# BERYL RegEx

Emerald Creations

BERYL regEx

# Chapter 1
# Introduction

Beryl RegEx engine is intended as a more powerful system than Perl-Compatible Regular Expression (PCRE) with conditions, loops, and variable access. The main purpose of Beryl RegEx is pattern-matching. Everything else plays a support role.

## 1.1. Syntax

Beryl syntax can be summarizes as:

```
REFERENCE : Type & Commands : Search String or Pattern : Additional
   Code;;
```

Example:

```
1 : EX : Foo : || {} ||;;
2 : EQ { IGNORE CASE; IGNORE DIGITSEPARATOR; } {}
          : fOoBArbAZ %MATCH @POSITION
          : ||
              match_lowerCase := toLower(%MATCH);
              function F(s)

                  for i : 1 to length(%MATCH) do
                      s : s + ‘‘a’’;
                  end for
                  return s;
              end function

              #q : F(‘‘b’’);

              #qq : q;
              #l : length(q);
              #m : length(%MATCH);
           ||;;
3 : ER {} {OFFSET 4; RANGE 20;} : {
          {{ REPEAT 2 }} {{
          1 : EQ {IGNORE CASE} : BAZ ;;
          }}
        } : || ||;;
```

### 1.1.1. Important Concepts

The main goal of Beryl is to match patterns in serial data, usually given in string. Example, we might want to search for the substring Foo in the string FooBar . Throughout the document, we will refer to Foo or its equivalent as **Search String**. The string FooBar will be referred to as **Test String**.

Sometimes, it might be the case, that the Search String was not specified exactly. We might want to search for F*o, where the asterisk is an wildcard for any character (or multiple characters) possible. In the Test String, we find that the character o fits the wildcard. In this example, F*o does play the role of the Search string, but it is not exactly defined as a Search String. This will be called a **Pattern**. The substring that matches the pattern, namely Foo, will be referred to as **Match**.

In case of an exact definition, the substring that matches the Search String is exactly the same as the Search String. The former can also be called a Match.

## 1.1.2. Structure of a Beryl regEx instruction

Each Beryl instruction has 4 segments. These are:

1. Reference
2. Match Type, Type Control, and Generic Control information
3. Search String or Pattern, with default information extraction instructions
4. Additional code

#### 1.1.2.1. Separation, Enclosure, Delimitation and Termination

Each segment of a Beryl regEx instruction is separated from its neighbors with :, a colon. Every complete Beryl regEx instruction ends with ;;, double semicolon. Inside the 4th segment, the additional code, each line of code must end with ;, a single semicolon.

Type control and Generic control instructions are also terminated by a single semicolon. We will explain what these are later. Type control information are enclosed in a pair of curly braces, {} and Generic control information are enclosed in a separate pair as well. These pair of curly braces are delimited by one or more whitespace character.

Search string or pattern can be complex and nested. They have their own set of enclosures, as will be discussed later. Additional code is enclosed between a pair of double vertical bars || as shown above.

#### 1.1.2.2. Invoking a Beryl regEx command

Usually, a different programming language will be invoking or calling beryl regEx. Assuming it is an object oriented (fictional) language, a call might look like this:

```
testString := ‘‘FooBarBaz’’;

 regExObj1 := new regExHandler(
             \
               42 : EX : Bar : ||{}||;;
             \
            );

         M := regExObj1.match_againstTestString(testString);

assert(M.status = True);
assert(M.results[42] = ‘‘Bar’’);
```

Here, we have a Test String, with the content FooBarBaz. A new regExHandler object is created in an OOP paradigm. The constructor accepts a string, which is here shown enclosed in a pair of backslash. A real, non-fictional OOP language will probably not allow it.

The regExHandler object exposes a method match_againstTestString. This method accepts the Test String as an argument. It returns an object in return, here stored in the variable M.

The variable M has a field status, which reports whether the search was successful or not. Another field, results, stores a dictionary where all the matches will be stored. This is merely an example. In practice, we can also write:

```
testString := ‘‘FooBarBaz’’;

 regExObj1 := new regExHandler(
              \
                42 : EX : Bar : ||{}||;;
              \
            );

         regExObj1.testString := testString;
         regExObj1.match_againstTestString();
         M := regExObj.get_results(42);

assert(M.status = True);
assert(M.result = ‘‘Bar’’);
```

The exacts are left to the implementing language. The Lazarus/Freepascal implementation uses the second variant. Note that we already know the references of each command, when we are programming the regExHandler. Therefore, we can always call get_results with absolute certainty of the reference.

## 1.2. Segments of a Beryl Instruction

Now we take a closer look at the example Beryl regEx instruction. We already mentioned the 4 segments.

### 1.2.1. The Reference

The results field in M is a dictionary, where each key is an integer. Every time Beryl instruction succeeds in finding a match, the match is inserted into the results field, with the reference mentioned in the instruction as the corresponding key.

For now, the reference is set as an integer. This can change in future. The regEx itself could be supplied as a string, implying that the reference also remains a string. However, the regEx engine will have to convert the string to an integer will populating the result.

### 1.2.2. The Type and Type Commands

There are several type of patterns and Search Strings that can be used. It is also necessary to specify additional information. The syntax and the type system is designed specifically to harmonize the placement of such instructions in various use cases.

#### 1.2.2.1. Types

Type refers to the categorization of Search Strings. Even though there will be additional capability to perform simple scripting, such script will not impose a type system.

The Search String types are :

1. EXACT (EX)
2. EQUIVALENT (EQ)
3. ENSEMBLE (ES)
4. ECHELON (EC)
5. EVERYTHING (EV)
6. ENQUIRY (EN)
7. ENCORE (ER)
8. EMPTY (EY)

The details will be discussed in later sections. The second section of a regEx instruction contains the associated type designator as the first space delimited item.

#### 1.2.2.2. Type specific commands

Type specific commands immediately follow enclosed in a pair of curly braces, {} after the type designator. These instructions modify the behavior of the regEx engine. In case there are multiple such instructions, they are terminated by a single semicolon, ;. These instructions will be discussed in detail later. In case there are no Type specific commands, but General commands (next section) are needed, then a pair of empty pair of curly braces, {}, with or without white space in between will be supplied before the General commands are specified. Conversely, if Type specific commands are supplied, followed by a pair of empty curly braces, then the later is simply ignored.

#### 1.2.2.3. General commands

There are some commands that apply to every type. These commands are evaluated before the type specific commands. These commands are:

**Offset** This command describes where the regEx engine will start matching in the Test String. Example:

```
testString := ‘‘FooBarBaz’’;

 regExObj1 := new regExHandler(
                 \
                   1 : EX {} {OFFSET 2;} : Bar : || {} ||;;
                 \
               );

         M := regExObj1.match_againstTestString(testString);

assert(M.status = True);
assert(M.results[42] = ‘‘Bar’’);
```

OFFSET is given as 2. In this example, the regEx engine will simply ignore the first 2 characters. Therefore, the Test String in this example appears as oBarBaz to the matching algorithm.

If offset is not specified, then it is assumed to be 0.

If offset is equal or goes beyond length of Test String minus length of Search String, then no search is performed. However, no error is thrown. In the current implementation, only a warning is shown.

If offset is negative, an error is thrown.

**Range** Range specifies how far in the Search String the regEx matching algorithm can look into. Example:

```
testString := ‘‘FooBarBaz’’;

 regExObj1 := new regExHandler(
             \
              1 : EX {} {OFFSET 2; RANGE 2;} : Bar : || {} ||;;
             \
            );

         M := regExObj1.match_againstTestString(testString);

assert(M.status = False);
```

This example sets the regEx engine to look no further than 2 characters after the offset, which is also 2. So the algorithm starts looking at the second o of the Test String, and looks at no further than 2 characters after that. Thus the total examined part of the Test String here is oB. Of course, this leads to a match failure.

If Range is not specified, then it is assumed to be the whole length of the Test String.

If Range is negative, or less than the length of the Search String, an error is thrown in the current implementation.

If Range goes beyond the length of the Test String, then it is reduced to the length of the Test String. No error is thrown.

**Fence** Fence sets a limit on how far the search algorithm can go, starting from the right end of the Test string. Example:

```
testString := ‘‘FooBarBaz’’;

 regExObj1 := new regExHandler(
             \
              1 : EX {} {FENCE 2; } : Baz : || {} ||;;
             \
            );

         M := regExObj1.match_againstTestString(testString);

assert(M.status = False);
```

Here, the algorithm will see FooBarB as the Test String.

If Fence is not specified, it is assumed to be 0.

If fence is negative, an error is thrown.

If fence is equal or larger than the length of the Search String, no error is thrown in the current implementation. A warning is generated, that no match is possible.

**Retreat** It is the same as range, but computed backwards from the Fence point of the Test String. Example:

```
testString := ‘‘FooBarBaz’’;

 regExObj1 := new regExHandler(
             \
               1 : EX {} {FENCE 2; RETREAT 4;} : Baz : || {} ||;;
             \
           );

         M := regExObj1.match_againstTestString(testString);

assert(M.status = False);
```

Here, the algorithm will see BarB as the Test String.

Retreat behaves very similarly to Range, but in the opposite direction.

**Anchor** If it is needed, that the Search String appears at a specified index of the Test String, then the Anchor command can be used. Example:

```
testString := ‘‘FooBarBaz’’;

 regExObj1 := new regExHandler(
             \
               1 : EX {} {ANCHOR 4;} : Bar : || {} ||;;
             \
           );

         M := regExObj1.match_againstTestString(testString);

assert(M.status = False);
```

Here, the Anchor command requires the Search String Bar to appear at Index 4 of the Test String. The substring at Index 4 of the Test String is arBaz. The match thus fails. Indexing starts at 0 with Anchors.

Anchor command can be specified with a negative argument, then the algorithm starts at a position counting from the end of the Test String, given by the absolute value of the argument. Example:

```
testString := ‘‘FooBarBaz’’;

 regExObj1 := new regExHandler(
             \
               1 : EX {} {ANCHOR -6;} : Bar : || {} ||;;
             \
           );

         M := regExObj1.match_againstTestString(testString);
```

```
assert(M.status = True);
assert(M.results[1] = ‘‘Bar’’);
```

Here, the Anchor command requires the Search String Bar to appear at Index -6 of the Test String, counting from the End of the String. The backword counting Index starts at -1. The substring at Index -6, counting backwards from the end of the Test String is BarBaz. The match thus succeeds. If not specified, the value of Anchor is **Undefined**.

