Parsing Chemical Formulas and Equations

Anton Antonov
book for Wiley 2009-2010

Introduction

Let us suppose that we want to write a program that balances chemical equations. If we, say, have the unbalanced equation

$C_3 H_8 + O_2 \text{ -> } C O_2 + H_2 O,$

that can be re-written as

$a\, C_3 H_8 + b\, O_2 \text{ -> } c\, C O_2 + d\, H_2 O,$ (1)

we want a program to find the stoichiometric coefficients a, b, c, and d.

Because of the law of conservation of masses, the amount of atoms of an element that appears on the left side of the equation should equal the amount of atoms that appear at right side of the equation. Therefore, the problem of balancing a chemical equation can be seen as a problem of solving a system of linear equations. A basic step of the Chemical Equation Balancer (CEB) is to find how many atoms of each element are on the left side and how many atoms of each element are on the right side. The mathematical background and programming of the CEB are described in the next chapter, "Chemical equation balancing." In this chapter we are going to design and program an interpreter of strings representing chemical formulas. That interpreter, which we will call Chemical Formula Parser (CFP), would automatically recognize how many atoms of each element are in a chemical formula.

Let us explicitly formulate the problem we want to solve using the parser described in this chapter.

Problem formulation

Given a string of the molecule formula of a chemical compound we want to compute various quantitative measures and statistics for the compound's molecule. More precisely, we want to find the total molecule mass, and the percentage of the mass that corresponds to each of the molecule's elements. For example, for the molecule $C_6 H_{12} O_6$ we want to obtain the following data:

Molecule formula: C6H12O6
Molecular mass: 180.156

Element	***Mass %***	***Atomic mass***	***number of atoms***
C	0.400010	12.0107	6
H	0.067138	1.00794	12
O	0.53285	15.9994	6

In order to compute the quantitative measures we need to determine which elements and what number of their atoms are in the compound's molecule.

Approaches

We are going to show two approaches to parsing molecules. The first one uses string matching patterns and it is described in the section "String match parser." Reading only that section would provide enough background to move to the next chapter, "Chemical equation balancing." The second approach is to define in a formal manner the grammar of the chemical molecule notation, and using that grammar to program the concrete steps of the parser. The formal description of the chemical formulas notation would be done using the so called Backus-Naur Form (BNF), a language for grammar specification.

The reason we accentuate the second approach is that in general in engineering disciplines we prefer the use of standard solutions instead of particular (or peculiar) ones. By standard solutions here we mean solutions derived within a well understood theoretical or practical framework of concepts, approaches, and methodologies. Using these approaches and methodologies would, to a degree, guarantee the derivation of the required solution.

The method we follow in this chapter belongs to the theory of compilers, programs that translate code of human readable (or high order) programming languages to machine code which is readily executable. The method has the following steps:

- Given a language describe its grammar in a formal manner using BNF or similar formalism.
- For each rule of syntactical construction (or language pattern) choose a data structure that represents it.
- Write a syntax analyzer (or parser) that from sequences of symbols produces syntactical structures defined by the grammar of the language.
- For each rule of syntactical construction write an interpreting function based on the data structure that represents it.

Because we want the section "String matching parser" to be independent from the rest as much as possible, we are going to discuss first data constructs and interpretation.

The rest of the chapter

Thinks to speak about

Definitions

Lexical analysis

Syntactic analysis

Parser = Lexical analyser + Syntactic analyzer

Semantic analysis and Code generation

Finite automata

WFinite automata

- ### Formula representation with data structures

One way to approach this task is to transform the string that is the chemical formula into a data structure that is suitable for calculating molecular mass and what percentage to that mass is made by which elements.

We can think that a string of a molecule formula is composed of concatenated strings of sub-molecules.

