Design

The project converts a user's molecule into a systematic name, with the help of two a user-customisable files, one ending .PeriodicTable and the other ending .ExamSpec

1. The program procedurally generates a graph from the user's input to model the molecule they want named. This is the internal 'image' the program forms, so it can examine and operate on the structure the user has described. This graph is stored as the class *ElementGraph*. The program has no prior knowledge of this molecule, so the user's input is the only information it recieves about the specific input molecule.
2. Next, the program validates that the user input is valid and possible to name by finding the important bits, known as the (functional) groups and making sure the program knows how to name them (from the info provided in the .ExamSpec file).
3. If so, the program makes a series of decisions to find the optimal 'backbone' of the molecule. This is known as the (parent) chain, and is a core path of carbons everything else is based on.
4. The functional groups are converted into words like "hydroxy" or "chloro" which the .ExamSpec file defined to represent them. The position where these groups are attached to the parent chain is represented by a number.
5. These words and numbers are joined together using specific alphabetic, vowel and priority rules, which are also taken from the .ExamSpec file. The name of the molecule is output.

In for the program to work on a near infinite number of real or imaginary molecules, everything is strictly procedural.

Some of the features I used were:

- **Procedurally Generated Objects** allow the graph of *Elements* to be as large as the user requires.
- **Hashset** operations are used to identify subsets of functional groups or unions where chains and branches join together.
- **Lambda Expressions** are used throughout the program to efficiently extract subsets from arrays or lists. This allows information to be extracted from classes very quickly and simply.
- **Recursive DFS** is an algorithm used to find the path between any two *Element* objects with name 'carbon'.
- **Inheritance** is used to allow different *FunctionalGroup* objects to store different data.
- **Binary Files** are used to specify the user defined elements and specific naming rules. This stores several dictionaries which map complex data such as hashsets to strings or strings to nested tuples. This data is converted between forms to optimise file space.
- **REGEX** is used to ensure the name meets vowel rules.

The program works in two distinct sections. The user input builds an *ElementGraph* and the program validates that the input meets some basic rules. The naming stage then applies rules included in the user specification file to construct a name from this graph after several rounds of 'narrowing down' the possible ways to name it.

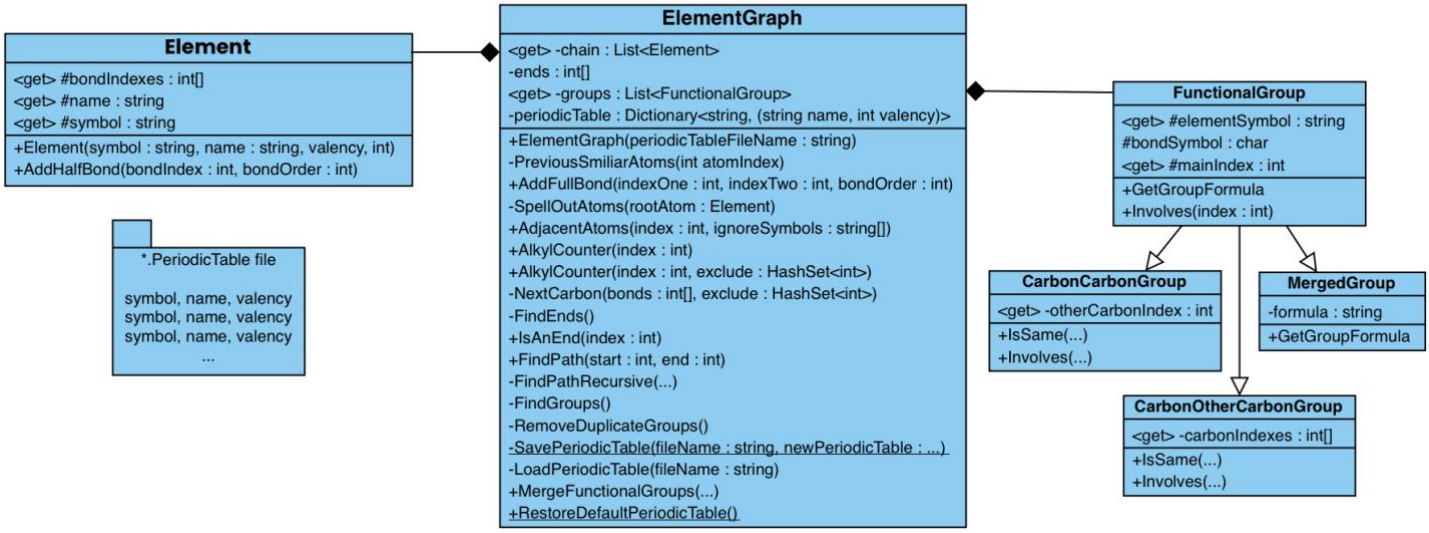
ElementGraph

This class operates on a graph of *Element* objects which represents the structure of atoms joined together to form a molecule. This abstraction allows the molecule to be named purely procedurally, so the program works offline, satisfying objective **4.c**. Each unique graph structure represents a possible arrangement of atoms and thus a molecule, so the graph is the intermediate representation needed for objective **4.a**. The order that the user decides to describe the atoms would not affect the name of the molecule as each graph produced would isomorphic. This means there is a many-to-one relationship between several user inputs and a graph for objective **4.b** to be met.

- The graph is **unweighted** and bonds are stored in an adjacency list within each node. Nodes can form multiple edges with each other, representing bonds of higher order. This was chosen over weightings as each atom must make a set number of bonds (valency), so the adjacency list is an array of fixed size.
- The graph is **nondirected** as a bond is identical for both participant molecules as the program does not support asymmetric bonding i.e. dative bonds.
- The graph is currently **connected** as all atoms are inputted by growing off a base atom. Furthermore, the graph of only carbon atoms must also be **connected** as breaks in the carbon chain are not yet supported and listed as constraints.
- The graph is **acyclic** as cyclic and aromatic molecules cannot be inputted and are listed as constraints.

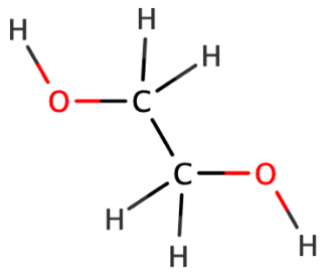
The graph of carbon atoms (if all non-carbon atoms are ignored) must be **acyclic** and **connected** as this class uses **recursive depth first search** on carbon atoms to find carbon chains and subsequently carbon numbers. This requires the graph of carbon atoms to behave as a tree which is why breaks in the carbon chain are currently constraints.

This class also generates a list of *FunctionalGroup* objects which store the important bits of the molecule for naming. For example, a carbon double bonded to an oxygen atom is stored as an object of type *FunctionalGroup*, where *elementSymbol* is “O” and *bondOrder* is 2. The *GroupFormula* would then be returned as “C=O”. This ‘formula’ string uniquely identifies the important information of the functional group. The *CarbonCarbonGroup* class stores groups involving two carbon atoms such as “C=C” and the *CarbonOtherCarbonGroup* class stores groups such as “C-O-C” where the carbon chain is split by a different atom. The *CarbonOtherCarbonGroup* class is only used for futureproofing so molecules that break the carbon chain can be included in the future.



SpellOutAtoms

The user input is simple and command line based, so the user can 'spell out' what atoms are bonded together. As the target user is inexperienced in organic chemistry, the program ensures they make no mistakes and guides them through the process, as it is the main area where the user interacts with the program. This is how the program works out what molecule the user wants to be named.




```
What is bonded to Carbon number 1?
4 bonds left.
C
How many bonds do these two atoms form?
Maximum 4.
2
What else is bonded to Carbon number 1?
2 bonds left.
O
How many bonds do these two atoms form?
Maximum 2.
1
What else is bonded to Carbon number 1?
1 bonds left.
H
What else is bonded to Carbon number 2?
2 bonds left.
O
How many bonds do these two atoms form?
Maximum 2.
1
What else is bonded to Carbon number 2?
1 bonds left.
H
What else is bonded to Oxygen number 1?
1 bonds left.
H
What else is bonded to Oxygen number 2?
1 bonds left.
H
```

This function is the main part of the constructor of the *ElementGraph* class and is where objects are generated dynamically into the graph structure.