**Combination** These values can lead to conflicts. In order to resolve, the following rules are implemented.

- With Offset and Range both specified, if Range goes beyond the length of the Test String minus Offset, then it is reduced to the length of the Test String minus Offset.
- With Offset and Fence pecified, if Offset goes beyond Fence, then an Error is Thrown.
- With Range and Fence both specified, in the current FreePascal implementation, Range is overwritten if needed, and the Range algorithm is used. (see below)
- If Range does not go beyond Fence, then Fence is ignored. Range is not overwritten.
- If Range goes beyond Fence, then Fence gets priority, regardless of Offset. Range is overwritten with length of the Test String minus Fence, and the Range algorithm is used. This is equivalent to the case if there was no Range specified, but the same value of Fence was given.
- With Offset, and Retreat Specified, if Retreat goes beyond Offset, then Offset gets priority, regardless of Fence.
- Every other error leads to an Error condition. **This can change in future.**
- The Anchor command can’t be combined with an Offset or Fence command.

It should be noted that currently, Right to Left languages are not supported.

### 1.2.3. The Match Instructions

Immediately following the Type and Type commands comes the Match instructions. Here, the Search String or similar info are specified. The details will be described later.

In order to programmatically control the behavior of the algorithm, some information and the current match may be needed. For this reason, several capture variables can be mentioned here as well. These are:

**%MATCH** Any variable, that is prefixed with the symbol % can be used to capture the exact match.

**@POSITION** Any variable, that is prefixed with the @ symbol, is used to capture the index of the Search String in the Test String.

**_COUNT** Any variable, that is prefixed with the _ symbol, is used to capture the number of matches found. This is only applicable with ENCORE or EVERYTHING matches.

**$CONDITION** Any variable, that is prefixed with the $ symbol, is used to capture the supplied conditions, and their truth value in case of a conditional match. This is only applicable with ENQUIRY matches.

These variables can be specified immediately after the Search String or equivalent, in any sequence. The prefixes must be respected. Otherwise, the regEx will lead to an error.

### 1.2.4. Additional Code

Every match instruction may contain additional code, in its final segment, enclosed between a pair of vertical bars || as mentioned before.

The additional code is implemented in a simple scripting language. Variables are untyped. Usual conditions ( if-then-else ), loops ( for ), switch, etc are available. Functions are also supported.

Various predefined functions for number and string manipulation are also available. Details will be specified later.

### 1.2.5. Escape Sequences

A number of punctuation marks are used as special control characters (tokens) in Beryl regEx already. For this reason, to indicate the character without its special role, as simply a string or a character without any contextual meaning, it needs to be specifically marked as such. The marking is usually done by prefixing a \ (backslash) character. The following are escaped.

| | | | |
|---|---|---|---|
| Semicolon : \; | Colon : \: | Open Parenthesis : \( | Open Brace : \{ |
| Vertical Bar : \\| | At symbol : \@ | Close Parenthesis : \) | Close Brace : \} |
| Open Bracket : \[ | Percent Symbol : \% | Dollar Symbol : \$ | Backslash : \\ |
| Close Bracket : \] | Underscore Symbol : _ | Hash Symbol : \# | Exclaimation : \! |
| Tilda : \~ | Slash symbol : \/ | Plus symbol : \+ | Star symbol : * |

# Chapter 2

# EXACT MATCHES

The most basic matching method is an Exact match. In here, an exact copy of the Search String is sought inside the Test String. An exact match, by itself, captures only **one** copy of the Search String. For multiple copies, an ENCORE Match is needed (described later). The exact match is designated by the type keyword **EX** or **EXACT**.

## 2.1. Special Considerations

### 2.1.1. Escaped Characters

The characters inside an exact match may be escaped as needed. However, when populating the result, the escape symbols prefixing the characters will be removed, unless the host language also requires the same escape prefix.

### 2.1.2. Type Commands

This type does not have any type specific commands. Therefore, in order to supply any generic command, a pair of curly braces, {} must be supplied.

### 2.1.3. Pattern or Match Length

In many case, we will see, that the algorithm matches only one character of a category or a range. In case of an exact match, the entire supplied pattern is considered one symbol, with symbol length 1. Notice, that symbol length is different from the actual length in terms of character. Exactly one copy of it is matched.

### 2.1.4. String Types

**TBD, UNICODE**

# Chapter 3
# EQUIVALENT MATCHES

The next type is Equivalent Matches. In here, a similar (Equivalent) copy of the Search String is sought inside the Test String. An equivalent match, by itself, also captures only **one** copy of the Search String. For multiple copies, again, an ENCORE Match is needed (described later). The equivalent match is designated by the type keyword **EQ** or **EQUIVALENT**.

## 3.1. Type Specific Commands

This type accommodates for type specific commands. The most important ones are:

### 3.1.1. IGNORE commands

These commands can be used to perform various *insensitive* comparisons. For example, ignore case will perform a case insensitive comparison. More specifically, at present the current options are available.

#### 3.1.1.1. IGNORE CASE

This command performs a match while ignoring the case of both the Search String and (targeted segment of) the Test String. Example:

```
testString := ‘‘FooBarBaz’’;

 regExObj1 := new regExHandler(
                \
                  42 : EQ {IGNORE CASE;} {OFFSET 3;} : bAr : ||{}||;;
                \
             );

         M := regExObj1.match_againstTestString(testString);

assert(M.status = True);
assert(M.results[42] = ‘‘Bar’’);
```

First, the general commands are evaluated, where we have the Test String FooBarBaz with offset 3. Therefore, the matching algorithm only sees BarBaz as Test String. Next, the type specific commands are handled. Here, we have IGNORE CASE; as supplied command. This will cause the Test String to appear as barbaz, and the Search String (note the capitalization as shown in the example pseudo-code) will appear as bar. Of course, this leads to a match. IGNORE CASE; is symmetric. That is, the match algorithm ignores the case both in the Test String as well as the Search String.

IGNORE CASE; result returns the substring as found in the Test String, not the supplied Search String. For example, here Bar is returned, even though bAr was supplied. The ignore case command is based on the concept of case in the European Languages. For other languages, more research is necessary.

### 3.1.1.2. IGNORE DIGITSEPARATOR

This enables comparison between values like 1000 and 1,000. In English language, both are the same. But in German, they are not. Sometimes, it may be necessary to compare such things. Here, this ignore command can come into play. Example:

```
testString := “1,040,823”;

 regExObj1 := new regExHandler(
             \
              42 : EQ
                      {IGNORE DIGITSEPARATOR ENGLISH;}
                      {FENCE 1;}
                    : 10408 : ||{}||;;
             \
           );

         M := regExObj1.match_againstTestString(testString);

assert(M.status = True);
assert(M.results[42] = “1,040,8”);
```

The digit separator here is chosen to correspond with English language. The digit separator is comma (,). The matching algorithm only sees 1,040,82 as the Test String (due to the FENCE command). In it, the commas are ignored, and the match is performed, leading to a success.

Note, that the result contains the substring of the actual Test String, together with the digit separator. IGNORE DIGITSEPARATOR; result also returns the substring as found in the Test String, not the supplied Search String. Like IGNORE CASE; it is also symmetric, i.e. ignores digit separator in both the supplied Search String, as well as the Test String.

The result may not match the English language way of using comma. The correct comma-placement would have been 104,082 - but that would require editing the match, and it is not the task of the RegEx.

Digit separators are Language Sensitive. For this reason, a language needs to be mentioned. The default is English. If a non-standard language is needed that is not predefined, then a path to a file describing the ignorable characters can be supplied using the FILE keyword. Example:

```
testString := “1’040,823”;

 regExObj1 := new regExHandler(
             \
              42 : EQ
                      {IGNORE DIGITSEPARATOR FILE
  /path/to/file/describing/comma/and/apostrophe/as/ignorable/chars;}
                      {FENCE 1;}
                    : 10408 : ||{}||;;
             \
           );

         M := regExObj1.match_againstTestString(testString);

assert(M.status = True);
assert(M.results[42] = “1’040,8”);
```

The exact methods to create such a file is TBD.

#### 3.1.1.3. IGNORE ACCENT

This will ignore text ACCENTS, such as the difference between c and ç. Accents are also Language Sensitive. For this reason, a language needs to be mentioned. The accent symbol **is ignorable in the supplied language**. The default is English. The example here, comparison between c and ç only works in English, because in English, the tail under the letter ç can be ignored. If the supplied language was French, it would fail. Example:

```
testString := ‘‘Françoise’’;

 regEx0bj1 := new regExHandler(
             \
              42 : EQ
                      {IGNORE ACCENT FRENCH;}
                      {}
                    : Francoise : ||{}||;;
             \
           );

         M := regEx0bj1.match_againstTestString(testString);

assert(M.status = False);
```

If a non-standard language is needed that is not predefined, then a path to a file describing the ignorable characters can be supplied using the FILE keyword as before. This is also symmetric as before.

#### 3.1.1.4. IGNORE ADORNMENTS

This is used to compare between the letter a and the Unicode character a with a circle - and considers them equal. There are currently no need to specify a language. This is also symmetric as before.

### 3.1.2. INTERPRET commands

Sometimes it may be necessary to expand a given statement. For such statements the INTERPRET command can be used **TBD**

**Additional commands : TBD**

## 3.2. Additional Commands

The General Commands, such as **OFFSET**, **FENCE**, **RANGE**, and **RETREAT** apply as expected.

**TBD**

# Chapter 4
# ENSEMBLE MATCHES

This is used, when a symbol can be matched from a set of symbols. The Ensemble Match is designated by the type keyword **ES** or **ENSEMBLE**.

## 4.1. Discreet Ensembles

In general it may be the case, that a particular symbol in the Test String is undefined, but it is known that it must be an element of a known set. More specifically, assume that our Test String could be either FooBar or FooBaz. We can say, that the last character in the string will be a member of the set {r,z}. So the Search String should somehow include this information. For example:

```
testString1:= “FooBar”;
testString2:= “FooBaz”;

 regExObj1 := new regExHandler(
                \
                  1 : ES {} {} : (r z) : ||{}||;;
                \
              );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1] = “r”;

         M2:= regExObj1.match_againstTestString(testString2);

assert(M2.status = True);
assert(M2.results[1] = “z”);
```

Here, only the last character is matched. Ideally, we would want FooBa to match first as well, followed by r, or z. But we will discuss such combinations later. Here, r and z together form a set, which is called the **Ensemble**. Each element, r and z, individually are called **Symbols**.

Symbols are supplied separated by whitespace. The Ensemble is enclosed in suitable enclosers, as it will be discussed next. In an Ensemble Match instruction (recall, each instruction has it’s own reference) other matches can’t appear, i.e. 1 : ES {} {} : FooBa(r z) : ||{}||;; is an invalid instruction. The proper syntax will be discussed later.

### 4.1.1. Symbol vs Character

A string is made out of discreet elements, called characters. In an alphabet, each character has it’s own unique symbol, not counting ligatures. Each character is often considered an atomic element in most of the programming languages.