- ### Chemical formulas grammar

```
<element> :: = "H" | "He" | "Li" | "Be" | "B" | "C" | "N" | "O" | "F" | "Ne" | "Na"
| "Mg" | "Al" | "Si" | "P" | "S" | "Cl" | "K" | "Ar" | "Ca" | "Sc" | "Ti" | "V" |
"Cr" | "Mn" | "Fe" | "Ni" | "Co" | "Cu" | "Zn" | "Ga" | "Ge" | "As" | "Se" | "Br" |
"Kr" | "Rb" | "Sr" | "Y" | "Zr" | "Nb" | "Mo" | "Tc" | "Ru" | "Rh" | "Pd" | "Ag" |
"Cd" | "In" | "Sn" | "Sb" | "I" | "Te" | "Xe" | "Cs" | "Ba" | "La" | "Ce" | "Pr" |
"Nd" | "Pm" | "Sm" | "Eu" | "Gd" | "Tb" | "Dy" | "Ho" | "Er" | "Tm" | "Yb" | "Lu" |
"Hf" | "Ta" | "W" | "Re" | "Os" | "Ir" | "Pt" | "Au" | "Hg" | "Tl" | "Pb" | "Bi" |
"Po" | "At" | "Rn" | "Fr" | "Ra" | "Ac" | "Pa" | "Th" | "Np" | "U" | "Am" | "Pu" |
"Cm" | "Bk" | "Cf" | "Es" | "Fm" | "Md" | "No" | "Rf" | "Lr" | "Db" | "Bh" | "Sg" |
"Mt" | "Rg" | "Hs" | "Ds" | "Uub" | "Uut" | "Uuq" | "Uup" | "Uuh" | "Uus" | "Uuo"
```

```
<digit> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

```
<number> ::= <digit>|<digit><number>
```

```
<molecule> ::= <element>|<replicated>|<connected>|"("<molecule>")"
```

```
<replicated> ::= <molecule><number> | <molecule>"_"<number> | Subscript[<molecule>,
<number>]
```

```
<connected> ::= <molecule><molecule>
```

```
<mix> ::= <molecule> | <molecule> "+ hν" | <mix> "+" <molecule>
```

```
<equation> ::= <mix> "->" <mix> | <mix> "=" <mix> | <mix> "⇆" <mix>
```

- **Example: Random molecule creation**

```
<molecule>
<connected>
<molecule><molecule>
<molecule><replicated>
<element><element><number>
CO2
```

- **Example: Random equation creation**

```
<equation>
<mix> -> <mix>
<molecule> + <molecule> -> <molecule> + <molecule>
<molecule><replicated> + <molecule><replicated> -> <molecule><replicated> +
<replicated><molecule>
<element><element><number> + <element><element><number> ->
<element><element><number> + <element><number><element>
CO2 + ClH4 -> CH3 + Cl3O
```

- **Parsing formulas into data structures**
- **Parsing with StringMatchQ**
- ***Other formulations***

The first, most basic step is from given a molecule formula to find which elements and what number of their atoms compose that molecule formula.

These measures can be used for chemical equation balancing that we are going consider in the next chapter ("Chemical Equation Balancing"). In this chapter we are going to design and program an interpreter for strings of chemical formulas.

Chemical formulas can be interpreted as ideograms and sumarization of the chemical reactions they participate in, [Lazlo "Conventionalities in Formula Writing"], but we are going to see them as specifications of atoms presence in a molecule.

The chemical formulas can be seen as commands that have to interpreted in order to do symbolic manipulations that reflect the structure and properties of the compounds they represent. [Laz01]

- ***Code***

Parsing and interpretation

Backus-Naur Form

The notation system Backus-Naur Form (BNF) can be used to specify language grammars that are free of context. (By "free of context" we mean that according to our BNF specification "O2H" is correct. The context of bonds, valency, etc. is not in the BNF specification.) The specification of a grammar using BNF is a list of rules. Each rule has a symbol on the left hand side, and one or more sequences of symbols on the right hand side. If there are more than one sequence on the right hand side they are separated with "|" that specifies alternative. Here is a rule

```
<product> ::= <term> | <term> <product>
```

This rule defines a product to be either a term or term written next to a product. Note that a symbol can appear both on the left and right hand sides. Symbols that do not appear on the left hand side are called *terminals*. (See below.)

Let us describe using the Backus-Naur Form (BNF) for the chemical formula admissible morphemes. We can make the observation the element names are abbreviated, the first letter is always a capital. Abbreviations are unique. We can enumerate all admissible elements as terminal symbols that comprise the rule for the symbol <element>.