On the right, the user input is outlined for the molecule drawn out (ethane-1,2-diol). The user enters the symbol of an atom which is bonded to the atom the program is focusing on. This approach starts from a root carbon atom and continues for each atom until every atom has made sufficient bonds, which is when the input stage halts. To differentiate between atoms of the same element, the program numbers them in the order they were created. For example, "Oxygen number 1" was the first oxygen atom the user created and "Oxygen number 2" was the most recent one created, so the user does not get confused. These numbers play no significance in the naming of the molecule (as carbon numbers do later on) but simply avoid ambiguity or confusion, and do not exist beyond this method. For larger molecules it is recommended that the user keeps an image of the structure in front of them and labels the atoms in the order they are inputted, so they do not forget. This type of input assumes no prior knowledge except a visual idea of the molecule's structure.

`List<Element> atoms:` the list of every atom in the molecule. This holds each node in the graph.

`Element rootAtom:` the atom the program is currently focusing on and surrounding with other atoms, until it has made enough bonds.

`int rootAtomIndex:` the index of the *rootAtom* in *atoms*.

`int currentAtomIndex:` the index of the atom the user is currently creating, which surrounds the *rootAtom*.

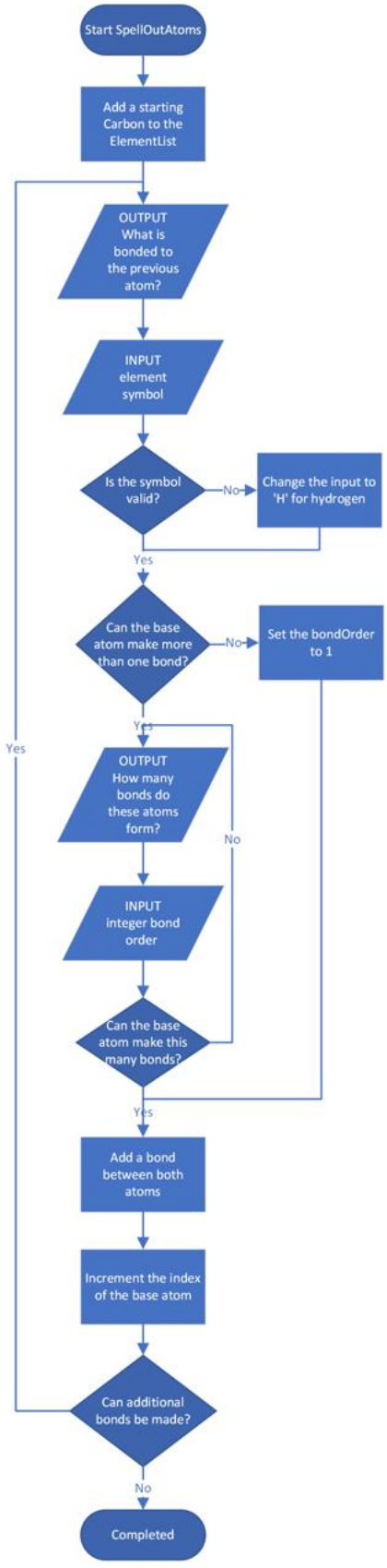
`int maximumBonds:` the maximum number of bonds the *rootAtom* and the *currentAtom* can form. This is the smallest between the number of free bonds each atom can make.

`bool validBondOrder:` a Boolean value to indicate if it is possible for the two atoms to make the number of bonds the user requests.

`int bondOrder:` the number of bonds for the two atoms to form between each other. In this program, double bonds consist of two edges rather than simply having a weighting of two.

Pseudocode

```
Method SpellOutAtoms(rootAtom)
    atoms ← [rootAtom]
    rootAtomIndex ← 0
    currentAtomIndex ← 1
    Repeat
        While atoms[rootAtomIndex].BondsFree > 0
            bondOrder ← 1
            Display "What {else} is bonded to {element}
{element number}”
            Display "{bonds free} bonds left."
            symbolInput ← Input()
            symbolInput.Capitalise()
If periodicTable.ContainsKey(symbolInput)
    atoms.Add(new Element(…))
Else If symbolInput = ""
    symbolInput ← "H"
    atoms.Add(new Element(…))
Else
    Display "Invalid... Please try again."
    Continue (go back to the start of the loop)
End If
MaximumBonds ← Min(rootAtom.BondsFree,atoms.Last().BondsFree)
            validBondOrder ← False
            Repeat
                If MaximumBonds > 1
                    Display "How many bonds do these two
atoms form?"
                    Display "Maximum {MaximumBonds}."
                    validBondOrder ← TryParse(bondOrder ←
Input()) AND bondOrder = MaximumBonds
                Else
                    bondOrder ← 1
                    validBondOrder ← True
                End If
```



```
                If NOT validBondOrder
                    Display "Invalid... Please try again."
                End If
            Until validBondOrder = True
            AddFullBond(rootAtomIndex, currentAtomIndex, bondOrder)
            currentAtomIndex ← currentAtomIndex + 1
        End While
        rootAtomIndex ← rootAtomIndex + 1
        rootAtom ← atoms[rootAtomIndex]
    Until CountAtomsWithBonds() > 0 // do while there are bonds to be made
End Method
```

FindPath

This function finds the indexes of the path between two nodes. As cycles are forbidden, the *ElementGraph* is an **undirected acyclic graph**, which is essentially a tree. This means that there is a one-to-one relationship between a pair of nodes and the path between them. This means there is no use using a shortest path algorithm as only one path exists. All this function does is call the *FindPathRecursive* function with initial conditions. The path returned is an array of integers which are the indexes of carbon atoms in the *ElementGraph* atoms *Element* list.

```
Pseudocode
 Method FindPath(start, end)
    path ← new List<int>()
    visited ← new HashSet<int>()
    FindPathRecursive(start, end, ref visited, ref path)
    Return path
End Method
```

FindPathRecurisve

This function uses the current index, the end index, the *HashSet* of visited nodes and the list of indexes in the path. The *HashSet* and list are mutated by the recursive function. They are passed by reference in C# by default, but the `ref` keyword has been included for clarity. The function returns a bool to indicate which paths are successful. When the end index is met, the function returns True, which validates the path that led to that index when the stack is emptied. However, if a dead end is reached, the current index is removed from the graph and False is returned. Each time the function is called, if every bond at the current index returns False from the recursive function, the current

index is also part of the dead-end path so is removed too. This method uses the mutability of lists to fill up the possible paths before 'pruning' the dead ends. This is an implementation of **recursive depth first search**.

Pseudocode

```
Method FindPathRecursive(current, end, ref visited, ref path)
    visited.Add(current)
    path.Add(current)
    If current = end
        Return True
    End If
    bonds ← AdjacentAtoms(current)
    For Each bond In bonds
        If NOT visited.Contains(bond)
            If FindPathRecursive(bond, end, ref visited, ref path)
                Return True
            End If
        End If
    Next
    path.Remove(current)
    Return False
End Method
```

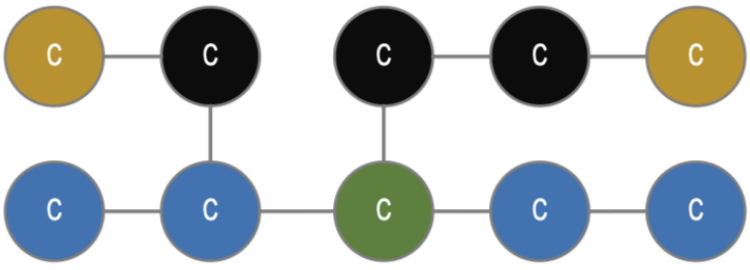
```
private bool FindPathRecursive(int current, int end, ref HashSet<int> visited, ref List<int> path)
{
    visited.Add(current);
    path.Add(current);

    if (current == end)
    {
        return true;
    }
    int[] bonds = AdjacentAtoms(current);
    foreach (int bond in bonds)
    {
        if (!visited.Contains(bond))
        {
            if (FindPathRecursive(bond, end, ref visited, ref path))
            {
                return true;
            }
        }
    }

    path.Remove(current);
    return false;
}
```

The nested conditional statements highlighted in the pseudocode are intentional to show that the *FindPathRecursive* method should only be called if the first condition is met, as the method has side effects and calling it on every visited bond causes inefficiency in the best of cases and stack overflow in the worst of cases. The C# Boolean operator '&&' has the same function but nested if statements are clearer and more readable.