However, here we speak of **Symbols**. A **Symbol** is an element can contain multiple characters. Example:

```
testString1:= ''FooBar'';
testString2:= ''FooBaz'';
testString3:= ''FooBaPQRS'';

 regExObj1 := new regExHandler(
                \
                  1 : ES {} {} : (r z PQRS) : ||{}||;;
                \
              );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1] = ''r'');

         M2:= regExObj1.match_againstTestString(testString2);

assert(M2.status = True);
assert(M2.results[1] = ''z'');

         M3:= regExObj1.match_againstTestString(testString3);

assert(M3.status = True);
assert(M3.results[1] = ''PQRS'');
```

Here, PQRS contains 4 characters. But, it is a singular element in the Ensemble, and therefore, is considered a Symbol.

### 4.1.2. And vs Or

In the previous example, any one Symbol in the Ensemble can match the Test String. The regEx algorithm stops at the first successful match. Therefore the Symbols behave as if there is a **or** relation between them. This **or** relation is represented by the round parenthesis, ().

If instead an **and** relation is desired, then instead curly braces, {}, can be used. This will enable the regEx algorithm to continue search, until every Symbol has been matched. the matches can appear in any order, in the suitable substring (extracted either via the use of **OFFSET**, **FENCE**, etc - or due to the use of regEx instruction combinations). Example:

```
testString1:= ''FooBar'';

 regExObj1 := new regExHandler(
                \
                  1 : ES {IGNORE CASE;} : {b o f R} : ||{}||;;
                \
              );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1] = {''F'', ''o'', ''B'', ''r''});
```

We see that every Symbol in the Ensemble {b o f R} has been matched, ignoring the case. The order is not important. Additionally, this time result returns a tuple or a list (decision left to the implementing host language), with the symbols in the order they are found, and in the exact form (such as preserving the case, accent, adornment etc) they are found in the Test String.

If it is necessary to match the Symbols in the exact order given, then the symbols need to be separated by an ampersand, &. Example:

```
testString1:= ‘‘FooBar’’;

 regExObj1 := new regExHandler(
              \
                1 : ES {IGNORE CASE;} : {b & o & f & R} :
                    ||{}||;;
              \
            );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = False);

 regExObj2 := new regExHandler(
              \
                1 : ES {IGNORE CASE;} : {f & o & b & R} :
                    ||{}||;;
              \
            );

         M2:= regExObj2.match_againstTestString(testString1);

assert(M2.status = True);

assert(M2.results[1] = {‘‘F’’, ‘‘o’’, ‘‘B’’, ‘‘r’’});
```

The first one tries to match the symbols in the order given, which clearly fails. The second one does succeed, because it supplied the symbols in the correct order.

The and/or relations can be nested and mixed liberally as needed. Example:

```
testString1:= ‘‘FooBar’’;

 regExObj2 := new regExHandler(
              \
                1 : ES {IGNORE CASE;} : {f & (o p) & {R b}} :
                    ||{}||;;
              \
            );

         M2:= regExObj2.match_againstTestString(testString1);

assert(M2.status = True);

assert(M2.results[1] = {‘‘F’’, ‘‘o’’, ‘‘B’’, ‘‘r’’});
```

Here, the algorithm has three sub-ensembles in the Ensemble. These are f, (o p), and {R b}, in that order. So the algorithm will (ignoring cases) first try to match f - which succeeds at the first character of FooBar. The remaining substring of Test String is ooBar. Then it tries to match either o or p, which succeeds in the second character, leaving oBar as the remaining substring.

Here, both R and b needs to match in any order in the remaining substring, again ignoring the case - which is done successfully.

Some of these things could also be matched by the placeholder asterisk familiar from PCRE. Despite that, the above functionality is allowed.

## 4.2. Continuous Ensembles

Sometimes, it may be necessary to supply many items in standard lists. For example, we may want to say, any character between b and y, including both. Using the syntax of the previous case, we will have to supply 24 separate symbols. This is error prone. For this reason, it may be better to use different syntax.

Alphabets, digits, etc constitute well defined ensembles. These ensembles, or their parts, can be supplied using continuous Ensemble syntax. Example:

```
testString1:= ‘‘FooBar’’;

 regExObj2 := new regExHandler(
             \
               1 : ES {IGNORE CASE;} : (a - h) :
                   ||{}||;;
             \
           );

         M2:= regExObj2.match_againstTestString(testString1);

assert(M2.status = True);

assert(M2.results[1] = {‘‘F’’});
```

Continuous Ensembles are represented with the hyphen or minus symbol -. This means, the Ensemble will be constructed from the letters of the (English) Alphabet from a to h, in the order they appear. The Alphabet is the natural basis to form such an Ensemble. If an alphabet is used as a basis of an Ensemble, then a language needs to be specified as a type command. The type command is USE LANGUAGE - the default of which is English. If the Language is not predefined, then a path to a file can be specified as before.

Here, they are enclosed in round parenthesis, (). Therefore, any Symbol between a and h will match, and the algorithm will stop after the first match. The algorithm tries to match the characters in order as they appear in the Test String, not in the order the Symbols are supplied in the Ensemble.

Here in this example, the character a is the first Symbol in the Ensemble. It could be matched with 5th character in the Test String. However, the algorithm tries to match the first character of the Test String first, which is F. This will also match as it falls between a and h. This way, the first match becomes F, the regEx instruction succeeds, and it stops.

If it is necessary to match every element in an Ensemble is matched, then again, the curly braces, {}, can be used.

The following typical Ensembles are supported by default.

→ a - z will match any lower case English letter

→ A - Z will match any upper case English letter (this one and the previous one has the same effect if IGNORE CASE; command is used).

→ 0 - 9 will match any roman numeral

→ a - ü will match any lower case German letter

→ A - Ü will match any lower case German letter

## 4.3. Type specific Commands

We already mentioned that USE LANGUAGE is a type specific command for Ensemble Matches. Here we review them.

### 4.3.1. USE Commands

These commands are the main Type specific commands for Ensemble Matches.

#### 4.3.1.1. USE LANGUAGE

The terminal (First and Last) letters of the English alphabet is the same as the Spanish one. So to specify that we want to match any letter in the Spanish alphabet, we need to specify the language. This can be done as:

```
testString1:= ‘‘FooBar’’;

 regExObj2 := new regExHandler(
             \
               1 : ES {IGNORE CASE; USE LANGUAGE SPANISH;} :
                   (a - z) :
                   ||{}||;;
             \
           );

         M2:= regExObj2.match_againstTestString(testString1);

assert(M2.status = True);

assert(M2.results[1] = {‘‘F’’});
```

This will include the letter ñ in the alphabet. The Ensemble becomes ( a b c d e f g h i j k l m n ñ o p q r s t u v w x y z ).

If the language is not predefined, then we can use a path to the file containing the letters of the alphabet. The syntax is: USE LANGUAGE /path/to/file;. Again, the exact syntax for such files is **TBD**.

#### 4.3.1.2. USE NUMBERSYTEM

This is used to select a number system whose numerical digits will be used in an Ensemble. The typically supported values are:

– USE NUMBERSYSTEM DECIMAL : uses the decimal system, so the ensemble consists of the decimal digits

- USE NUMBERSYSTEM BINARY : uses the binary system, so the ensemble consists of 0 and 1
- USE NUMBERSYSTEM HEXADECIMAL : uses the hexadecimal system
- USE NUMBERSYSTEM OCTAL : uses the octal system
- USE NUMBERSYTEM BASE36 : uses the base-36 number system

The symbols used for numerals (digits) are by default English/Latin. Combined with an use language instruction, other numerical digits may be used. For example: USE LANGUAGE Arabic; USE NUMBERSYTEM DECIMAL; will produce an Ensemble of digits 0 to 9 in Arabic language (assuming that the digits are defined).

### 4.3.2. Other Commands

**TBD**.

# Chapter 5
# ECHELON MATCHES

Sometimes, there are no natural ordered continuous Ensembles. For example, if we want to match every lower case letter in the English alphabet, we can write a - z. This covers the entire alphabet, because we know that the first letter of the alphabet in lower case is a, and the last one is z. However, if we want an continuous ensemble of all punctuations, we can't simply write , - ., because no one has defined the comma as the first punctuation, and the period as the last.

For this purpose a better solution is needed. Such solution is implemented in Echelon Matches. The Echelon Match is designated by the type keyword **EC** or **ECHELON**.

## 5.1. The concept of Echelon

An Ensemble can be thought as a shorthand to an ordered set, or its segment. If the order is absent, or isn't enforced, then it becomes an Echelon. Example:

```
testString1:= "Foo,Bar";

 regExObj2 := new regExHandler(
             \
               1 : EC {} {}:
                   !p :
                   ||{}||;;
             \
           );

        M2:= regExObj2.match_againstTestString(testString1);

assert(M2.status = True);

assert(M2.results[1] = ",");
```

Note the !p - this tells the algorithm to match any punctuation. Similarly, we can replace (a - z) with !c, meaning any lower case (English) letter.

### 5.1.1. The Built in Echelons

The following Echelons are supported:

- !p : All punctuations in text, not matching a decimal dot or arithmetical symbols between digits. So if the Test String is $a+b$, the $+$ symbol will be matched, but $2 + 2$ will return no match.
- !c : All lower case characters, if applicable, or all characters, if special commands are used.
- !C: All upper case characters, if applicable.
- !d : All digits.
- !s : All punctuations and symbols, that are neither characters, nor digits, nor whitespace, regardless of whether they are in text, or in a mathematical expression.

- !o : All punctuations and symbols in mathematical context, including decimal dot. So, this time, the Test String is foo.bar, there will be no match, but 2.2 will return the . symbol as match.
- !w: All white space characters.
- !u : Special use - see below.

The Echelon Match also only matches the first possible match.

### 5.1.2. Comparison between Ensemble and Echelons

While it may appear that Ensemble matches are unnecessary (after all a - z is the same as !c), there are specific use cases of both.

The Ensemble syntax allows the selection of a sub-section of an ordered set. For example b - y allows matching any lower case character between b and y in English Language. This is not possible to perform using Echelons, as there can be a very large number of such possible sub-sections, requiring an unusually large number of symbols, each dedicated to a particular sub-section. For such purposes, it is a simpler solution to handle an explicitly specified Ensemble.

One could also a large array of files together with the !u syntax (described later) to match every possible sub-section of an ordered list - but also that is difficult to manage, and the !u syntax can only handle predefined sets of symbols (saved as files). This can be exhausting and error prone, which is best avoided.

Conversely, Ensemble syntax can't handle unordered set. The example of punctuation marks is already mentioned. For this reason, a different syntax is necessary, which is fulfilled by the Echelon Syntax.

## 5.2. Type Specific Commands

There are two main type specific commands applicable here.