```
<element> :: = "H" | "He" | "Li" | "Be" | "B" | "C" | "N" | "O" | "F" | "Ne" | "Na"
| "Mg" | "Al" | "Si" | "P" | "S" | "Cl" | "K" | "Ar" | "Ca" | "Sc" | "Ti" | "V" |
"Cr" | "Mn" | "Fe" | "Ni" | "Co" | "Cu" | "Zn" | "Ga" | "Ge" | "As" | "Se" | "Br" |
"Kr" | "Rb" | "Sr" | "Y" | "Zr" | "Nb" | "Mo" | "Tc" | "Ru" | "Rh" | "Pd" | "Ag" |
"Cd" | "In" | "Sn" | "Sb" | "I" | "Te" | "Xe" | "Cs" | "Ba" | "La" | "Ce" | "Pr" |
"Nd" | "Pm" | "Sm" | "Eu" | "Gd" | "Tb" | "Dy" | "Ho" | "Er" | "Tm" | "Yb" | "Lu" |
"Hf" | "Ta" | "W" | "Re" | "Os" | "Ir" | "Pt" | "Au" | "Hg" | "Tl" | "Pb" | "Bi" |
"Po" | "At" | "Rn" | "Fr" | "Ra" | "Ac" | "Pa" | "Th" | "Np" | "U" | "Am" | "Pu" |
"Cm" | "Bk" | "Cf" | "Es" | "Fm" | "Md" | "No" | "Rf" | "Lr" | "Db" | "Bh" | "Sg" |
"Mt" | "Rg" | "Hs" | "Ds" | "Uub" | "Uut" | "Uuq" | "Uup" | "Uuh" | "Uus" | "Uuo"
```

We need should define numbers. A number is comprised of digits written next to each other. So we define a rule for what a digit is and use it in the definition of number. We define the symbol <digit> as one of the (terminal) symbols 1, 2, 3, 4, 5, 6, 7, 8, 9, 0:

```
<digit> ::= "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"0"
```

Now we can define the number symbol <number> recursively: a number is a digit or a digit written next to a number.

```
<number> ::= <digit>|<digit><number>
```

Next we need to define molecules. A molecule can be an element. Alternatively, a molecule can be put in parentheses. Alternatively, a molecule can be an element and a number next to it. Last, a molecule can also be recursively defined as a molecule written next to another molecule.

```
<molecule> ::= <element>|"("<molecule>")"|<replicated>|<joined>
```

The first alternative, <element>, is already defined. The second alternative is needed to describe formulas like $Ce(SO_4)_2$, i.e. (SO_4) is valid molecule, and it is derived from SO_4.

We define <replicated> as a molecule next to a number or a molecule, underscore, and a number next to each other.

```
<replicated> ::= <molecule><number> | <molecule>"_"<number>
```

We define <connected> simply as two molecules next to each other.

```
<connected> ::= <molecule><molecule>
```

Note that instead of having separate symbol <joined> we could have inserted its definition into the definition of <molecule>:

```
<molecule> ::= <element>|"("<molecule>")"|<replicated>|<molecule><molecule>
```

```
<mix> ::= <molecule> | <molecule> "+" "hν" | <mix> "+" <molecule>
```

```
<equation> ::= <mix> "->" <mix> | <mix> "=" <mix> | <mix> "⇆" <mix>
```

Other formulations

(By "free of context" we mean the grammar and syntax of a sentence like "The cat is flying climbingly." is correct, but the meaning might be absent.)

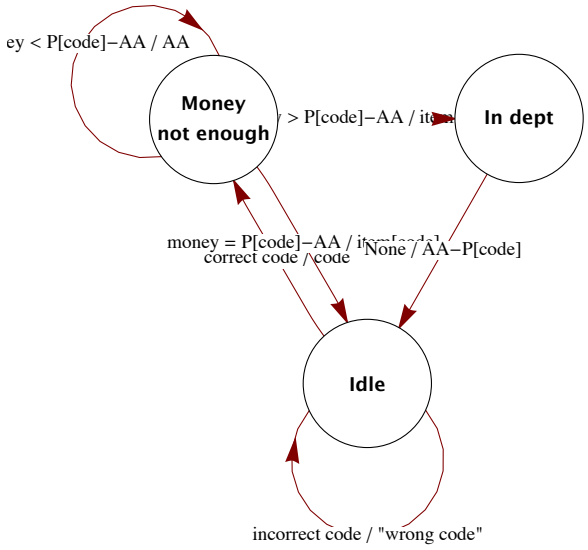
Data structures representing molecules

Finite State Machines

The concept of finite state machine can be used to explain and understand the work of many concrete or theoretical devices encountered in real life and sciences. A typical example of a finite state machine is a vending machine.

A vending machine at its initial state waits for one of two types of events to happen: (1) money deposit, or (2) correct dial pad code entry. If an event of type (1) happens the vending machine waits for an event of type (2) to happen. Similarly, if an event of type (2) happens the vending machine waits for an event of type (1) to happen. After an event of each type has happened, the vending machine checks is the amount of money deposited enough to purchase an item corresponding to the code dialed. If yes, the machine checks does it have an item in stock. If yes, the machine would dispense an item. If the amount of money deposited is larger than the amount of money required the vending machine would return change.

As we can see from this description the vending machine passes through various stages of event awaiting. The reaction to an event would change the vending machine state. For example, after dispensing an item, more money will be in the vending machine bank, and the vending machine will have one less item corresponding to the code dialed. The vending machine has also intermediate states, like counting the deposited money, and buttons pressed.



state digit accumulation

element accumulation

- ***Code***

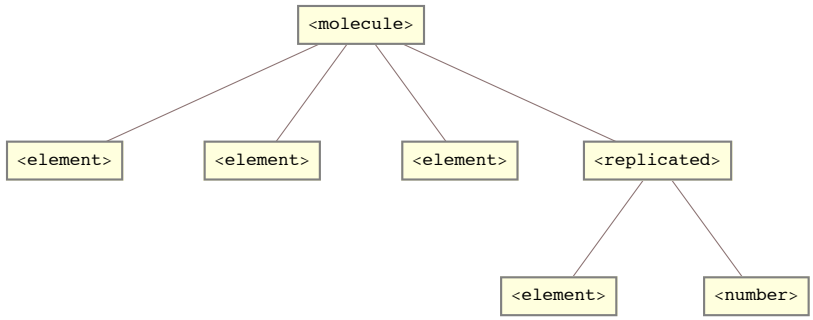
Parser of chemical formulas (first)

- **The basic idea**

We are going to program the parser in such a way that the lexical analysis is combined with the syntactica analysis. We this is easier to do for the problem we are solving would become clear in the exposistion below.

In order to program the parser let us all look at some examples of formulas and their syntactical diagrams. (I don't think programming can be done just considering abstract rules of a specification. One needs to think with concrete examples and contraexamples, and consider how they relate to the absrtact speciction. This point should be stated in the preface and/or in each chapter.)

Let us consider molecules like CaHIO3, CBrClF2, CHBrCl2. They have the same syntactic tree structure:



Syntactic tree of $CaHIO_3$

We can say that they a derived from the symbol <molecule> using the following sequence of transformations.