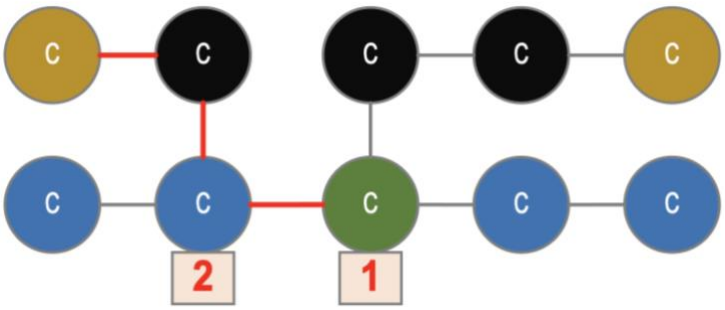
FindBranches

```
public List<List<int>> FindBranches(List<int> path)
{
    List<List<int>> branches = new List<List<int>>();
    foreach (int i in path)
    {
        if (AlkylCounter(i) > 2)
        {
            foreach (int end in ends)
            {
                if (!path.Contains(end))
                {
                    List<int> branch = FindPath(i, end);
                    if (branch.Intersect(path).Count() == 1) //new branch
                    {
                        branches.Add(branch);
                    }
                }
            }
        }
    }
    return branches;
}
```

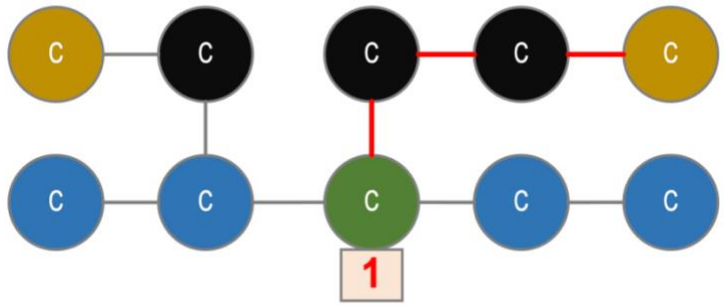
This method takes a chain and finds branches of excluded carbons. If a carbon in the chain has more than two carbons connected to it (*AlkylCounter*), one of its adjacent carbons is not in the chain . The method finds the path between that carbon and each of the ends of the molecule. If this path only shares one element with the main chain (the point where they join), this branch is separate and valid.



This diagram represents the carbons in an *ElementGraph*. The blue represents the main chain, black atoms are not part of the chain, yellow atoms are ends that are not part of the chain and green is the current carbon. The red will represent the route of a possible branch.



This possible branch shares two atoms in common with the main chain, so is not being expressed in its simplest terms. This means that the current atom is not the point where the branch leaves the main chain, so this branch is not valid.



This possible branch shares one atoms in common with the main chain, the current atom. This means that the current atom is the point where the branch leaves the main chain, so this branch is valid and is added to the list of branches.

FindGroups

This function uses a range of overloaded methods from the *ElementGraph* class to get information about the atoms bonded together. This is taken from the *Element* class, meaning that four classes are involved in this method.

Pseudocode

```
Method FindGroups()
    newGroups ← new List<FuncionalGroup>()
    For atomIndex ← 0 To atoms.Count
        atom ← atoms[atomIndex]
        If atom.Name = "Carbon"
            unvisitedBonds ← AdjacentAtoms(atomIndex)
            ForEach bondIndex In unvisitedBonds
                bondedAtom ← atoms[bondIndex]
                order ← BondOrder(atomIndex, bondIndex)
                If bondedAtom.Name = "Carbon" AND order ≠ 1
                    newGroups.Add(new CarbonCarbonGroup(…))
                Else If bondedAtom.Name ≠ "Carbon"
                    newGroups.Add(new FuncionalGroup(…))
                End If
            Next
        Else If atom.Name ≠ "Hydrogen" AND AdjacentAtoms(…).Count() > 0
            ERROR
        End If
        Else If AlkylCounter(atomIndex) > 1
            For Each bond In atom.BondIndexes
                If atoms[bond].Name = "Carbon"
```

```
                    If BondOrder(atomIndex, bond) > 1
                        ERROR
                    End If
            For Each bond In atom.BondIndexes
                If atoms[bond].Name = "Carbon"
                    If BondOrder(atomIndex, bond) > 1
                        ERROR
                    End If
                    carbonIndexes.Add(bond)
                End If
            Next
            newGroups.Add(new CarbonOtherCarbonGroup(…))
        End If
    Next
    Return newGroups
End Method
```

This method iterates through every atom and checks their bonds for functional groups which can be stored as objects in a list, which the naming stage of the program can use.

This method is unique as it manages all of the program's error handling. Objective **5**, aims to reject structures which are vaguely theoretically possible but make no sense. This is done with a variety of if statements along with small bits of graph traversal. The conditional statements in the pseudocode are colour coded and explained below:

- If two carbon atoms make a double or triple bond, that is recognised here by repeated edges being found with the *BondOrder* method in the *ElementGraph* class. This is then recorded as a *CarbonCarbonGroup*.
- If a carbon atom is bonded to an atom that is neither hydrogen nor carbon, this is recorded as a regular *FunctionalGroup*.
- If a non-carbon atom is bonded to multiple carbon atoms, this is recognised here and is recorded as a *CarbonOtherCarbonGroup*. This is currently rejected by the naming part of the program as breaks in the carbon chain are not delt with, but this allows for futureproofing.
- If this *CarbonOtherCarbonGroup* contains any double or triple bonds, the structure no longer makes sense, and is rejected.
- If multiple non-carbon atoms are bonded together, the structure is rejected.

MergeGroups

This function is the first to use the rules defined in the specification file. It searches the molecule for carbons with multiple functional groups (i.e. C-O and C=O), so these can be combined into one new

functional group with an arbitrary formula (i.e. COOH). This allows these groups to be named differently fulfilling objective **1.e**. In this case, an alcohol and aldehyde on the same carbon combine to make a carboxylic acid. The variable `toMerge` is a dictionary that maps hashsets of strings (functional group formulae) to a string (the new combined formulae).

Pseudocode

```
Method MergeFunctionalGroups(toMerge)
    For i ← 0 To atoms.Count - 1
        formulae ← GroupsOnCarboni
        For Each collect In toMerge.Keys
            If collect.IsSubsetOf(formulae)
                formulae ← new HashSet<string>(collect) DEBUG EVALUATION *
                groups.Add(new MergedGroup(toMerge[collect], i))
                For j ← groups.Count - 1 To 0 Step -1
                    If groups[j].Involves(i) And FormulaeContainsGroup
                        formulae.Remove(groups[j].GroupFormula)
                        groups.RemoveAt(j)
                    End If
                Next
            End If
        Next
    Next
End Procedure
```

The highlighted line here uses hashset operations to determine if the current carbon atom contains all the groups required for a merger to take place, then the merger is executed. `toMerge[collect]` is used instead of `toMerge[formulae]` as when hashsets are used as keys for dictionaries in C#, dictionaries of equivalent elements are strangely not mapped the same way.

IUPAC

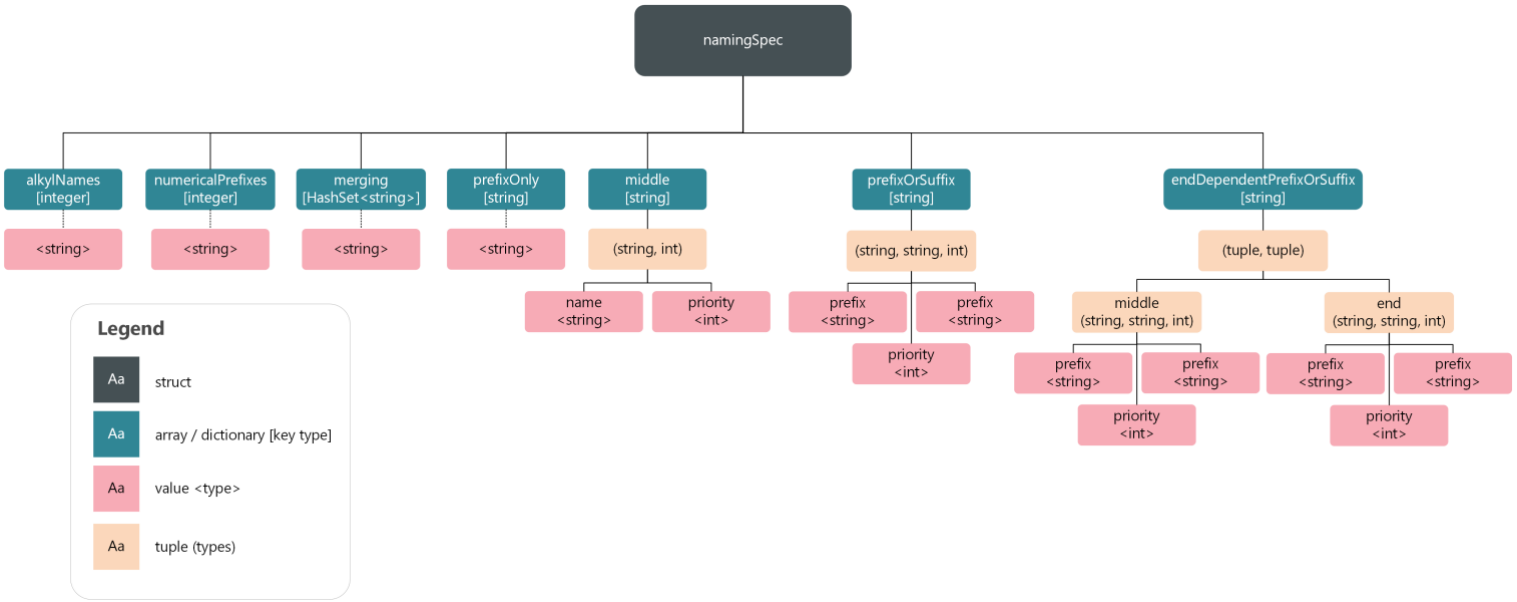
IUPAC stands for “The International Union of Pure and Applied Chemistry”, which maintain a global standard for how to consistently name molecules. The rules are very complete and, besides a few cases, essentially define a formal language. This class is where the naming part of the program happens. The class uses the graph of atoms and the list of functional groups along with a user defined naming specification derived from a binary file. The challenge of naming molecules is that functional groups that form part of the name are either at the start of the name (prefix), middle of the name (middle) or at the end (suffix) and this depends on the type of group along with the ‘priority’. The highest priority groups are chosen to be at the suffix, and the rest are used as prefixes, which are ordered alphabetically. The naming part of the program is in this class so multiple IUPAC