### 5.2.1. IGNORE Commands

Also in case of Echelons, sometimes it may be necessary to ignore cases, accents, etc. All Ignore commands mentioned previously can be used here.

### 5.2.2. USE Commands

Punctuations and such can be language dependent. For this purpose, the USE LANGUAGE command may be used as usual. The default language, again, is English.

### 5.2.3. IMPLEMENT Commands

There might be other kinds of writing systems, such as Abugidas, where other kinds of symbols (such as diacritical marks) may be used. To match such symbols, we may need to implement different echelons. The syntax would here call upon the !u Echelon. The !u Echelon is always supplied with an IMPLEMENT type command which takes an additional parameter, where a file path is supplied with the diacritical marks are listed. Example:

```
testString1:= Text with Diacritical Marks;

 regExObj2 := new regExHandler(
             \
               1 : EC {IMPLEMENT /path/to/file;} {}:
                    !u  : ||{}||;;
               \
              );

          M2:= regExObj2.match_againstTestString(testString1);

     assert(M2.status = True);

     assert(M2.results[1] = Match whatever is in the file);
```

Here, the !u matches everything that is specified in the file. **The exact syntax is TBD**. for now, the !u echelon will always be associated with a IMPLEMENT command.

### 5.2.4. Combination

Sometimes, it may be necessary to combine multiple Echelons. The syntax is as follows:

```
     testString1:= ‘‘Foo,Bar’’;

      regExObj2 := new regExHandler(
               \
                1 : EC {} {}:
                    (!p !c):
                    ||{}||;;
               \
              );

          M2:= regExObj2.match_againstTestString(testString1);

     assert(M2.status = True);

     assert(M2.results[1] = ‘‘F’’);
```

As before, due to the presence of round parenthesis, the algorithm here matches either a punctuation or a character. The first match is the first character found, namely F. As usual, the algorithm stops after first successful match. Moreover, curly brace and ampersand syntax for **and** relation and **ordered and** relation can also be used respectively. Using **and** (ordered or otherwise) syntax will cause, as usual, a Tuple or List (as implemented in the host language) to be returned.

# Chapter 6
# EVERYTHING MATCHES

In order to match anything, or nothing at all, the Everything Syntax can be used. The Everything Match is designated by the type keyword **EV** or **EVERYTHING**. Example:

```
testString1:= ‘‘Foo,Bar’’;

 regExObj2 := new regExHandler(
              \
                1 : EV {} {}:
                    * :
                    ||{}||;;
              \
            );

         M2:= regExObj2.match_againstTestString(testString1);

assert(M2.status = True);

assert(M2.results[1] = ‘‘F’’);
```

The asterisk designates that anything (or nothing at all) will be a valid match. In this case, the letter F is the first match, and it returns. More details will be discussed in combination of multiple matches.

If possible, the Everything syntax will match one Symbol. If not, then the result will contain nothing, but the status will still be set as True. Examples follow.

# Chapter 7
# MATCH COMBINATIONS

Often, we will be in a situation, where we will have to test multiple Search Strings against a Test String. For Example, if we want to match an Address, then we might have to match:

- → A house number, with unspecified number of digit
- → Optional whitespaces
- → Optional slash or dash symbol
- → Optional whitespaces
- → Optional further digits
- → Optional whitespaces
- → A comma
- → Optional more whitespace
- → A string of Alphabetic Characters, and whitespace, but no digit.

For simplicity, we first consider all optional points mandatory. Also, imagine that every item is exactly one character long. We will consider the case of possible matches with longer length later.

## 7.1. Search Priority

The items are evaluated in the order they are supplied, regardless of the expected location of the match in the Test String. If there is a need to change the evaluation order, then it is enforeced via the Enquiry Matches, which will be described later.

Because of the combined nature of the set of items, multiple references need to be supplied. Each individual regEx instruction will be associated with an unique reference. These references do not have to occur in a consecutive manner. From there, it follows, that the search algorithm does not utilize the references to establish the Search order.

## 7.2. Building the regEx command

The regEx command will be a combination of all the items, in the specified order. In our case, this becomes:

```
testString1:= “1 / 4 , My strange AddRESs”;

 regExObj1 := new regExHandler(
              \
         {
                1 : EC {} {} : !d : || {} ||;;
              &
                2 : EC {} {} : !w : || {} ||;;
              &
                3 : ES {} {} : (/ -) : || {} ||;;
```

```
                    &
                      4 : EC {} {} : !w : || {} ||;;
                    &
                      5 : EC {} {} : !d : || {} ||;;
                    &
                      6 : EC {} {} : !w : || {} ||;;
                    &
                      7 : EX {} {} : , : || {} ||;;
                    &
                      8 : EC {} {} : !w : || {} ||;;
                    &
                      9 : EC {IGNORE CASE;} {} : (!p !c !w) : || {} ||;;
                }
                    \
                  );

               M1:= regExObj1.match_againstTestString(testString1);

      assert(M1.status = True);
      assert(M1.results[1]  = “1”);
      assert(M1.results[2]  = “ ”);
      assert(M1.results[3]  = “/”);
      assert(M1.results[4]  = “ ”);
      assert(M1.results[5]  = “4”);
      assert(M1.results[6]  = “ ”);
      assert(M1.results[7]  = “,”);
      assert(M1.results[8]  = “ ”);
      assert(M1.results[9]  = “M”);
```

### 7.2.1. Multiple Search Strings in a Specified Order

It is already established, that we need to match a house number, which is given by a number, **and** a whitespace, **and** a slash (or a dash) symbol, **and** a whitespace, **and** a comma ... etc. Moreover, the matches must be performed in the specified order.

We have a total of 9 items to match, connected by the **and** relation. Each item corresponds to its own regEx command, with its associated reference. In other words, the complete regEx handler consists of 9 commands. Taking inspiration from Ensemble Matches, we wrap the entire regEx handler in curly braces {} - which implies that all items must be matched.

In order to establish that we need to follow the exact order supplied, we place the ampersand ( & ) symbol after the double semicolon ( ;; ) terminator at the end of each regEx command, besides for the last. In other word, between each pair of consecutive regEx commands, after the double semicolon of the former and before the reference number of the later, an ampersand symbol in inserted.

This will only succeed, it the items (patterns or Search Strings) appear in the Test String exactly in the order specified. Morever, each item must appear immediately after the previous one (unless modified by the OFFSET command). Example:

```
      testString1:= “1 / 4 HELLO , My strange AddRESs”;

       regExObj1 := new regExHandler(
                    \
                {
                      1 : EC {} {} : !d : || {} ||;;
                    &
                      2 : EC {} {} : !w : || {} ||;;
                    &
                      3 : ES {} {} : (\/ -) : || {} ||;;
                    &
                      4 : EC {} {} : !w : || {} ||;;
                    &
                      5 : EC {} {} : !d : || {} ||;;
                    &
                      6 : EC {} {} : !w : || {} ||;;
                    &
                      7 : EX {} {} : , : || {} ||;;
                    &
                      8 : EC {} {} : !w : || {} ||;;
                    &
                      9 : EC {IGNORE CASE;} {} : (!p !c !w) : || {} ||;;
                }
                    \
                  );

               M1:= regExObj1.match_againstTestString(testString1);

      assert(M1.status = False);
```

This is called an **ORDERED AND** combination. Here, we see that in the supplied Test String, after the final digit ( 4 ), before the comma, the substring HELLO appears. However, the regEx handler expects in item 5 a digit, then a whitespace, and a comma immediately after that in item 7. This is not satisfied by the Test String. Hence the match fails.

It can be imagined, that after each match, the Test String is truncated so that the part where the match appears is removed. In the last successful example, the initial Test String is 1 / 4 , My strange AddRESs. The first item is a digit to be matched, which succeeds with 1, At this point, the Test String will be truncated, such that this 1 (and everything before that, if there was anything before this 1) is removed, leaving / 4 , My strange AddRESs as the new value of the Test String (there is a leading whitespace before the slash symbol). The next item to match is a whitespace. This whitespace must be matched with an implicit Anchor of 0.

After the next match, the Test String becomes / 4 , My strange AddRESs (no leading whitespace). Now the item to match is a slash - which also must appear at Anchor 0. In general, the anchor of the first item to match can be anywhere, but after the first match, the Anchor of every subsequent item in the truncated Test String must be 0. An overlap between matched item is not possible by design.

### 7.2.2. Matching Every Item without a specified order

Sometimes, it may be impossible to specify the order. This can be the case of a code, where the order may be unimportant, or a chemical formula, where the appearance of symbols may not be dictated. To perform such matches, the ampersands between individual instructions may be omitted. Example:

```
      testString1:= “C6H4CNCOOH”;

       regExObj1 := new regExHandler(
                      \
                  {
                        1 : EX {} {} : COOH : || {} ||;;
                        2 : EX {} {} : CN : || {} ||;;
                  }
                      \
                    );

                M1:= regExObj1.match_againstTestString(testString1);

      assert(M1.status = True);
      assert(M1.results[1] = ‘‘COOH’’);
      assert(M1.results[2] = ‘‘CN’’);
```

This is called an **AND** combination. Here we see that both items are matched, but they do not appear in the Test String in the order as specifed by the regEx handler. In this case, however, the matches must be two distinct substring of the Test String without overlaps. Otherwise, a failure will occur. Example:

```
testString1:= ‘‘C6H4CNO’’;

 regExObj1 := new regExHandler(
                \
          {
                1 : EX {} {} : NO : || {} ||;;
                2 : EX {} {} : CN : || {} ||;;
          }
                \
              );

          M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = False);
```

Here, CN and NO are not distinct. The N is an overlap. Here, the algorithm encounters a failure.

### 7.2.3. Matching Every Item without a specified order with overlaps

In case of an **AND** combination without order, a overlap may be necessary. To do that, we must use the plus (+) symbol between elements. Example:

```
testString1:= ‘‘C6H4CNO’’;

 regExObj1 := new regExHandler(
                \
          {
                1 : EX {} {} : NO : || {} ||;;
              +
                2 : EX {} {} : CN : || {} ||;;
          }
                \
              );

          M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1] = ‘‘NO’’);
assert(M1.results[2] = ‘‘CN’’);
```

Here, CN and NO are not distinct. The N is an overlap. But the algorithm succeeded.

