct(half,(add(abs(Fm_vv),(ExpandProduct(Fm_vv,D_v))))))
cv = Selection(c,node,node_v)
ca = reduceProduct(cv,,node_v & species)
fn_v = fVp
Pn_dd = Selection(Selection(Pn_NA,node,node_d),arc,arc_d)
fn_d = B_d
xi_ns = reduceProduct(xi,Pc_rnr,conversion)
tnr = reduceProduct(Nnrs,xi_ns,node_conversion & conversion)
Fn_v = Selection(Fn,arc,arc_v)
Fn_d = Selection(Fn,arc,arc_d)
ndiff = reduceProduct(Fn_v,add(fn_v,reduceProduct(Fn_d,fn_d,arc_d &
  ↪ species)),arc_v & species)
ndiff = reduceProduct(Fn_v,add(fn_v,reduceProduct(Fn_d,add(fn_d,reduceProduct(
  ↪ Pn_rn,tnr,node_conversion & species)),arc_d & species)),arc_v &
  ↪ species)
n = Integral(ndiff ,t ,t_0,t_e)
ntot = reduceProduct(e_NS,n,species)
pg = Instantiate(p)
pg = ExpandProduct(ntot,ExpandProduct(T,ExpandProduct(Rg,inv(V))))
fu = ExpandProduct(Bx,u)
fx = reduceProduct(Ax,x ,node & signal)
xdiff = reduceProduct(Nx,add(fx ,reduceProduct(Nu,fu,arc & signal)),arc &
  ↪ signal)
fn = inv(ntot)
pp = ExpandProduct(T,ExpandProduct(Rg,inv(V)))

```