```
<molecule>
<connected>  /  by the rule <molecule>
<molecule><molecule>  /  by the rule <connected>
<connected><connected> / by the rule <molecule>
<molecule><molecule><molecule><molecule>  / by the rule <connected>
<element><element><element><replicated>   / by the rule <molecule>
<element><element><element><element><number> / by the rule <replicated>
```

If we read the strings of these molecules from left to right, we are reading four element names and then we read a number. The number is applied to the last element name. Before reaching a digit we would read a number of letters -- these formulas can be split into two parts: an all letters one, and a digit one. The capital letters can seen as delimiters that separate the element names from each other. We also observe that if we append the element names to `Molecule[]`, we will have almost correct syntactic structure, only the last node is incorrect. For example,

```
Fold[Append, Molecule[], {"Ca", "H", "I", "O"}]
```

```
Molecule[Ca, H, I, O]
```

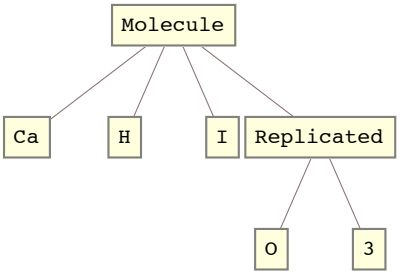
The last node needs to be converted to `Replicated` sub-experssion with the number at the end of the formulas. We can do that using `MapAt`:

```
MapAt[Replicated[#, 3] &, %, -1]
```

```
Molecule[Ca, H, I, Replicated[O, 3]]
```

With these two operations we have obtained the correct syntactical diagram.

```
TreeForm[%]
```



If we think about all formulas that start with a capital letter using the above observations we can come up with the following algorithm:

1. The input is a string `formula`.

2. Initialize sytactical tree `tree` with `Molecule[]` .

3. Read letters until reaching a non-letter (a digit or a left parrenthesis, according to the grammar). Split the formula into all letters part and all digits part.

4. Extract and validate element names from the all letters part. Append the element names to `tree`.

5. Make a number `n` from the all digits part and change the last node into `Replicated[Last[tree],n]`.

This algorithm can be applied to the sub-strings between a matching pair of parethesis. Therefore, we can call it recursively when we encounter "(" and make sure there is a closing ")".

Before giving the complete algorithm, let us resolve the questions of extracting and validating elements from the beginning of a chemical formula string.

Extracting element name abbreviations

First let us get the element name abbreviations in sutiably named varible:

```
elements = ElementData[#, "Abbreviation"] & /@ ElementData[]
```

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

We are going to use the string pattern matching capabilities in *Mathematica*, and more precisely use regular expressions in `StringCases`. For example, using the regular expression string pattern "F[emr]?" we can extract from a string all element name abbreviations that start with "F".