### 7.2.4. Matching Every Item with a specified order with overlaps

In case of an **ORDERED AND** combination without order, a overlap may be necessary. To do that, we must use the plus (*) symbol between elements. Example:

```
testString1:= ‘‘C6H4CNO’’;

 regExObj1 := new regExHandler(
                 \
         {
               1 : EX {} {} : CN : || {} ||;;
             *
               2 : EX {} {} : NO : || {} ||;;
         }
                 \
           );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1] = ‘‘CN’’);
assert(M1.results[2] = ‘‘NO’’);
```

Again, CN and NO are not distinct. The N is an overlap. But the algorithm succeeded. But, if we used the following example, this would have failed:

```
testString1:= ‘‘C6H4CNO’’;

 regExObj1 := new regExHandler(
                 \
         {
               1 : EX {} {} : NO : || {} ||;;
             *
               2 : EX {} {} : CN : || {} ||;;
         }
                 \
           );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = False);
```

### 7.2.5. Matching any of the supplied Search Strings

If instead all the commands of the regEx handler was enclosed in a pair of round parentheses, then only one need to be matched. The algorithm stops after encountering the first successful match. This is called an **OR** combination. Example:

```
testString1:= ‘‘C6H4CNO’’;

 regExObj1 := new regExHandler(
                    \
                (
                      1 : EX {} {} : OH : || {} ||;;
                      2 : EX {} {} : CN : || {} ||;;
                )
                    \
                  );

            M1:= regExObj1.match_againstTestString(testString1);

   assert(M1.status = True);
   assert(M1.results[2] = ‘‘CN’’);
```

Here, CN, corresponding to the second command, is a succesful match. The reference of the command leading to a successful match, namely 2, is reflected in the index of M1.results.

### 7.2.6. Matching as many items as possible

Generally, the regEx algorithmis greedy. That is, unless specified, they only match the least number of items that must be matched. An **OR** combination, for example, stops after the first successful match. Sometimes, we may want to match *as many items as possible*. To perform such an operation, slight adjustment is necessary.

By enclosing the entire list of commands in round parentheses and using the slash (/) symbol between them, we achieve the goal of as many matches as possible. Example:

```
   testString1:= ‘‘C6H4CNNH2OH’’;

    regExObj1 := new regExHandler(
                    \
                (
                      1 : EX {} {OFFSET 4; RANGE 4;} : CN : || {} ||;;
                    /
                      2 : EX {} {OFFSET 2; RANGE 3;} : CH5 : || {} ||;;
                    /
                      3 : EX {} {OFFSET 2; RANGE 3;} : OH : || {} ||;;

                )
                    \
                  );

            M1:= regExObj1.match_againstTestString(testString1);

   assert(M1.status = True);
   assert(M1.results[1] = ‘‘CN’’);
   assert(M1.results[2] = NULL);
   assert(M1.results[3] = ‘‘OH’’);
   assert(M1.matchCount = 2);
```

The status will be set to True if at least 1 item is matched, and false otherwise. The total number of successful match is given by the matchCount variable (here there are 2 successful matches).

This syntax is not needed for ensemble matches, because the effect can be achieved also using ENCORE MATCHES which will be described later.

### 7.2.7. Matching EXACTLY one item

An **OR** combination, as stated, stops after the first successful match. Other items may be a successful match. But, we may want to enforce that *more than one successful match will lead to a failure*. This may be emulated with all possible combinations of one item match and other negative match instructions - which will become very complex, with incrreasing number of items. For this, we consider the **XOR** combination. To perform such an operation, again adjustment is necessary.

By enclosing the entire list of commands in double round parentheses, we achieve the goal of as many matches as possible. Example:

```
testString1:= ‘‘C6H4CNNH2OH’’;

 regExObj1 := new regExHandler(
                 \
             (
                   1 : EX {} {OFFSET 4; RANGE 4;} : CN : || {} ||;;
                 ~
                   2 : EX {} {OFFSET 2; RANGE 3;} : CH5 : || {} ||;;
                 ~
                   3 : EX {} {OFFSET 2; RANGE 3;} : OH : || {} ||;;

             )
                 \
               );

            M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = FALSE);
```

The tilda (˜) indicates the **XOR** combination. The status will be set to True if exactly 1 item is matched, and false otherwise.

The double round parenthesis is not needed for ensemble matches, because the effect can be achieved also using ENCORE MATCHES which will be described later.

## 7.3. Search Space

### 7.3.1. RANGE and OFFSET

The combination of various matches require careful consideration of Offset and Range commands.

#### 7.3.1.1. OFFSET

The offset command applies can apply on the entire combination or any individual item.

**OFFSET with ordered AND combination** If the first item is supplied with an Offset, then the entire combination will be offset accordingly. Example:

```
testString1:= ‘‘C6H4CNOH’’;

 regExObj1 := new regExHandler(
                 \
             {
                   1 : EX {} {OFFSET 4;} : CN : || {} ||;;
                 &
                   2 : EX {} {} : OH : || {} ||;;
```

```
            }
                \
              );

           M1:= regExObj1.match_againstTestString(testString1);

  assert(M1.status = True);
  assert(M1.results[1] = ‘‘CN’’);
  assert(M1.results[2] = ‘‘OH’’);
```

Here, the first item, corresponding to CN, is supplied with an offset of 4. Thus the algorithm only sees a part of the Test String C6H4CNOH starting at offset 4. Specifically, the algorithm only encounters the substring CNOH. At this point, the first match can be found at the beginning of this substring. The second match follows immediately.

After the first match, the initial Test String is truncated. If subsequent items were to be associated with an OFFSET command, then a match corresponding to the instruction can't follow its predecessor immediately. The OFFSET command is computed from the beginning of the remaining string after truncation. Example:

```
  testString1:= ‘‘C6H4CNNH2OH’’;

   regExObj1 := new regExHandler(
                \
            {
                  1 : EX {} {OFFSET 4;} : CN : || {} ||;;
                &
                  2 : EX {} {OFFSET 2;} : OH : || {} ||;;
            }
                \
              );

           M1:= regExObj1.match_againstTestString(testString1);

  assert(M1.status = True);
  assert(M1.results[1] = ‘‘CN’’);
  assert(M1.results[2] = ‘‘OH’’);
```

Here, the first instruction is supplied with an OFFSET 4. Therefore the substring that is visible to the algorithm is: CNNH2OH - this is the de-facto Test String. The first match immediately succeeds. Afterwards, the de-facto Test String is truncated (the matched section is removed) to NH2OH. Now, the next item has an offset of 2. That leads to the algorithm seeing 2OH as the de-facto Test String now. This leads to a successful match.

If OFFSET overflows the remaining length, then it leads to an error condition.

**OFFSET with unordered AND combination** If the first item is supplied with an Offset, then the first instruction will be offset accordingly, as before. Example:

```
  testString1:= ‘‘C6H4CNOH’’;

   regExObj1 := new regExHandler(
                \
            {
                  1 : EX {} {OFFSET 4;} : CN : || {} ||;;
                2 : EX {} {} : OH : || {} ||;;
            }
               \
              );

          M1:= regExObj1.match_againstTestString(testString1);

  assert(M1.status = True);
  assert(M1.results[1] = ‘‘CN’’);
  assert(M1.results[2] = ‘‘OH’’);
```

Here, as before, the first item, corresponding to CN, is supplied with an offset of 4. Thus the algorithm only sees a part of the Test String C6H4CNOH starting at offset 4. Specifically, the algorithm only encounters the substring CNOH. At this point, the first match can be found at the beginning of this substring. The second match follows immediately.

After the first match, the initial Test String is however no truncated in case of an unordered **AND** combination. The OFFSET command here is computed from the beginning of the original Test String. Example:

```
  testString1:= ‘‘C6H4CNNH2OH’’;

   regExObj1 := new regExHandler(
               \
            {
                1 : EX {} {OFFSET 4;} : CN : || {} ||;;
                2 : EX {} {OFFSET 8;} : OH : || {} ||;;
            }
               \
              );

          M1:= regExObj1.match_againstTestString(testString1);

  assert(M1.status = True);
  assert(M1.results[1] = ‘‘CN’’);
  assert(M1.results[2] = ‘‘OH’’);
```

Here, the first instruction is supplied with an OFFSET 4. Therefore the substring that is visible to the algorithm is: CNNH2OH - this is the de-facto Test String. The first match immediately succeeds. Now, the next item has an offset of 8, which will again be computed from the beginning of the original Test String. That leads to the algorithm seeing 2OH as the de-facto Test String now. This leads to a successful match once again.

If OFFSET overflows the remaining length, then it leads to an error condition.

**OFFSET with OR combination** With an **OR** combination, the OFFSET command will behave similar to the case with unordered **AND** combination. The algorithm stops at the first match, and the regEx handler reports a success.

#### 7.3.1.2. RANGE

Similar concerns apply on the RANGE command.

**RANGE with ordered AND combination** RANGE will be computed starting from each individual offset of each individual instruction. Since each offset is guaranteed to start at the beginning of a new, truncated substring, we can rest assured that there is no overlap. Example:

```
testString1:= ‘‘C6H4CNNH2OH’’;

 regExObj1 := new regExHandler(
                \
            {
                  1 : EX {} {OFFSET 4; RANGE 4;} : CN : || {} ||;;
                &
                  2 : EX {} {OFFSET 2; RANGE 3;} : OH : || {} ||;;
            }
                \
              );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1] = ‘‘CN’’);
assert(M1.results[2] = ‘‘OH’’);
```

Here, the first instruction is supplied with an OFFSET 4, and RANGE 4. Therefore the substring that is visible to the algorithm is: CNNH - this is the de-facto Test String. The first match immediately succeeds. Afterwards, the de-facto Test String is truncated (the matched section is removed) to NH2OH. Now, the next item has the generic commands OFFSET 2, and RANGE 3. That leads to the algorithm seeing 2OH as the de-facto Test String now. This leads to a successful match as well.

**RANGE with unordered AND combination** Here, care must be taken such that a combination of RANGE and OFFSET does not lead to a overlap. In case of an overlap however, the status will be returned as False. The algorithm keeps track of all matches.

### 7.3.2. FENCE and RETREAT

Similar concerns apply for FENCE and RETREAT commands. Because these are computed from the end of the Test String, computation are simpler. Even after truncation, the end of the Test String does not change. Recall that, Right to Left languages aren't supported yet.

### 7.3.3. ANCHOR

**TBD**

## 7.4. Combining AND / OR combinations

As expected, **AND/OR** combinations can be mixed using a pair of round parentheses instead of curly braces, just like the case of Ensemble or Echelon Matches. Example:

```
testString1:= ‘‘C6H4CNNH2OH’’;

 regExObj1 := new regExHandler(
                \
              (
                {
                  1 : EX {} {OFFSET 4; RANGE 4;} : CN : || {} ||;;
                &
                  2 : EX {} {OFFSET 2; RANGE 3;} : OH : || {} ||;;
                }
```

```
                  {
                    3 : EX {} {} : COOH : || {} ||;;
                  }

                  {
                    4 : EX {} {} : H4 : || {} ||;;
                    5 : EX {} {} : C6 : || {} ||;;
                  }
                )
                  \
                );

           M1:= regExObj1.match_againstTestString(testString1);

  assert(M1.status = True);
  assert(M1.results[1] = ‘‘CN’’);
  assert(M1.results[2] = ‘‘OH’’);
```

Here, the entire set of regEx commands are enclosed in a pair of round patentheses. Inside the outer pair of round parentheses, the instructions are arranged in three blocks. Each of these blocks can be considered as an unit. These units are connected to each other by an **OR** relation. There are three blocks, each enclosed by a pair of curly braces.