objects can be created using different specifications on the same *ElementGraph*. This allows a molecule to be named in several different ways at once without requiring the user input to be repeated.

NamingSpec

```
public struct namingSpec
{
    public string[] alkylNames;
    public string[] numericalPrefixes;
    public Dictionary<HashSet<string>, string> merging;
    public Dictionary<string, string> prefixOnly;
    public Dictionary<string, (string name, int priority)> middle;
    public Dictionary<string, (string prefix, string suffix, int priority)> prefixOrSuffix;
    public Dictionary<string, ((string prefix, string suffix, int priority) middle, (string prefix, string suffix, int priority) end)> endDependentPrefixOrSuffix;
}
```

This struct stores the naming information derived from the binary file. Dictionaries are used to map functional group formulae to their naming information (as defined by the user). For example, a (C-F) group maps to the prefix “fluoro”.



The name of the molecule can be split into three separate sections, the prefix, the middle and the suffix. The suffix contains the group with the highest priority, and all other groups are added to the prefix. The groups that are named this way have their naming information stored in the “*prefixOrSuffix*” dictionary. Three pieces of information are provided: how to name it as a prefix, how to name it as a suffix, and the priority to decide which to use. Some groups never join the suffix of the name, and always join the prefix, so have their naming information stored in the “*prefixOnly*” dictionary. This means the group is only ever named one way, so only one string is provided. Some other groups (usually carbon-carbon double/triple bonds) are only expressed in the middle of the name, so have their naming information stored in the “*middle*” dictionary. These are named in one way, so only one string is provided. However, a priority is also provided, which is used by other methods. Some groups that would usually be named from the “*prefixOrSuffix*” dictionary are named differently based on where the group lies on the molecule. These groups require two sets of naming information, one for if the group is in the middle of the *ElementGraph* and one for a group on the ends of the *ElementGraph*. This means the information would be ‘end-dependent’ so would be stored in the “*endDependentPrefixOrSuffix*” dictionary. In the pseudocode provided in this documentation, “*endDependentPrefixOrSuffix*” may be abbreviated to “*endDependent…*”. An

example of a group named with this dictionary is the C=O group which is either an aldehyde (if on the end) or a ketone (if in the middle). This information would be stored as below where the first value tuple is for 'middle' groups and the second is for 'end' ones.

```
fullSpec.endDependentPrefixOrSuffix.Add("C=O", (("oxo", "one", 2), ("formyl", "al", 3)));
```

SaveSpecification

This static method takes a *namingSpec* variable and filename as creates a binary file to store this config. This method saves space in the binary file by using the fact that the *priority* described earlier is simply used as an order for comparisons. This means that the method can convert the dictionaries into arrays of tuples which can be ordered by this *priority* and saved into the binary file in order of increasing *priority*. This removes the need to save *priority* as a separate integer in the file and avoids any logical errors that may arise from two groups having the same *priority*.

Pseudocode

```
Method SaveSpecification(fileName, spec)
    fileName += ".ExamSpec"
    Using writeFile
        writeFile.Write(((short)spec.alkylNames.Length)) // less than 32,000
        For Each name In spec.alkylNames
            writeFile.Write(name)
        Next
        writeFile.Write((short)spec.numericalPrefixes.Length)
        For Each pref In spec.numericalPrefixes
            writeFile.Write(pref)
        Next
        writeFile.Write(((short)spec.merging.Count))
        For Each v In spec.merging
            writeFile.Write(((short)v.Key.Count))
            For Each s In v.Key
                writeFile.Write(s)
            Next
            writeFile.Write(v.Value)
        Next
        writeFile.Write(((short)spec.prefixOnly.Count))
        For Each v In spec.prefixOnly
            writeFile.Write(v.Key)
```

```
            writeFile.Write(v.Value)
        Next
        sortedMiddle ← convert to (key, value) tuple array
        sortedMiddle ← order by priority
        writeFile.Write(((short)sortedMiddle.Count))
        For i ← sortedMiddle.Count - 1 To 0 Step -1
            writeFile.Write(sortedMiddle[i].key)
            writeFile.Write(sortedMiddle[i].name)
        Next
        sortedSuffixes ← convert to (key, value) tuple array
        sortedSuffixes ← add middle part of endDependentPrefixOrSuffix tuple array
        sortedSuffixes ← add end part of endDependentPrefixOrSuffix tuple array
        sortedSuffixes ← order by priority
        writeFile.Write(((short)sortedSuffixes.Count))
        For i ← sortedSuffixes.Count - 1 To 0 Step -1
            writeFile.Write(sortedSuffixes[i].key)
            writeFile.Write(sortedSuffixes[i].middleOnly)
            writeFile.Write(sortedSuffixes[i].endOnly)
            writeFile.Write(sortedSuffixes[i].prefix)
            writeFile.Write(sortedSuffixes[i].suffix)
        Next
        writeFile.Close()
        Function getPriorityForSuffixSort(x)
            If x.middleOnly
                Return spec.endDependentPrefixOrSuffix[x.key].middle.priority
            ElseIf x.endOnly
                Return spec.endDependentPrefixOrSuffix[x.key].end.priority
            Else
                Return spec.prefixOrSuffix[x.key].priority
            End If
        End Function
    End Using
End Method
```

This method simply creates the binary file as usual, and the later part of the method is involved in converting the dictionaries to arrays for storage. The highlighted lines use LINQ (Language Integrated Query) syntax to extract data from the dictionary in a similar way to a database, using SQL style queries.

```
List<(string key, string name)> sortedMiddle = new List<(string key, string name)>();
sortedMiddle = spec.middle.Select(kv => (kv.Key, kv.Value.name)).ToList();
sortedMiddle = sortedMiddle.OrderBy(x => spec.middle[x.key].priority).ToList();
writeFile.Write(((short)sortedMiddle.Count));
for (int i = sortedMiddle.Count - 1; i >= 0; i--)
{
    writeFile.Write(sortedMiddle[i].key);
    writeFile.Write(sortedMiddle[i].name);
}
List<(string key, bool middleOnly, bool endOnly, string prefix, string suffix)> sortedSuffixes = spec.prefixOrSuffix.Select(kv => (kv.Key, false, false, kv.Value.prefix, kv.Value.suffix)).ToList();
sortedSuffixes.AddRange(spec.endDependentPrefixOrSuffix.Select(kv => (kv.Key, true, false, kv.Value.middle.prefix, kv.Value.middle.suffix)).ToList());
sortedSuffixes.AddRange(spec.endDependentPrefixOrSuffix.Select(kv => (kv.Key, false, true, kv.Value.end.prefix, kv.Value.end.suffix)).ToList());
sortedSuffixes = sortedSuffixes.OrderBy(x => getPriorityForSuffixSort(x)).ToList();
writeFile.Write(((short)sortedSuffixes.Count));
for (int i = sortedSuffixes.Count - 1; i >= 0; i--)
{
    writeFile.Write(sortedSuffixes[i].key);
    writeFile.Write(sortedSuffixes[i].middleOnly);
    writeFile.Write(sortedSuffixes[i].endOnly);
    writeFile.Write(sortedSuffixes[i].prefix);
    writeFile.Write(sortedSuffixes[i].suffix);
}
writeFile.Close();

int getPriorityForSuffixSort((string key, bool middleOnly, bool endOnly, string prefix, string suffix) x)
{
    if (x.middleOnly)
    {
        return spec.endDependentPrefixOrSuffix[x.key].middle.priority;
    }
    else if (x.endOnly)
    {
        return spec.endDependentPrefixOrSuffix[x.key].end.priority;
    }
    else
    {
        return spec.prefixOrSuffix[x.key].priority;
    }
}
```