```
StringCases["FrClBFmCHF2", RegularExpression["F[emr]?"]]
```

```
{Fr, Fm, F}
```

So let us divide the elements into groups of elements that start with the same letter. We can accomplish that by gathering them -- using `GatherBy` -- according to their first letter. (We also sort the result of `GatherBy` alphabetically.)

```
elementGroups = SortBy[Sort /@ GatherBy[elements, StringTake[#, 1] &], First];
elementGroups // ColumnForm
```

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

Then we construct a regular expression for each group.

```
elementGroupPatterns =
 MapThread[
  If[Length[#2] > 0,
    #1 <> "[" <> Apply[StringJoin, #2] <> "]" <> If[#3, "?", ""], #1] &,
  {StringTake[#[[1]], {1}] & /@ elementGroups,
   Union[Flatten[Map[Characters[StringTake[#, {2, -1}]] &, #]]] & /@ elementGroups,
   MemberQ[#, StringTake[#[[1]], {1}]] & /@ elementGroups
  }, 1]
```

```
{A[cglmrstu], B[aehikr]?, C[adeflmorsu]?, D[bsy], E[rsu], F[emr]?, G[ade],
 H[efgos]?, I[nr]?, K[r]?, L[airu], M[dgnot], N[abdeiop]?, O[s]?, P[abdmortu]?,
 R[abefghnu], S[bcegimnr]?, T[abcehilm], U[bhopqstu]?, V, W, X[e], Y[b]?, Z[nr]}
```

Then we combine the group regular expressions into one.

```
elementsPattern = StringJoin @@ Riffle[elementGroupPatterns, "|"]
```

```
A[cglmrstu]|B[aehikr]?|C[adeflmorsu]?|D[bsy]|E[rsu]|F[emr]?|G[ade]|H[efgos]?|I[nr]?|K[r]?|L[
  airu]|M[dgnot]|N[abdeiop]?|O[s]?|P[abdmortu]?|R[abefghnu]|S[bcegimnr]?|T[abcehilm]|U[
  bhopqstu]?|V|W|X[e]|Y[b]?|Z[nr]
```

We can check and demonstrate that the pattern is working by applying it to random molecules from `ChemicalData`.

```
randomCompounds = ChemicalData[#, "CompoundFormulaString"] & /@
  ChemicalData["MetalCarbonyls"][[RandomInteger[{1, 50}, 5]]]
```

```
{C7H8O4Rh, C12H2Fe3O12, C6O6W, C11H11NO5W, C12H10CrO5}
```

```
randomCompoundElements =
 Map[StringCases[#, RegularExpression[elementsPattern]] &, randomCompounds]
```

```
{{C, H, O, Rh}, {C, H, Fe, O}, {C, O, W}, {C, H, N, O, W}, {C, H, Cr, O}}
```

formula	*elements*
C7H8O4Rh	{C, H, O, Rh}
C12H2Fe3O12	{C, H, Fe, O}
C6O6W	{C, O, W}
C11H11NO5W	{C, H, N, O, W}
C12H10CrO5	{C, H, Cr, O}

Extracting numbers

This is straightforward. For example,