The first one, as well as the third one will lead to a successful match. However, recall that the algorithm evaluates the items in the order they are specified. This translates to blocks as well. Blocks are evaluated in the order they are specified. In this example, the first block will be evaluated first, leading to a successful match, upon which the algorithm stops.

### 7.4.1. References

There is no hierarchy in this structure. All instructions, regardless of which block or sub-block they are in, exists in the same level. Therefore, each individual instruction must be associated with an unique reference, wherein the block level will be ignored. In the result, only the references of successful matches will appear.

### 7.4.2. Results overview

In the previous example, there are 5 items supplied (ignoring the block structure). However only 2 of them lead to a successful match. For a quick overview of the successful match instructions, the succeededInstructions variable may be used. This variable is a List (or Tuple, depending on the exact implementation and host language) containing the corresponding references. Example:

```
  testString1:= ‘‘C6H4NH2’’;

   regExObj1 := new regExHandler(
                  \
                (
                  {
                    1 : EX {} {OFFSET 4; RANGE 4;} : CN : || {} ||;;
                  &
                    2 : EX {} {OFFSET 2; RANGE 3;} : OH : || {} ||;;
                  }

                  {
                    3 : EX {} {} : COOH : || {} ||;;
                  }
```

```
                 {
                   4 : EX {} {} : H4 : || {} ||;;
                   5 : EX {} {} : C6 : || {} ||;;
                 }
               )
                 \
               );

            M1:= regExObj1.match_againstTestString(testString1);

   assert(M1.status = True);
   assert(M1.succeededInstructions = {4,5});
   assert(M1.results[4] = ‘‘H4’’);
   assert(M1.results[5] = ‘‘C6’’);
```

### 7.4.3. Depth

There is no upper limit of how deep the blocks of **AND/OR** combination can go. It is only limited by the available memory and computation resource.

## 7.5. Nesting of Commands

In ENQUIRY and ENCORE MATCHES, it will be necessary to nest the instructions in tree like structures. In any other cases, a nesting is not yet supported.

Nesting of Commands is a concept very close to the concept of mixing **AND/OR** combination of commands. However, in the former case, in place of the Search String or Pattern (third segment) of an individual instruction (such as an Encore or an Enquiry instruction), further complete instructions may be inserted. But in case of the mixing of **AND/OR** commands, there is no instruction inserted within another one.

The general syntax for nesting is enclosing the segment in double curly braces ( `{{}}` ). For ENQUIRY MATCHES the if-elseif-else blocks can be supplied in consecutive blocks enclosed by double curly braces. For ENCORE matches, only one such block is needed. Example:

```
   testString1:= ‘‘C6H4CNNH2OH’’;

    regExObj1 := new regExHandler(
                 \
             {
                   1 : EX {} {} : CN : || {} ||;;
                   2 : EC {} {} :
                           {{
                     HERE GOES THE INSTRUCTIONS
                     TO EXECUTE
                     ON SUCCESS OF CONDITION EVALUATION
                           }}
                           {{
                     HERE GOES THE ELSE BLOCK
                           }}
                                : || {} ||;;
             }
                 \
               );
```

```
            M1:= regExObj1.match_againstTestString(testString1);

   assert(M1.status = True);
```

Here we see that inside the **EC** command, where the Search String or Pattern needs to be, further, fully formed instructions (commands) are inserted. This does not happen with mixing of **AND/OR** commands. In the later case, each instruction is in the same level as described above.

Inside the nested block, references and instructions can be applied as usual. Details follow. Also there is no limit of depth for nesting of instruction.

# Chapter 8
# ENQUIRY MATCHES

In order to match anything by a condition, the ENQUIRY Syntax can be used. The Enquiry Match is designated by the type keyword **EN** or **ENQUIRY**. Example:

```
testString1:= ‘‘Foo,Bar,Baz,Zoo,Lorem,Ipsum,Init’’;

 regExObj2 := new regExHandler(
              \
                1 : EN {} {}:
                    {{
                        {
                          LOOKFORWARD BAZ;
                          LOOKFORWARD Lorem;
                        }
                      {
                        1 : EX {IGNORE CASE;} {} : BAR : || {} || ;;
                        2 : EX {} {} : Zoo : || {} || ;;
                      }
                    }}
                    {{
                        {
                          LOOKFORWARD Zoo;
                          LOOKFORWARD Qoo;
                        }
                      {
                        1 : EX {IGNORE CASE;} {} : HELLO : || {} || ;;
                      &
                        2 : EX {} {} : WORLD : || {} || ;;
                      }
                    }}
                    {{
                      {
                        1 : EX {IGNORE CASE;} {} : NO : || {} || ;;
                        2 : EX {} {} : Answer : || {} || ;;
                        3 : EX {} {} : FOUND : || {} || ;;
                      }
                    }}
                  : ||{}||;;
              \
            );

         M2:= regExObj2.match_againstTestString(testString1);

assert(M2.status = True);

assert(M2.results[1][1] = ‘‘Bar’’);
assert(M2.results[1][2] = ‘‘Lorem’’);
```

The system appears complicated at start. We investigate it step by step. Immediately following the EN command. As usual, we can supply Type Specific and General commands, which will be investigated later.

## 8.1. IF - ELSEIF - ELSE

In the Search String / Pattern of an Enquiry Match, several blocks enclosed by double curly braces are found. These correspond to if - elseif (optionally multiple instances) - else blocks. The general syntax is:

```
reference : EN {} {}:
                    {{
                        {
                          IF CONDITION
                        }
                      {
                        BLOCK TO EXECUTE
                        WHEN IF CONDITION EVALUATES TO TRUE
                      }
                    }}
                    {{
                        {
                          ELSE CONDITION
                        }
                      {
                        BLOCK TO EXECUTE
                        WHEN ELSE IF CONDITION EVALUATES TO TRUE
                      }
                    }}

                    . . . ADDITIONAL ELSE IF BLOCKS . . .

                    {{
                      {
                        BLOCK TO EXECUTE
                        WHEN ALL IF and ELSE IF CONDITIONS
                        EVALUATES TO TRUE
                        I.E.  ELSE BLOCK
                      }
                    }}
                  : ||{}||;;
```

Generally the dependent block needs to be enclosed a pair of braces {} even if there is a single insstruction. For **OR** or **XOR** combinations they may be enclosed in a pair of round parentheses (). The condition instruction, if a single one, may be kept without braces.

### 8.1.1. IF block

The first of these blocks correspond an IF section.

#### 8.1.1.1. The condition

Inside the IF block, there are two blocks. First of them is the block where the IF conditions go. As usual, we can enclose these conditions with round parentheses and curly braces for **OR** and **AND** combinations as needed. Each Individual combination must be terminatedwith a semicolon. Since the conditions are all evaluated to True or False, the operations are commutative, and the order does not matter. Hence, in here, the ampersand symbol can't be used.

In the first example of this chapter, there are some special commands, such as LOOKFORWARD. These are just examples, which can be emulated in different, more reliable ways. We will discuss them later.

#### 8.1.1.2. The Test Strings / Match Patterns

The next block contains the Test Strings that need to be matched. This by itself is a valid block of items able to perfom a combined match of several instructions. Taken out of the IF they will execute on their own as well. Each of these items can even have a set of attached code with it as well.

These items can be enclosed by round parentheses or curly braces, with or without the ampersand symbol as usual. We will call this a **Dependent Instruction Block**.

#### 8.1.1.3. Priority of Evaluation

The IF block is always the first block. This must contain some condition. Otherwise an error is generated. The IF block conditions are evaluated first. If it succeeds, the associated items are matched. If the control reaches the items associated with the IF block, then egardless of success or failure of these items, other blocks will not be evaluated.

Inside the IF block, inside the Dependent Instruction Block, the Items execute in the order specified.

#### 8.1.1.4. References

References inside the IF block may start from 1 again. Recall that the entire Dependent Instruction Block inside the IF block itself is a valid set capable of executing stand-alone. The references inside it is fully independent of the references of the enclosing IF block or its siblings. This applies at any nesting depth. To accomodate for such nesting, the results have to be an array. Example:

```
testString1:= ‘‘C6H4CNNH2OH’’;

 regExObj1 := new regExHandler(
                     \
          {
                1 : EX {} {} : CN : || {} ||;;
                2 : EN {} {} :
                        {{
                          {
                        LOOKFORWARD OH;
                          }
                          {
                            10 : EX {} {} : C6 || {} || ;;
                            20 : EX {} {} : H4 || {} || ;;
                          }
                        }}
                            : || {} ||;;
          }
                     \
                   );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1] = ‘‘CN’’);
assert(M1.results[2][10] = ‘‘C6’’);
assert(M1.results[2][20] = ‘‘H4’’);
```

The reference 2 refers to the Enquiry Match. Inside the IF block, inside the Dependent Instruction Block, there are the references 10 and 20. These 10 and 20 can be any integer, and are fully independent to the reference of the Enquiry Match, 2, itself. We need to represent that the Matches corresponding to references 10 and 20 are sub-elements of the instruction given by reference 2. Therefore. the item of the results array corresponding to index 2 itself is also an array. Items of this later array are indexed with respective references as given inside the IF block. This way, the results array can be indefinitely nested, and jagged.

### 8.1.2. ELSE-IF Block and ELSE Block

ELSE-IF blocks immediately follow the IF block. ELSE-IF blocks must be supplied along with another set of conditions.

The Last ELSE-IF block may come without a condition. In such case, it will be considered an ELSE block. If there are any other block besides the last one without any conditions, it will lead to an error.

Everything else inside these blocks are the same as an IF block. Blocks are executed in order they are supplied. If there is are two ELSE-IF blocks, and both conditions are likely to be true, then the algorithm stops after first ELSE-IF block (becuase those are the first set of conditions to be met successfully).

### 8.1.3. Conditions

Now we take a look at the conditions and their syntaxes. There are the following types of conditions.

- → TRUE, FALSE (direct declerations)
- → VARIABLE (That is evaluated by the embedded scripting language, that has a truth value of True or False

#### 8.1.3.1. Direct Declarations

It is possible to supply values such as TRUE or T; or FALSE or F, which can be immediately recognized by the Engine. Note that these values are case sensitive. Any other abbreviation such as TR or FL is not accepted. The variables must be explicitely boolean. No type coercion is applicable.