This C# code shows the part of the method that creates and saves the array, including the LINQ statements. The Boolean values in the suffix array indicate whether the entry requires the group to be on the end of the molecule / in the middle. The table below shows which values would be used for some examples.

table 1	***middleOnly***	***endOnly***
Ketone (e.g. propan-2-one)	True	False
Aldehyde (e.g. propan-1-al)	False	True
Alcohol (e.g. propan-1-ol, propan-2-ol)	False	False

Groups that are not named differently based on where they are on the molecule (e.g. alcohols) simply use ‘false, false’ as they must be stored in the same array as end-dependent groups. This is because the order of priority is shared across all of these group, so they must be stored together for order to represent priority.

LoadSpecification

The specification is loaded as soon as an IUPAC object is created as the namingspec variable is fundamental for the IUPAC class to function. The main challenge is converting the ordered data back into dictionaries and the extract below shows only that part of the method.

```
Pseudocode
```

```
spec.middle ← new Dictionary<string, (string name, int priority)>()
middleOnlyCount ← readFile.ReadInt16()

For i ← middleOnlyCount - 1 To 0 Step -1
    key ← readFile.ReadString()
    name ← readFile.ReadString()
    spec.middle.Add(key, (name, i+1))
Next
spec.prefixOrSuffix ← new Dictionary<…>()
spec.endDependentPrefixOrSuffix ← new Dictionary<…>()
suffixCount ← readFile.ReadInt16()
For i ← suffixCount - 1 To 0 Step -1
    key ← readFile.ReadString()
    middleOnly ← readFile.ReadBoolean()
    endOnly ← readFile.ReadBoolean()
    prefix ← readFile.ReadString()
    suffix ← readFile.ReadString()
    If middleOnly
        If spec.endDependentPrefixOrSuffix.ContainsKey(key)
            v ← spec.endDependentPrefixOrSuffix[key]
            v.middle.prefix ← prefix
            v.middle.suffix ← suffix
            v.middle.priority ← i + 1
            spec.endDependentPrefixOrSuffix[key] ← v
        Else
            spec.endDependentPrefixOrSuffix.Add(key, ((prefix, suffix, i + 1), ("",
"", -1)))
        End If
    ElseIf endOnly
        If spec.endDependentPrefixOrSuffix.ContainsKey(key)
            v ← spec.endDependentPrefixOrSuffix[key]
            v.end.prefix ← prefix
            v.end.suffix ← suffix
            v.end.priority ← i + 1
            spec.endDependentPrefixOrSuffix[key] ← v
```

```
        Else
            spec.endDependentPrefixOrSuffix.Add(key, (("", "", -1), (prefix, suffix,
i + 1)))
        End If
    Else
        If spec.prefixOrSuffix.ContainsKey(key)
            v ← spec.prefixOrSuffix[key]
            v.prefix ← prefix
            v.suffix ← suffix
            v.priority ← i + 1
            spec.prefixOrSuffix[key] ← v
        Else
            spec.prefixOrSuffix.Add(key, (prefix, suffix, i + 1))
        End If
    End If
Next
```

The loading is unique as 'end' and 'middle' versions of *endDependentPrefixOrSuffix* entries are stored separately in the file (as they have different priorities), so they must be combined into a single dictionary entry here. The program checks if an entry containing the functional group exists in the dictionary and fills the information out accordingly. This is where the Boolean values described in *table 1* are used to describe which dictionary the entry belongs in, and if the entry is end-dependent.

FindHighestPrioritySuffix

This method is the first to use the specification loaded from the binary file and searches the list of *FunctionalGroup* objects to find the highest priority group (using the priority information from the specification). The function then sets additional attributes of the IUPAC class that aid in naming. This is important as the highest priority group decides the final part of the name. The method also sets Boolean attributes of the *IUPAC* class to represent whether the suffix group is end-dependent. This information is represented in the table below.

table 2	***suffixIsMiddle***	***suffixIsEnd***
Not end-dependent (e.g. alcohols)	False	False
Must be on an end (e.g. aldehyde)	False	True
Must be in the middle (e.g. ketone)	True	False

Pseudocode

```
Method FindHighestPrioritySuffix()
    priority ← -1
    For Each fg In groups
        currentPriority ← -1
        If spec.prefixOrSuffix.ContainsKey(fg.GroupFormula)
            currentPriority ← spec.prefixOrSuffix[fg.GroupFormula].priority
        ElseIf spec.endDependentPrefixOrSuffix.ContainsKey(fg.GroupFormula)
            If atoms.IsAnEnd(fg.MainIndex)
                currentPriority ← spec.endDependent…[fg.GroupFormula].end.priority
            Else
                currentPriority ← spec.endDependent…[fg.GroupFormula].middle.priority
            End If
        Else
            Continue (go to the next iteration)
        End If
        If currentPriority > priority
            priority ← currentPriority
        End If
    Next
    For Each v In spec.prefixOrSuffix
        If v.Value.priority = priority
            suffixFormula ← v.Key
            suffixRoot ← spec.prefixOrSuffix[v.Key].suffix
        End If
    Next
    For Each v In spec.endDependentPrefixOrSuffix
        If v.Value.middle.priority = priority
            suffixIsMiddle ← True
            suffixIsEnd ← False
            suffixFormula ← v.Key
            suffixRoot ← spec.endDependent…[v.Key].middle.suffix
```

```
        End If
        If v.Value.end.priority = priority
            suffixIsMiddle ← False
            suffixIsEnd ← True
            suffixFormula ← v.Key
            suffixRoot ← spec.endDependent…[v.Key].end.suffix
        End If
    Next
End Method
```

The method begins by iterating though every *FunctionalGroup* to find the maximum priority, then doing so again to find the suffix group. This has since been improved to include only one iteration by utilising the fact that each group has a unique priority, since priority is normalised from the binary file which stores it as an 'order'. The improved version is also simpler and easier to follow.

Pseudocode

```
Method FindHighestPrioritySuffix()
    maxPriority ← -1
    For Each fg In groups
        currentPriority ← -1
        If spec.prefixOrSuffix.ContainsKey(fg.GroupFormula)
            currentPriority ← spec.prefixOrSuffix[fg.GroupFormula].priority
            If currentPriority > maxPriority
                maxPriority ← currentPriority
                suffixIsMiddle ← False
                suffixIsEnd ← False
                suffixFormula ← fg.GroupFormula
                suffixRoot ← spec.prefixOrSuffix[fg.GroupFormula].suffix
            End If
        ElseIf spec.endDependentPrefixOrSuffix.ContainsKey(fg.GroupFormula)
            If atoms.IsAnEnd(fg.MainIndex)
                currentPriority ← spec.endDependent…[fg.GroupFormula].end.priority
                If currentPriority > maxPriority
                    maxPriority ← currentPriority
                    suffixIsMiddle ← False
```

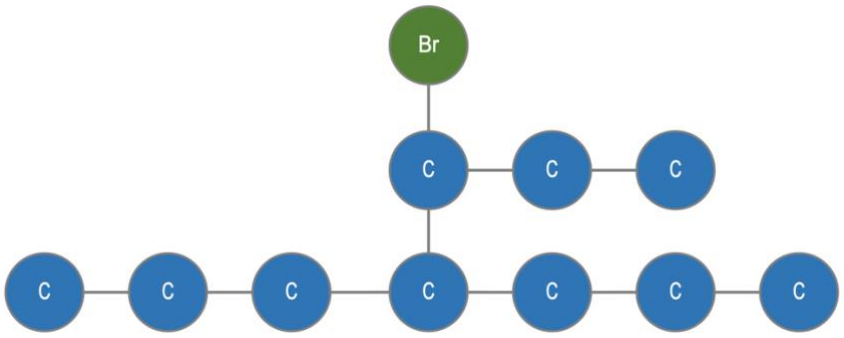
```
                    suffixIsEnd ← True
                    suffixFormula ← fg.GroupFormula
                    suffixRoot ← spec.endDependent…[fg.GroupFormula].end.suffix
                End If
            Else
                currentPriority ← spec.endDependent…[fg.GroupFormula].middle.priority
                If currentPriority > maxPriority
                    maxPriority ← currentPriority
                    suffixIsMiddle ← True
                    suffixIsEnd ← False
                    suffixFormula ← fg.GroupFormula
                    suffixRoot ← spec.endDependent…[fg.GroupFormula].middle.suffix
                End If
            End If
        End If
    Next
End Method
```

This method continues to iterate through each group, then identifies if it could be represented in the suffix of the name. If so, the priority of the group is compared with the priority of the current suffix group, and if it is higher, the previous information and priority is overwritten.