```
StringCases["12H10CrO5", RegularExpression["^\\d+"], 1]
```

```
{12}
```

The parser as a finite state machine

Out[607]=

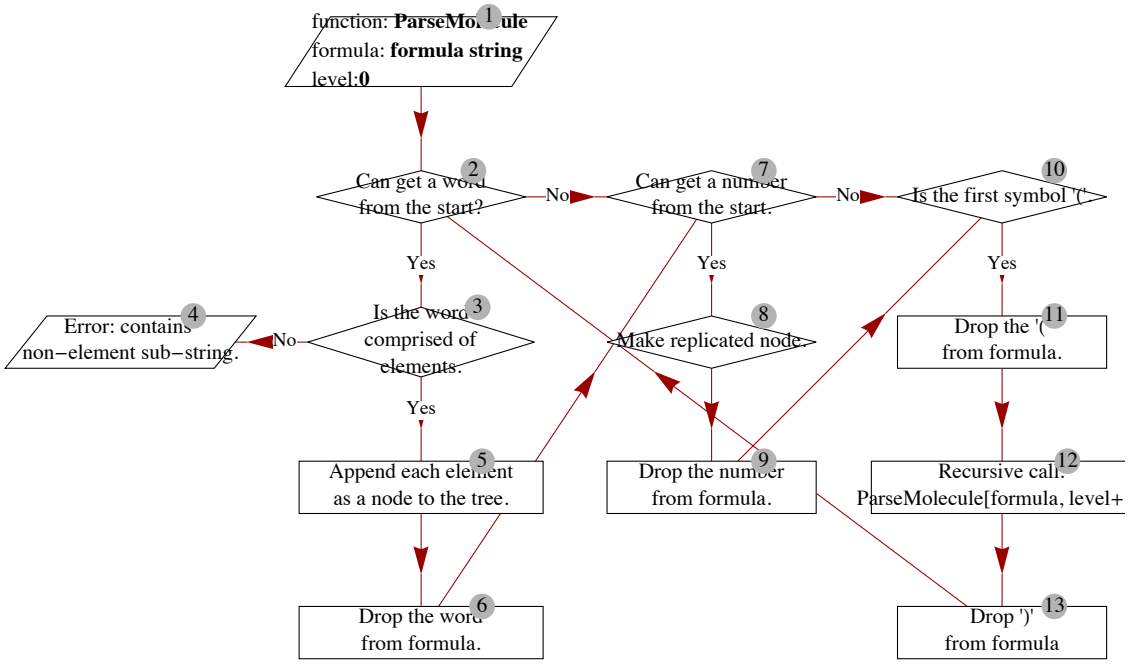
state	*input*	*new state*	*action*
INIT		WORD READ	tree=Molecule[]
WORD READ	formula[[1]] is a letter	WORD READ	Accumulate letters to word, formula=Rest[formula]
WORD READ	formula[[1]] is not a letter	ELEMENTS	
ELEMENTS	word is comprised of elements	POST-ELEMENTS	Append elements to tree
ELEMENTS	word has non-element sub-string	ERROREXIT	Message: wrong element(s).
POST-ELEMENTS	formula[[1]] is a digit	NUMBER READ	
POST-ELEMENTS	formula[[1]] is '_'	NUMBER READ	formula=Rest[formula]
POST-ELEMENTS	formula[[1]] is '('	SUB-FORMULA-READ	formula=Rest[formula]; level++
POST-ELEMENTS	formula[[1]] is ')'	EXIT	formula=Rest[formula]; Return[{tree,formula}]
NUMBER READ	formula[[1]] is a digit	NUMBER READ	Accumulate digits to num, formula=Rest[formula]
NUMBER READ	formula[[1]] is not a digit	REPLICATION	
REPLICATION	Length[tree]==0	ERROREXIT	Message: number without an element.
REPLICATION	Length[tree]>0	POST-REPLICATION	tree[[-1]]=Replicated[tree[[-1]],num]
POST-REPLICATION	formula[[1]] is a letter	WORD READ	
POST-REPLICATION	formula[[1]] is '('	SUB-FORMULA-READ	formula=Rest[formula]; level++
POST-REPLICATION	formula[[1]] is '_'	ERROREXIT	Message: '_' is not allowed after a number
POST-REPLICATION	formula[[1]] is ')'	EXIT	formula=Rest[formula]; Return[{tree,formula}]
SUB-FORMULA-READ		INIT	oldtree=tree
EXIT	level==0		Return[{tree,formula}]
EXIT	level>0	WORD READ	tree=Append[oldtree,tree]; level--

In[608]:=
```
stateTable[[All, 1]] // Union
% // Length
```

Out[608]=
```
{ELEMENTS, EXIT, INIT, NUMBER READ, POST-ELEMENTS,
 POST-REPLICATION, REPLICATION, SUB-FORMULA-READ, WORD READ}
```

Out[609]= 9

The complete algorithm



Parser of chemical formulas

In order to implement the parser we need to look

In order to implement the parser we have to imagine how by reading the formulas from left to right we

Reading a formula from left to right we would encounter different combinations of symbols that have to be discerned into separate entities. Let us make several observations. (Can be done from the BNF.)

- There are five types of symbols: capital letters, small letters, digits, left and right parenthesis, and the underscore.
- All element names begin with a capital letter.
- The underscore symbol is only followed by a digit.
- The number of left parenthesis is the same as the number of right parenthesis.
- Charaters between a matching pair of parenthesis can be seen as chemical formulas. We will call these substrings submolecules.
- Molecules and submolecules do not begin with a digit.

Not all sequences of characters appear. For example, a digit cannot be followed by the underscore. Let us find all possible occurances allowed by the BNF of one type of symbol followed by another. First, let us look at the BNF rules and

Out[461]=

description	BNF symbol sequence	example	rule of derivation
a letter followed by a letter	...<element><element>...	...ClF...	<connected> → <element><element>
a letter followed by a digit	...<element><number>...	...Cl2...	<replicated> → <element><number>
a letter followed by _	...<element>_<number>...	...O_2...	<replicated> → <element><number>
a letter followed by (	...<element>"("<element>...	...H(Cl...	<connected> → <element><molecule>
a letter followed by (...(	...<element>"("..."("<element>...	...H((Cl	<connected> → <element><molecule>
a digit followed by a capital letter	...<number><element>...	...2H...	<replicated><element> or <replicated><replicated>
a digit followed by (	...<number>"("...	...2(Mn..	<replicated><molecule>
) followed by a capital letter	...")"<element>...	...)C...	<molecule><molecule>
) followed by a digit	...")"<number>...	...)3...	<replicated>
) followed by _	...")"_<number>...	...)_3...	<replicated>
) followed by (	...")""("<element>...	...)(C...	<molecule><molecule>
) followed by (...(	...")""("..."("<element>...	...)((Cl...	<connected> repeated

Out[605]=

character type	capital letter	small letter	digit	(	)	_
capital letter	✓	✓	✓	✓	✓	✓
small letter	✓	✓	✓	✓	✓	✓
digit	✓		✓	✓	✓	
(	✓			✓		
)	✓		✓	✓	✓	✓
_			✓			

Parser of chemical equations

String matching parser

Miscellaneous

Exercises

Solutions

- **Exercise 3: Program that generates random molecules**
- **Exercise 4: Statistics over random molecules**
- **Exercise 5: Profiling**