#### 8.1.3.2. Variables

Beryl supports an embedded scripting language. This will be discussed later. If a variable is set by this language can be evaluated to a boolean value of True or False, then the Engine will recognize that value. This variable needs to be prefixed by an # symbol, and variable combinations must be terminated by a semicolon. Example:

```
testString1:= ‘‘C6H4CNNH2OH’’;

 regExObj1 := new regExHandler(
                 \
           {
                 1 : EX {} {} : CN @pos :
                 || {
                         #importantVar = False;
                         if ( #pos == 4) then #importantVar = True;
                         else #importantVar = False;
```

```
                    } ||;;

                2 : EN {} {} :
                        {{
                          {
                            #importantVar;
                          }
                          {
                            10 : EX {} {} : C6 || {} || ;;
                            20 : EX {} {} : H4 || {} || ;;
                          }
                        }}
                             : || {} ||;;
          }
              \
            );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1] = ‘‘CN’’);
assert(M1.results[2][10] = ‘‘C6’’);
assert(M1.results[2][20] = ‘‘H4’’);
```

Here, first, the first instruction matches the pattern CN. The position of the match is stored in the variable (note the @ symbol) pos. Then, the embedded script (encloded by ||{}||) performs some work, and sets a boolean variable #importantvar. This variable can be accessed by the Enquiry Match (note the # prefix, and the semicolon). Thus this variable can serve as a valid condition.

### 8.1.4. Accessing Condition Status

The conditions of IF-ELSEIF-ELSE blocks are all predefined, and associated with variables. Therefore, they are directly accessible via the variable names.

However, in order to detect combination values, inline variable may be defined. Example:

```
testString1:= ‘‘C6H4CNNH2OH’’;

 regExObj1 := new regExHandler(
              \
          {
                1 : EX {} {} : CN : || {} ||;;
                2 : EN {} {} :
                        {{
                  (
                    { #someVar #someOtherVar }  [$first]
                    { #myVar #myOtherVar }      [$second]
                  )
                          {
                            10 : EX {} {} : C6 || {} || ;;
                            20 : EX {} {} : H4 || {} || ;;
                          }
                        }}
                             : || {} ||;;
          }
```

```
                \
              );

          M1:= regExObj1.match_againstTestString(testString1);

 assert(M1.status = True);
 assert(M1.results[1] = ‘‘CN’’);
 assert(M1.results[2][10] = ‘‘C6’’);
 assert(M1.results[2][20] = ‘‘H4’’);
```

Here, the condition is an **OR** combination between two items. Each of them individually are an **AND** combination. These are:

- { #someVar #someOtherVar }
- { #myVar #myOtherVar }

Of course, these items can be calculated using the variables themselves. But it may be interesting to directly get the intermediate values. For such uses, the intermediate value inline variable declarations are used. These are variables declared directly after the item.

These variables are also prefixed with a $, and enclosed in a pair of square brackets, telling the engine that the truth values of the condition needs to be stored here, without the need of an assignment operator. Throughout the program, these variables can be accessed. But in the access time, they must be prefixed with a hash symbol (#) instead of the $ sign.

**FURTHER syntaxes TBD.**

## 8.2. Look-Forward and Look-Backward

Sometimes it may be necessary to match what lies before the current match, and what is ahead of it. Such cases are called look-backward, resp look-forward (or look-ahead) matches.

In our case we can easily emulate them by an ordered sequence of instructions. Example:

```
 testString1:= ‘‘C6H4CNNH2OH’’;

  regExObj1 := new regExHandler(
                \
          {
                  1 : EX {} {} : C6 : || {} ||;;
                &
                  2 : EX {} {} : CN : || {} ||;;
                &
                  3 : EX {} {} : OH : || {} ||;;
          }
                \
              );

          M1:= regExObj1.match_againstTestString(testString1);

 assert(M1.status = True);
 assert(M1.results[1] = ‘‘C6’’);
```

```
assert(M1.results[2] = ‘‘CN’’);
assert(M1.results[3] = ‘‘OH’’);
```

The sequence will *only* be successful, if all three instructions succeed, in the given sequence. Therefore, this structure is capable of emulating look-forward and look-backward behaviors, without explicitely having to invoke Enquiry Matches.

In addition, these instructions can be equipped with ANCHOR, OFFSET or similar commands, to ensure the location of such matches agree with the user's wish.

## 8.3. Type Specific Commands

**TBD**

# Chapter 9
# ENCORE MATCHES

Encore Matches are used to repeat a match a particular number of times. The Encore Match is designated by the type keyword **ER** or **ENCORE**. The syntax is:

```
testString1:= ‘‘C6H4CNNHNHNHNH2OH’’;

 regExObj1 := new regExHandler(
             \
         {
               1 : ER {} {} :
                   {{
                     REPEAT BLOCK

                     DEPENDENT INSTRUCTION BLOCK
                   }}
                 : || {} ||;;
         }
             \
           );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
```

## 9.1. Repeating a Match

In a conditional Enquiry Match, the **Dependent Instruction Block** is a fully valid standalone block of instruction. The same idea is applied here.

The ER Pattern consists of a **Repeat Block**, immediately followed by a **Dependent Instruction Block**. Just like an EN match, they both are each enclosed in double curly braces. The first of these encloser specifies how many times the following match need to be repeated. The **Dependent Instruction Block** in the next encloser specifies the exact match to be performed.

All instructions in the Dependent Instruction Block are executed in order, before repeating. Example:

```
testString1:= ‘‘C6H4CNNHOSNHOS2OH’’;

 regExObj1 := new regExHandler(
             \
         {
               1 : ER {} {} :
                   {{
                     REPEAT 2;
```

```
                      {
                          11 : EX {} {} : NH : || {} ||;;
                        &
                          12 : EX {} {} : OS : || {} ||;;
                      }
                    }}
                  : || {} ||;;
          }
              \
            );

          M1:= regExObj1.match_againstTestString(testString1);

   assert(M1.status = True);
   assert(M1.results[1][1][11] = ‘‘NH’’);
   assert(M1.results[1][1][12] = ‘‘OS’’);
   assert(M1.results[1][2][11] = ‘‘NH’’);
   assert(M1.results[1][2][12] = ‘‘OS’’);
```

Here the Algorithm first matches NH, then immediately OS (Note the & symbol, signifying the specific order). After that, the Test String is truncated as described previously. Then the match repeates for a second time (as specified in the repeat block). If we wanted to match NH 2 times, and then OS, then we would have to assign them to two different Encore Matches. The result is a multidimensional array, which will be discussed later.

Generally the dependent block needs to be enclosed a pair of braces {} even if there is a single insstruction. For **OR** or **XOR** combinations they may be enclosed in a pair of round parentheses (). The repeat instruction, if a single one, may be kept without braces.

## 9.1.1. Repeating Times

The number of times the Dependent Instruction Block repeats is generally specified by the REPEAT command. In the previous example, the we have: REPEAT 2. The REPEAT command must be followed by a positive integer, including 0, with some modifier symbol, optionally ending in any other number, terminated by a semicolon, or a range designator. Negative numbers are not allowed.

### 9.1.1.1. Exact Number of Times

To match the Dependent Instruction Block exactly N times, the integer N is supplied after the repeat command with whitespace in between. A semicolon terminates the statement. The above example shows a method to match the Dependent Instruction Block exactly 2 times.

### 9.1.1.2. At least N Times

The command REPEAT N+ is used to repeat the match at least N number of times. The algorithm will continue as many times as possible. Unlike many other cases, where the engine stops after the least amount of work to fulfil a requirement, here the maximum number of matches are performed. Example:

```
   testString1:= ‘‘C6H4CNNHNHNHNH2OH’’;

    regExObj1 := new regExHandler(
                  \
          {
                1 : ER {} {} :
                    {{
                      REPEAT 2+;
                    {
                      11 : EX {} {} : NH : || {} ||;;
                    }
                  }}
                : || {} ||;;
        }
            \
          );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1][1][11] = ‘‘NH’’);
assert(M1.results[1][2][11] = ‘‘NH’’);
assert(M1.results[1][3][11] = ‘‘NH’’);
assert(M1.results[1][4][11] = ‘‘NH’’);
```

Here we have the pattern NH, which we want to match at least 2 times. But it occurs 4 times, and all of them are found.

#### 9.1.1.3. At most N times

Similar to the above example, the command REPEAT N- can be used to match at most N times. The value of N can't be 0 here. In case N is 1, a special case occurs (negation of Match) that we will discuss later. This operation also makes the algorithm to match as many times as possible.

#### 9.1.1.4. Between M and N times

The command REPEAT M N can be used to match at least M and at most N times. The value of N can't be smaller than M here. This operation also makes the algorithm to match as many times as possible.

#### 9.1.1.5. Specific discreet number of times

Sometimes, we want the allowed number of repeatations be given by a set of discrete values. The command REPEAT (N1 N2 N3 N4) can be used to match either N1 or N2 or N3 or N4 times. These values are given by an **OR** relation, and any number of such values can be passed. Example:

```
testString1:= ‘‘C6H4CNNHNHNHNH2OH’’;

 regExObj1 := new regExHandler(
             \
        {
              1 : ER {} {} :
                  {{
                    REPEAT (2 4 6);
                    {
                      11 : EX {} {} : NH : || {} ||;;
                    }
                  }}
                : || {} ||;;
        }
            \
          );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1][1][11] = ‘‘NH’’);
assert(M1.results[1][2][11] = ‘‘NH’’);
assert(M1.results[1][3][11] = ‘‘NH’’);
assert(M1.results[1][4][11] = ‘‘NH’’);
```

This will also match the maximum number possible. The maximum number, in this case must be 2 or 4 or 6. In this case, the maximum number is 4, and match succeeds. If the number of match was 3, it would have failed because 3 is not in the list. Wecould also have had supplied (2, 3+ 6) as possible values. The first number above 3 that corresponds to the actual number of matches would have led to a success. Note that here, the actual number of matches must correspond to at least one of the items in the list. It is also possible to modify the list with the same **AND**, **XOR**, etc combinations as described above.

#### 9.1.1.6. Unknown Number of times

In normal PCRE, the asterisk symbol (*) is used to denote a repeatation of any number of times, including zero. We can use 0+ for the same effect. This will continue to match as long as possible.

### 9.1.2. Accessing the actual number of repeatation

To get the actual number of repeatation, we use the _ syntax, similar to ENQUIRY Match. Example:

```
testString1:= ‘‘C6H4CNNHNHNHNH2OH’’;

 regExObj1 := new regExHandler(
             \
         {
               1 : ER {} {} :
                   {{
                       REPEAT 2+ [ _count ];
                     {
                       11 : EX {} {} : NH : || {} ||;;
                     }
                   }}
                 : || {} ||;;
         }
             \
           );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1][1][11] = ‘‘NH’’);
assert(M1.results[1][2][11] = ‘‘NH’’);
assert(M1.results[1][3][11] = ‘‘NH’’);
assert(M1.results[1][4][11] = ‘‘NH’’);
```

Here, the repeat command has to run for at least two times. In the actual Test String, the pattern occurs four times, which is the actual (maximum) number of repeats. The engine automatically stores the number 4 in the variable _count. This variable is also later accessed by the hash symbol.