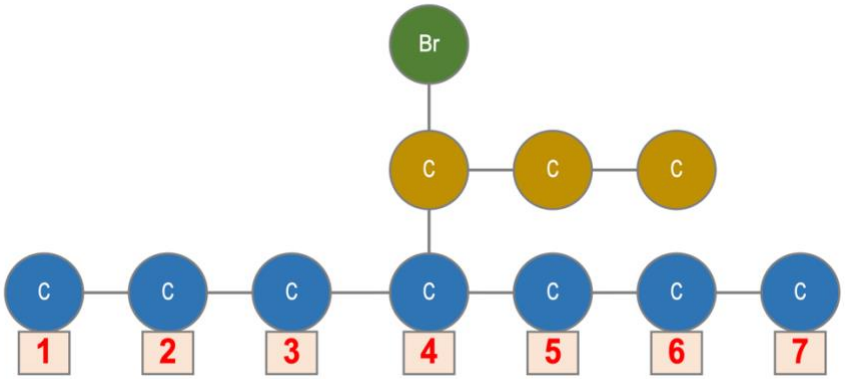
IUPAC

The most important step for naming a molecule is finding the parent carbon chain, which is the main task of the *IUPAC* constructor. The parent chain is the backbone running through the molecule, as the position where each functional group or branch falls on this chain determines the number they are given in the final name. This chain needs to be as long as possible and contain all the functional groups such that the groups find themselves at the lowest possible index (objective **2.b**). The *FindEveryLongestPath* method uses every 'end' of this molecule (atom bonded to only one other atom). The path between every pair of ends is found and is returned to *IUPAC* so the ideal chain can be found from these options. The process of finding the optimal chain involves removing every suboptimal chain and repeating until one remains (using several metrics).If we define a chain as a path between two ends, a graph with n 'end' carbon atoms has n^2 chains and (n^2 – n) nontrivial chains (more than one element). Branches are paths of carbon atoms that are not part of the parent chain.

This diagram represents an *ElementGraph*. The blue will represent the carbon chain, yellow will represent carbon branches and the green represents functional groups (in this case the C-Br group). This graph has 3 'end' carbons so 3^2 chains can be formed. If we discard the trivial paths of one element, there are 3^2 - 3 paths. Hydrogens have been ignored for simplicity.



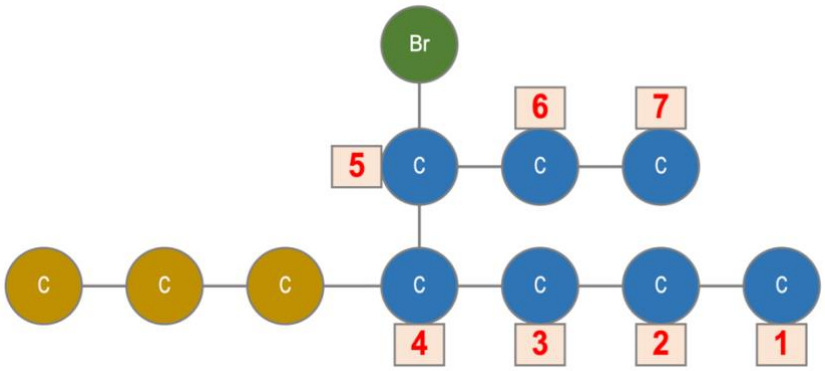
First, the chains are narrowed down by ensuring branches contain no functional groups, i.e. narrowing by branches.



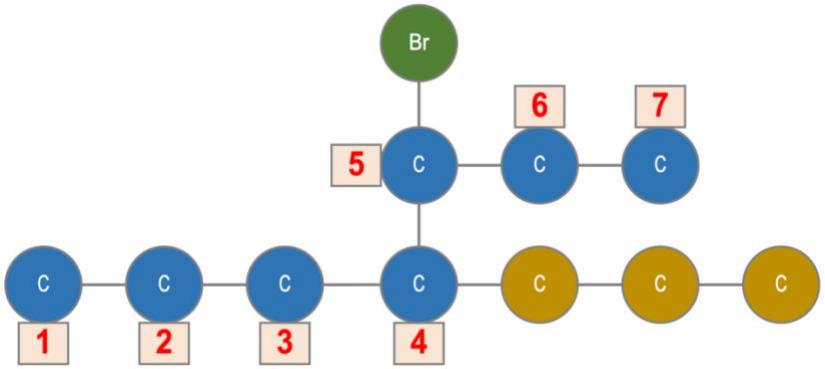
This shows an example of a chain that could be generated through this molecule as a candidate to be the parent carbon chain. The numbers represent the position the carbon takes in the chain. This is also known as the carbon number.

This chain would be rejected as the functional group is not included on the main chain, and functional groups cannot be placed on branches.

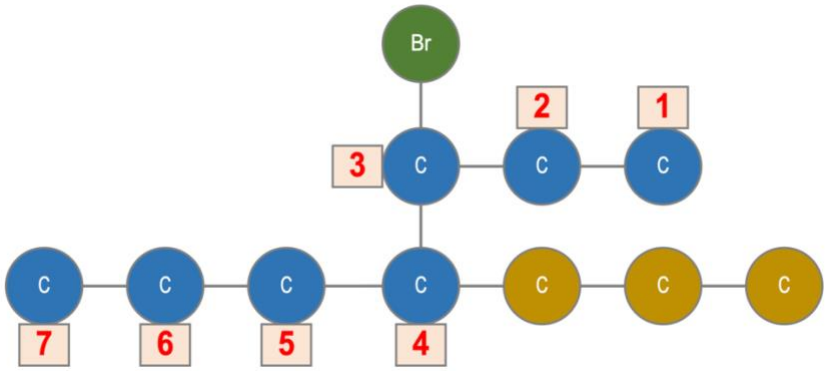
The chains are also narrowed down by length. Then, the chains are narrowed down to ensure the most important groups have the lowest carbon numbers (**objective 2.b.**). This process starts with the suffix groups, then the middle, then the prefix. In this example we assume C-Br is a prefix group (this would be decided by the *namingSpec*), so we skip the suffix and middle and only narrow down the prefix elements. Carbon branches are automatically named as prefix elements, so also considered here. The chains below are not an exhaustive list as only a few examples have been shown.



The spatial positioning of the atoms is arbitrary (as the molecule is a graph), so this chain is also valid as it passes through the functional group too. The C-Br group is on carbon number 5 and the branch comes from carbon number 4. The prefixes on this chain would be named “5-bromo-4-propyl…”.



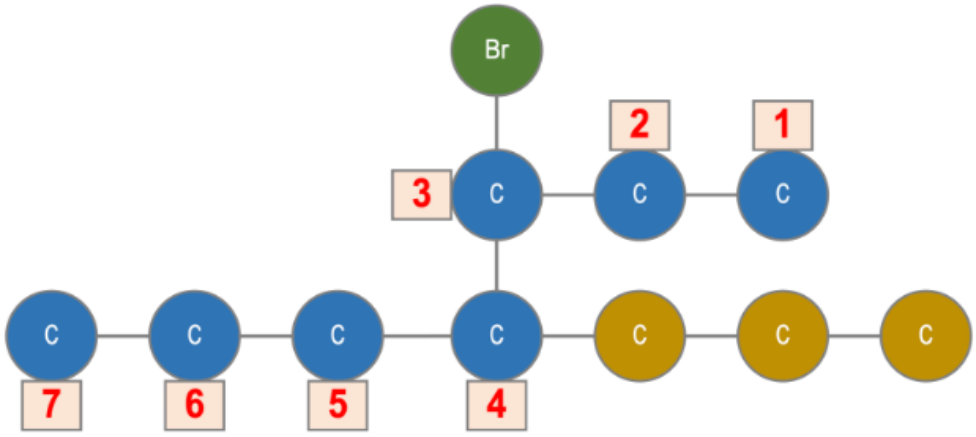
This chain would also include the functional group in the chain, so would be valid. The group is on carbon number 5 and the branch is on carbon number 4. The prefixes on this chain would also be named "5-bromo-4-propyl…".