```
rm = {};
While[! (290 ≤ LeafCount[rm] ≤ 300), rm = RandomMolecule[]]
```

```
MoleculeToAtoms /@ Table[rm, {100}]; // Timing
```

```
{0.96343, Null}
```

Exercise 6: Grouping by syntactic analysis trees

```
cfs = ChemicalData[#, "CompoundFormulaString"] & /@
   Apply[Join, ChemicalData[#] & /@ ChemicalData["Classes"]];
cfs //
 Length
```

```
115 486
```

```
pms = DeleteCases[First[ParseMolecule[#]] & /@ cfs, First[$Failed]];
pms // Length
```

```
113 297
```

```
mpatterns = Union[pms /. {s_String → (String), n_Integer → (Integer)}];
mpatterns // Length
```

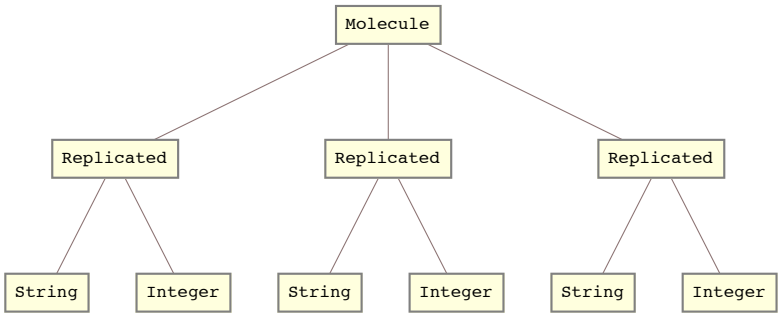
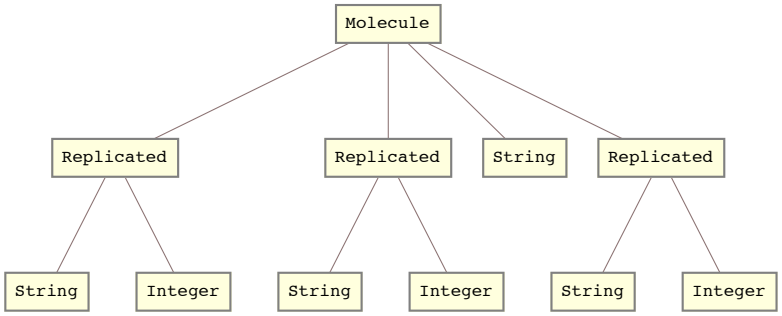
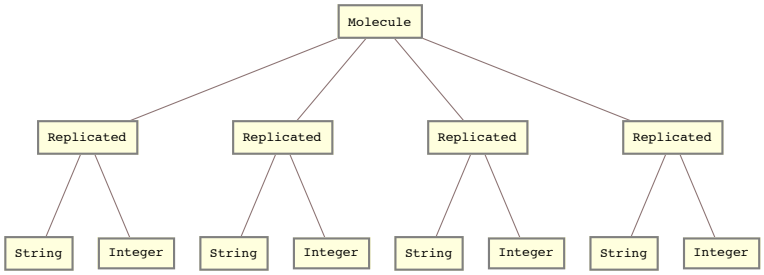
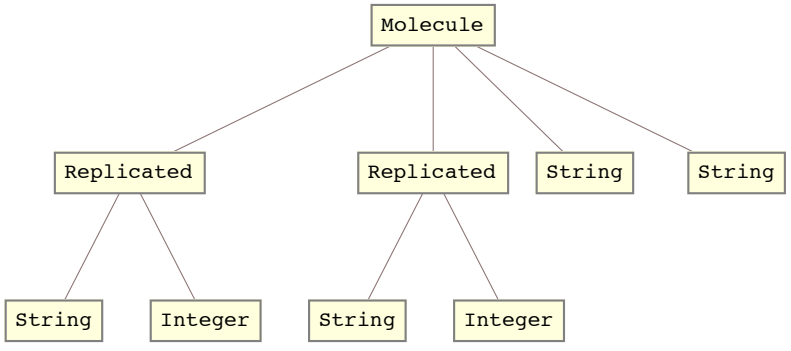
```
113
```