## 9.2. The results Variable

The results variable is relatively complex here. Consider the example again:

```
testString1:= ‘‘C6H4CNNHNHNHNH2OH’’;

 regExObj1 := new regExHandler(
             \
         {
               1 : ER {} {} :
                   {{
                     REPEAT 2+ [ _count ];
                     {
                       11 : EX {} {} : NH : || {} ||;;
                     }
                   }}
                 : || {} ||;;
         }
             \
           );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1][1][11] = ‘‘NH’’);
assert(M1.results[1][2][11] = ‘‘NH’’);
assert(M1.results[1][3][11] = ‘‘NH’’);
assert(M1.results[1][4][11] = ‘‘NH’’);
```

The Encore Match has the reference 1. Therefore, the item in the results variable corresponding to the Encore Match has an index of 1.

This item itself is an array, which has as many items as the number of successful matches. Here, it has 4 items.

Each of these 4 items will also be an array. Each of these arrays will then contain every instruction corresponding to the Dependent Instruction Block. Here, we have only one item in the Dependent Instruction Block, given by the reference 11. For additional nesting, the results variable will become more nested and jagged.

## 9.3. Negating a Match

Sometimes, we want that the match is expressed as *anything but the specified pattern*. In those cases, a Negative Match is needed. Negative matches are found by matching a pattern exactly zero times. Example:

```
testString1:= ‘‘C6H4CNNH2OH’’;

 regExObj1 := new regExHandler(
             \
         {
               1 : EX {} {} : CN : || {} ||;;
             &
               2 : ER {} {} :
                   {{
                     REPEAT 0;
                     {
                       11 : EX {} {} : OS : || {} ||;;
                     }
                   }}
                 : || {} ||;;
         }
             \
           );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[1] = ‘‘CN’’);
```

Here, we want to match the pattern CN and we want to ensure, that immediately followed by CN, in the remaining Test String (after truncation), there is no OS. This set of commands succeeds. It is also possible to equip the negative match with OFFSET, RANGE, and such similar general commands.

## 9.4. Repeat based on Condition

Sometimes, we want that the match is repeated *while a condition is true or until a condition becomes true*. In those cases, a WHILE, resp. UNTIL Match is needed. These are found by replacing the REPEAT command by WHILE resp UNTIL keywords.

### 9.4.1. ER MATCH with WHILE

WHILE is used to run the **ER Match** as long as a condition is true. Example:

```
 testString1:= ‘‘C6H4CNCNCNCNCNCN’’;

  regExObj1 := new regExHandler(
              \
          {
                1 : EY : NULL : || { #cnt := 0; #cnd := (#cnt<3);} ||;;
              &
                2 : ER {} {} :
                    {{
                      WHILE cnd [ _count ];
                      {
                        11 : EX {} {} : CN : || {
                                                    #cnt := #cnt + 1;
                                                    #cnd := (#cnt<3);
                                              } ||;;
                      }
                    }}
                  : || {} ||;;
          }
              \
            );

          M1:= regExObj1.match_againstTestString(testString1);

 assert(M1.status = True);
 assert(M1.results[2][1][11] = ‘‘CN’’);
 assert(M1.results[2][2][11] = ‘‘CN’’);
```

Here, we want to match the pattern CN as long as the count remains less that three. This set of commands succeeds. This particular example could have been performed by a strict REPEAT till 2, as the value 3 is known in advance. But sometimes it's not - and in such cases, WHILE is useful. The **EY** is an **EMPTY MATCH**, used to execute some code before the actual match begins. This will be discussed later.

### 9.4.2. ER MATCH with UNTIL

UNTIL is the opposite of WHILE - as long as the condition is False. **TBD**.

## 9.5. Prematurely stopping an ER MATCH

An **ER MATCH** is supposed to run as the **REPEAT**, **WHILE**, or **UNTIL** declarations dictate. However, if it is necessary to stop the repeating loop before the stop condition is encountered. For that, a BREAK instruction or a FAIL instruction may be supplied in another **EY MATCH**. Example:

```
testString1:= ‘‘C6H4CNCNCNCNCNCN’’;

 regExObj1 := new regExHandler(
             \
         {
               1 : EY : NULL : || { #cnt := 0; } ||;;
             &
               2 : ER {} {} :
                   {{
                     REPEAT 6;
                     {
                       11 : EX {} {} : CN : || #cnt := #cnt +1; ||;;
                       00 : EY : NULL : || if(#cnt > 2) {BREAK;} || ;;
                     }
                   }}
                 : || {} ||;;
         }
             \
           );

         M1:= regExObj1.match_againstTestString(testString1);

assert(M1.status = True);
assert(M1.results[2][1][11] = ‘‘CN’’);
assert(M1.results[2][2][11] = ‘‘CN’’);
assert(M1.results[2][3][11] = ‘‘CN’’);
```

Although planned to run for 6 times, the loop ends after 3 times. The match just before the BREAK command executes is included in the RESULT.

If we were to use a FAIL instruction instead of a BREAK instruction, the same example would have lead to the status being FALSE.

## 9.6. Type specific and General Commands

Everything in the Dependent Instruction Block as well as the Topmost Encore Matche can be equipped with General commands as expected.

**TYPE Specific commands: TBD.**

# Chapter 10
# scripting and variables

Beryl comes with in-built support for scripting.

## 10.1. Variables

Beryl Variables are all local, associated to its regEx unless specifically declared.

### 10.1.1. Global Namespace

Global Namespaces are declared as follows:

```
testString1:= ‘‘C6H4CNNH2OH’’;
 regExNS1  := new regExNamespace();
 regExObj1 := new regExHandler(
             \
         {
               1 : EX {} {} : CN @pos : || {push #pos regExNS1;} ||;;
             &
               2 : EX {} {} : NH : || {} ||;;

         }
             \
           );
 regExObj2 := new regExHandler(
             \
         {
               0 : EY {} {} : NULL : || { #pos := pull regExNS1 pos;} ||;;
               1 : EX {} {OFFSET #pos;} : OH : || {} ||;;

         }
             \
           );

        M2:= regExObj2.match_againstTestString(testString1);

assert(M2.status = True);
assert(M2.results[1] = ‘‘OH’’);
```

Here, the variable #pos (holding the location of the match in the first instruction of the first regEx handler) will only be available for the instructions within regExHandler1.

But using the predefined push command, the variable is pushed in a global namespace regExNS1. This is seen in the additional code section of the first instruction of regExObj1. Then later, in regExObj2, in an empty statement (given by **EY**), the predefined pull command extracts the variable from the global namespace regExNS1 given by the name pos. Now it is available locally with the same name in regExHandler2, also for the additional code anywhere in regExHandler2. Notice that the second pos is not prefixed by the hash symbol, as it represents a variable name. If the command was #pos = pull regExNS1 #posold;, then the system would use the value inside the variable posold as the variable name to look for, which may lead to an error, if not used carefully.

### 10.1.2. Empty instructions

Sometimes, we need to run some code in a regEx handler before any match. Such cases are covered by the **EMPTY** or **EY** Matches. Herein, the pattern is always NULL. No matches are performed, only the additional code is executed.

### 10.1.3. General Syntax

All variable must be prefixed with the hash (#) symbol. Variables are assigned by the := operator, also called the “walrus operator” in Python language.

Variable names must begin with a non-numeric character. Variable names can’t include any whitespace.

Variables are not typed. Type is automatically infered from context. When a variable is reassigned with a new value, its type is also reset. However, type coercion is not implemented.

### 10.1.4. Accessing Variables after regExHandler is executed

Variables may be accessed through the namespace element of the match object, or via the created namespace. Example:

```
testString1:= ‘‘C6H4CNNH2OH’’;
 regExNS1  := new regExNamespace();
 regExObj1 := new regExHandler(
             \
         {
               1 : EX {} {} : CN @pos : || {push #pos regExNS1;} ||;;
             &
               2 : EX {} {} : NH : || {} ||;;

         }
             \
           );
        M1:= regExObj1.match_againstTestString(testString1);
 regExObj2 := new regExHandler(
             \
         {
               0 : EY {} {} : NULL : || { #pos := pull regExNS1 pos;} ||;;
               1 : EX {} {OFFSET #pos;} : OH @pos_in_second_regEx: || {} ||;;

         }
             \
           );

        M2:= regExObj2.match_againstTestString(testString1);

assert(M2.status = True);
assert(M2.results[1] = ‘‘OH’’);
assert(M2.namespace[pos_in_second_regEx].intValue = 10);
assert(regExNS1[pos].intValue = 5;
```

Here, M2.namespace is a dictionary. Using the key pos_in_second_regEx we find the variable that was defined with the same name in regExHandler2. This returns an object. Since we know that it is supposed to be an integer, we can look at the intValue element to find the value we are seeking.

– Strings are found via stringValue;

– Floats and doubles and so on via realValue;
– Booleans via boolValue;
– **others TBD**

Similarly we can query the explicitely created namespace with the appropriate variable name, and look at suitable elements of the returned object. All variables, even the ones created inside the additional code section may be found in this way.

### 10.1.5. Modifying the regEx Result via code

We have already seen, that the usee of FAIL keyword can force the status to be False. In general we can use the SET command to set values in the output. For example:

```
testString1:= ‘‘C6H4CNNH2OH’’;
 regExNS1  := new regExNamespace();
 regExObj1 := new regExHandler(
             \
         {
               1 : EX {} {} : CN @pos : || {push #pos regExNS1;} ||;;
             &
               2 : EX {} {} : NH : || {} ||;;

         }
             \
           );
        M1:= regExObj1.match_againstTestString(testString1);
 regExObj2 := new regExHandler(
             \
         {
               0 : EY {} {} : NULL : || { #pos := pull regExNS1 pos;} ||;;
               1 : EX {} {OFFSET #pos;} : OH @pos_in_second_regEx:
                 || {
                   SET status False;
                   SET foo ‘‘bar’’;
                 } ||;;

         }
             \
           );

        M2:= regExObj2.match_againstTestString(testString1);

assert(M2.status = False);
assert(M2.foo = ‘‘bar’’);
```

Even though the match succeeded here, we have overwritten the status using the SET keyword. Similarly, we can also supply other parameters in the result and corresponding values, as done in SET foo “bar”.

## 10.2. Scripts

Currently Evil-Script by Kagamma (https://github.com/Kagamma/evil-script - and my forked version https://github.com/naturalmechanics/evil-script) is used in my Lazarus implementation of Beryl.