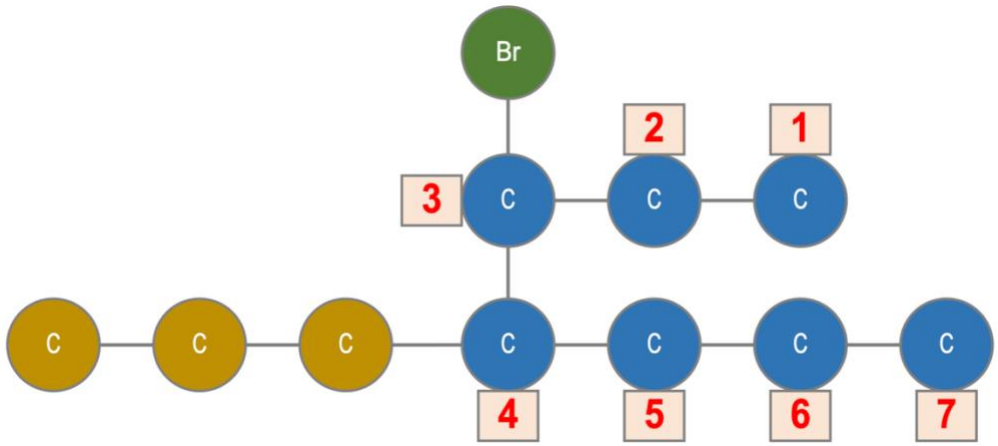
It is also important to note that opposite directions are stored separately. This means that chains can be paired together as forwards and backwards versions, both being valid. This chain is the reverse of the chain before it. The group is on carbon number 3 and the branch is on carbon number 4. The prefixes on this chain would be named "3-bromo-4-propyl…".

The lowest possible numbers for the group and branch are 3 and 4, so any chains that give prefix elements higher numbers are eliminated here. From the paths shown above, the first and second give prefix elements numbers that sum to 9, whereas the last one sums to 7, so the first two are eliminated.

The examples below list both chains that made it this far through the elimination.



This chain has length _



This chain has length

These two chains are in fact isomorphic and identical to each other, as you can imagine the lower half of the molecule pivoting around the bond connecting carbons 3 and 4. Once this far into the elimination process, any remaining chains are isomorphic and therefore produce the same name, or numbers could be switched around between prefix elements. These two chains would produce the same name of "3-bromo-4-propylheptane", which is the ideal name for the molecule.

NarrowDownChainsByBranches

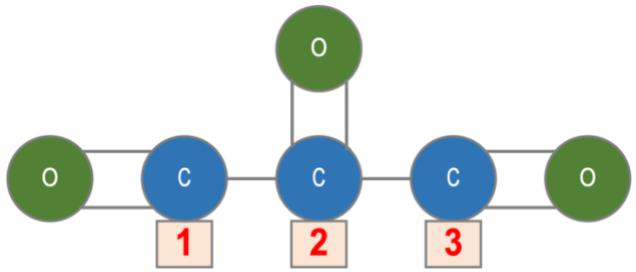
This is the first method which narrows down the possible parent chains to find the ideal one. This method also finds all the branches to pair with each chain. This method enforces the rule that functional groups cannot lie on branches. If a chain breaches this rule (the chain's branches contain functional groups), the chain and its branches are removed from the list.

```
private List<(List<int> chain, List<List<int>> branches)> NarrowDownChainsByBranches(List<List<int>> chains)
{
    List<(List<int> chain, List<List<int>> branches)> allChainsAndBranches = new List<(List<int> chain, List<List<int>> branches)>();
    foreach (List<int> chain in chains)
    {
        List<List<int>> branches = atoms.FindBranches(chain);
        allChainsAndBranches.Add((chain, branches));
    }
    for (int i = allChainsAndBranches.Count - 1; i >= 0; i--)
    {
        List<List<int>> branches = allChainsAndBranches[i].branches;
        if (!CheckBranchValidity(branches))
        {
            allChainsAndBranches.RemoveAt(i); //branch invalid, so whole chain invalid
        }
    }
    return allChainsAndBranches;
}
```

The other *NarrowDownChainBy…* methods do similar things, but count the carbon numbers of different groups instead to minimise these numbers (as shown above).

FindSuffixCarbons

After the one parent chain is found, the program must construct the name for each part of the molecule (prefix, middle, suffix). The suffix is easiest to form as it only represents one group (the highest priority group) as opposed to the middle or prefix which need to represent many groups in specific orders. If the suffix group is in the molecule more than once, the method returns all the carbon numbers where the group was found. The *FindHighestPrioritySuffix* method would have set information such as the formula of the suffix group and the Boolean values from table 2 which are used to identify the suffix group in this method.



This image (hydrogens have been omitted for simplicity) depicts an *ElementGraph* with three 'C=O' groups, one on each carbon, and the numbers show the carbon number of each carbon. If the C=O group is end-dependent, a C=O on the end of a molecule is essentially a completely different group from a C=O group in the middle of a molecule. This means the groups on carbons 1/3 and the group on carbon 3 are essentially different and are named separately.

Depending on the priorities defined in the *namingSpec*, the *FindHighestPrioritySuffix* method would have set attributes of IUPAC differently.

If C=O middle priority is higher

suffixFormula	"C=O"
suffixIsMiddle	True
suffixIsEnd	False

If the suffix group is the middle one, the *FindSuffixCarbons* method would simply return { 2 }, as the carbon numbered 2 is the only one carrying the highest priority part of the molecule. The suffix would ultimately be "…-2-one" and the other two groups on carbons 1 and 3 would be added to the prefix as "1,3-diformyl…".

If C=O end priority is higher

suffixFormula	"C=O"
suffixIsMiddle	False
suffixIsEnd	True

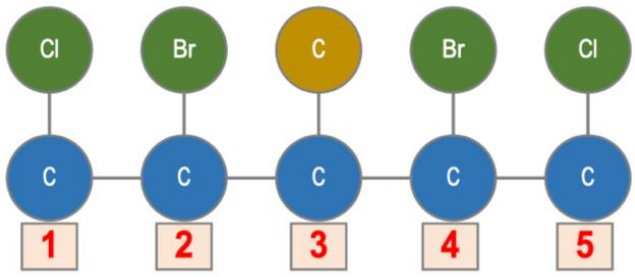
If the suffix group is the end ones, the *FindSuffixCarbons* method would return { 1, 3 }, as the carbons numbered 1 and 3 both carry the parts of the molecule with equally highest priority. The suffix would ultimately be "1,3-dial" and the other group on carbon 2 would be added to the prefix as "2-oxo…".

For a group to be included in the suffix, it would need to match both its formula and its end/middle status to that of the high priority found by *FindHighestPrioritySuffix*. This is checked in the method below and has been broken down into three Boolean parts for readability. The phrase "path.IndexOf(group.MainIndex) + 1" finds the carbon number of the carbon the group is attached to using the optimal chain the program decided on earlier.

```
private List<int> FindSuffixCarbons(List<int> chain)
{
    List<int> indexes = new List<int>();
    foreach (FunctionalGroup group in groups)
    {
        bool endsAreInvolved = suffixIsMiddle || suffixIsEnd;
        bool groupAndSuffixAreMiddle = suffixIsMiddle && !atoms.IsAnEnd(group.MainIndex);
        bool groupAndSuffixAreEnd = suffixIsEnd && atoms.IsAnEnd(group.MainIndex);
        if (group.GroupFormula == suffixFormula && (!endsAreInvolved || groupAndSuffixAreMiddle || groupAndSuffixAreEnd))
        {
            indexes.Add(chain.IndexOf(group.MainIndex) + 1);
        }
    }
    return indexes;
}
```

FindPrefixCarbons

This method and the *FindMiddleCarbons* method perform a similar task to the *FindSuffixCarbons* method, except they return a dictionary mapping strings to lists of integers. This is because the prefix or middle can contain multiple parts. This method is especially complicated as every group named under *prefixOrSuffix* that did not get the suffix position is added to the prefix. Every branch is also included in the prefix, and so are all the groups named under *prefixOnly*.