```
pcounts =
 Count[pms, #] & /@ (mpatterns /. {String → _String, Integer → _Integer})
```

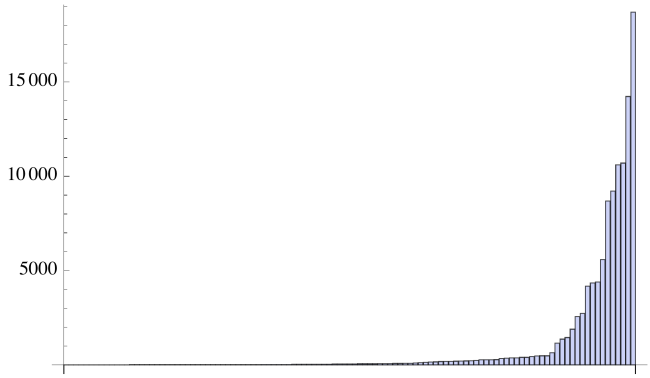
```
{486, 186, 38, 473, 442, 1153, 5584, 73, 277, 371, 273, 156, 275, 9210, 18 696, 28, 58, 66, 48, 104,
 132, 182, 177, 23, 79, 334, 276, 10 601, 14 224, 8683, 10 693, 9, 9, 6, 22, 3, 19, 9, 59, 14, 27,
 25, 21, 6, 6, 22, 59, 31, 62, 49, 54, 4178, 4393, 4347, 2729, 1889, 1366, 2560, 1448, 11, 14,
 11, 2, 8, 10, 7, 8, 3, 4, 5, 401, 400, 645, 232, 371, 162, 481, 196, 116, 76, 355, 88, 199, 38,
 219, 212, 8, 82, 22, 12, 67, 1, 4, 48, 6, 41, 6, 24, 4, 13, 17, 2, 6, 5, 3, 1, 8, 6, 14, 9, 7, 6, 3}
```

```
pos = Reverse@Ordering[pcounts, -4]
Grid[Prepend[Transpose[{pcounts⟦pos⟧, TreeForm[#, ImageSize → 400] & /@ mpatterns⟦pos⟧}],
  Style[#, Bold, Italic] & /@ {"Number of formulas", "Pattern"}],
 Dividers -> {All, Join[{True, True}, Table[False, {Length[pos] - 1}], {True}]}]
```

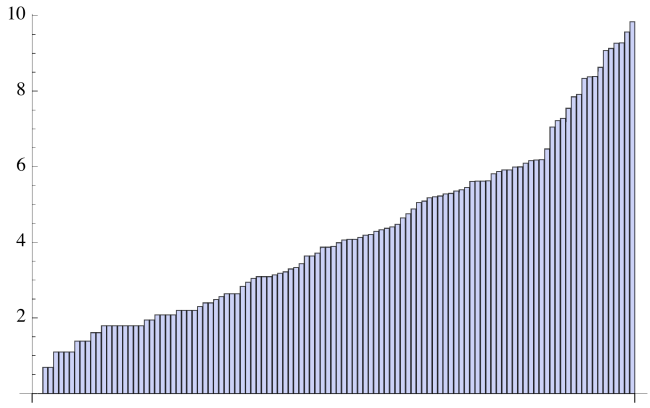
```
{15, 29, 31, 28}
```

Number of formulas	***Pattern***
18 696	
14 224	
10 693	
10 601	

```
BarChart[Sort@pcounts, PlotRange → All]
```




```
BarChart[Sort@Log@pcounts, PlotRange → All]
```



References

Section hyperlinks