This molecule has four groups (green) and one branch (yellow). Let's assume the *namingSpec* dictionary *prefixOnly* contains the entries {"C-Cl" -> "chloro", "C-Br" -> "bromo", …}. Then all four groups are included in the prefix, so the *FindPrefixCarbons* method will return the following dictionary mapping parts of the prefix to carbon numbers they come from:

{"chloro" -> {1, 5}, "bromo" -> {2, 4}, "methyl" -> {3}}

Pseudocode

```
Method FindPrefixCarbons(chain, branches)
    prefixesAndIndexes ← new Dictionary<string, List<int>>()
    For Each group In groups
        name ← ""
        carbonNumber ← chain.IndexOf(group.MainIndex) + 1
        If group.GroupFormula ≠ suffixFormula
            If spec.prefixOnly.ContainsKey(group.GroupFormula)
                name ← spec.prefixOnly[group.GroupFormula]
            ElseIf spec.prefixOrSuffix.ContainsKey(group.GroupFormula)
                name ← spec.prefixOrSuffix[group.GroupFormula].prefix
            End If
        End If
        If spec.endDependentPrefixOrSuffix.ContainsKey(group.GroupFormula)
            If atoms.IsAnEnd(group.MainIndex) AND Not (suffixIsEnd AND
group.GroupFormula == suffixFormula)
                name ← spec.endDependent…[group.GroupFormula].end.prefix
            ElseIf Not atoms.IsAnEnd(group.MainIndex) AND Not (suffixIsMiddle AND
group.GroupFormula == suffixFormula)
                name ← spec.endDependent…[group.GroupFormula].middle.prefix
            End If
        End If
```

```
        If name ≠ ""
            If prefixesAndIndexes.ContainsKey(name)
                prefixesAndIndexes[name].Add(carbonNumber)
            Else
                prefixesAndIndexes.Add(name, new List<int> { carbonNumber })
            End If
        End If
    Next
    For Each branch In branches
        carbonNumber ← chain.IndexOf(branch[0]) + 1
        name ← spec.alkylNames[branch.Count - 1] + "yl"
        If prefixesAndIndexes.ContainsKey(name)
            prefixesAndIndexes[name].Add(carbonNumber)
        Else
            prefixesAndIndexes.Add(name, new List<int> { carbonNumber })
        End If
    Next
    Return prefixesAndIndexes
End Method
```

The first part of the method works to add the non-suffix groups to the prefix. First, the groups who have a different formula to the suffix group are counted, as this is a sufficient condition. Next, the groups who have the same formula as the suffix are considered if they are also end dependent. These groups may still be different from the suffix group if they are on a different end, so this is checked. The conditional statements for end-dependent prefix components have been explained below.

- The *namingSpec* indicates a group must be named in an end-dependent way
  - The group is on the end of the molecule, so is an 'end' group. The group is not identical to the suffix group- NOT (the current group and suffix have the same formula AND the suffix is also on the end of the molecule).
    - Get the 'end' 'prefix' way of naming this group.
  - The group is in the middle of the molecule, so is a 'middle group. The group is not identical to the suffix group- NOT (the current group and suffix have the same formula AND the suffix is also in the middle of the molecule).
    - Get the 'middle 'prefix' way of naming this group.

The pink highlighted section checks if this prefix component has already been encountered before. If the entry for this prefix name already exists, the carbon number found is added to the list of all the others. If it is the first time the program is finding a group named in this way, a new entry is added to the dictionary.

The final part of the method adds all the branches to same dictionary. The name of the branch (that represents the length) is the key, and a list of carbon numbers where branches of that length meet the main carbon chain is the value. The branches names are stacked in the same way all other things are, so the pink highlighted code is repeated here.

NameSegment

This method takes the output of the *FindPrefixCarbons*, *FindMiddleCarbons* and *FindSuffixCarbons* methods and converts the dictionary into a string that represents that segment of the molecule's name. The example dictionary output of *FindPrefixCarbons* {"chloro" -> {1, 5}, "bromo" -> {2, 4}, "methyl" -> {3}} would return the string "2,4-dibromo-1,5-dichloro-3-methyl". Instead of the prefix containing "2-bromo-4-bromo-1-chloro-5-chloro…", the similar named components are joined together and a prefix is added to indicate how many have been combined (usually "di", "tri", "tetra"). This numerical prefix is taken from *namingSpec.numericalPrefixes*. The names joint together (e.g. "bromo", "chloro", "methyl") are ordered alphabetically in the name, and this alphabetisation ignores the numerical prefix (**objective 2.c**).

```
private string NameSegment(Dictionary<string, List<int>> namesAndCarbonNumbers)
{
    List<(string numbers, string name)> names = new List<(string, string)>();
    foreach (KeyValuePair<string, List<int>> groupData in namesAndCarbonNumbers)
    {
        string name = groupData.Key;
        List<int> carbonNumbers = groupData.Value;
        carbonNumbers.Sort();
        string numericalPrefix = spec.numericalPrefixes[carbonNumbers.Count];
        string numbers = "";
        for (int i = 0; i < carbonNumbers.Count; i++)
        {
            numbers += carbonNumbers[i];
            if (i != carbonNumbers.Count - 1)
            {
                numbers += ",";
            }
        }
        numbers += numericalPrefix;
        names.Add((numbers, name));
    }
    names = names.OrderBy(x => x.name).ToList();
    string nameSegment = "";
    for (int i = 0; i < names.Count; i++)
    {
        nameSegment += names[i].numbers + names[i].name;
    }
    return nameSegment;
}
```

This function produces the prefix, middle and suffix of the name. These parts are simply added together with the name of the chain length to produce the final name of the molecule.

FormatName

This method deals with the dashes and vowel rules that IUPAC names must follow (**objective 2.c.**). Dashes simply go between numbers and letters, so "ethane1,2diol" becomes "ethan-1,2-diol". Some names are difficult to pronounce, so a vowel is removed from the end of some naming components if two consecutive vowels cause problems. This is better illustrated if the carbon numbers are ignored. For example, "methan**e**ol" sounds unnatural, so the "e" is removed to form "methanol". Some examples require the "e", for example "methan**e**diol" is preferred to "methandiol". This matter is overcomplicated by the fact that only some naming components express this behaviour. For example, "nonanoct**a**ol" has two adjacent vowels, but the "oct**a**" keeps its "a". This matter is very niche and hard to find information on, but is part of formal IUPAC documentation, and exam board specifications. To indicate which naming components have optional vowels, special syntax is included in the *namingSpec* to allow the user to customise this and indicate where vowel sensitivity should be enforced. The following syntax is used: "*<naming component>*|*<optional vowel>*". If the *<optional vowel>* is followed by another vowel, the *<optional vowel>* is removed from the word. The bar symbol indicates the next character is an *<optional vowel>*, so this can be used in select circumstances.

```
private string FormatName(string name)
{
    Regex vowels = new Regex(@"\|(.)(?=[^a-z]*[aeiouy])");
    Regex startDashes = new Regex(@"(\w)(\d)");
    Regex endDashes = new Regex(@"(\d)(\w)");
    name = vowels.Replace(name, "");
    name = name.Replace("|", "");
    name = startDashes.Replace(name, (m) => m.Groups[1].Value + "-" + m.Groups[2].Value);
    name = endDashes.Replace(name, (m) => m.Groups[1].Value + "-" + m.Groups[2].Value);
    return name;
}
```

The *vowels* expression matches a "|" symbol followed by any character *(group 1)* and a lookahead ensures that the next alphabetical character is a vowel. The entire matched section (this does not include the lookahead, so is only the bar and optional character) is removed from the name. After this process is complete, all the "|" symbols are removed from the name, as some may remain for when the optional vowel proceeded a consonant and did not match the *vowels* expression.

The *startDashes* expression matches an alphanumerical character (*group 1)* meeting a numerical character (*group 2*), and the *endDashes* expression matches the inverse. The entire matched section is replaced with the (*group 1*) + "-" + (*group 2*). This essentially adds a "-" between the groups, as required.

CODE BIBLIOGRAPHY

Recursive depth first search: Modified from ChatGPT 3.5
C# to Pseudocode: Modified from ChatGPT 3.5
Finding first empty index of array: https://stackoverflow.com/questions/5400895/using-c-sharp-linq-to-return-first-index-of-null-empty-occurrence-in-an-array (Accessed 26/03/2024)
Remove items from list in one pass through: https://stackoverflow.com/questions/1582285/how-to-remove-elements-from-a-generic-list-while-iterating-over-it (Accessed 26/03/2024)