[This document is far from complete but discusses some pattern essentials. Check back for more in the future; ask questions for clarity]

uCalc Patterns

uCalc patterns represent a key element of all current uCalc products. Where and how it is used depends on the product. For the uCalc Search and Transform programs, the user enters uCalc patterns in the search and replace boxes. For uCalc Fast Math Parser, a file named Interface.Bas contains syntax constructs that define what uCalc considers a valid math expression; these constructs are defined using uCalc patterns. The same ucDefineSyntax() function found there can be used to further customize the math parser. For the Console Interpreter, if you browse through *.uc files, you will notice uCalc patterns used in the construction of various programming languages. If you run the interpreter, you can define patterns by starting a line with #Def Syntax: (or do the same but without the "#" for uCalc Graphing Calculator). For the uCalc General Parser you can use uCalc patterns by calling the uCalc() function with uc_DefineSyntax or uc_Define.

A uCalc pattern typically consists of a combination of one or more tokens interspersed with zero or more parameters. A pattern is designed to match the occurrence of a string of text within another string. If a text replacement operation is also being performed, an associated replacement pattern consists of either an empty string or a combination of plain characters (generally with no special token properties) interspersed with 0 or more parameters corresponding with parameters in the original pattern.

Examine the following example which can be pasted into the interpreter (paste the entire block at the same time):

```
#Def Syntax: pi ::= 3.14159
#Def Syntax: A computer program  ::= software
#Def Syntax: %{const} = {value}  ::= public const int {const} =
{value};
#Def Syntax: If {condition} Then {DoIt} ::= if ({condition}) {DoIt};
#Def Syntax: <a {tag:3} {etc}>   ::= <a {tag}>
#Def Syntax: {this} AND {that}   ::= "{this}" is not the same as
"{that}".
#Def Syntax: {same} AND {same}   ::= Two {same}s are better than one
{same}.
#Def Syntax: CountThem({x})      ::= 1
#Def Syntax: CountThem({x}, {y}) ::= ~Eval(1 + CountThem({y}))
```

Each line above tells the parser to search text for a pattern (the part in between `#Def Syntax:` and `::=`) and replace it with what's on the other side of `::=`. After pasting the above block of code, paste in the following block of code to see how text is transformed:

```
#Expand  The value of "pi" is pi.
#Expand  This is a computer program.
#Expand  %MyConstant = 123
#Expand  If x > 1 Then y = z
#Expand  orange AND orange
#Expand  Orange AND Apple
#Expand  <a href="http://www.ucalc.com" class="abc" title="Home page">
#Expand  A total of CountThem(key lime, orange, lemon) fruits were
found.
```

Here is the result:

```
> #Expand  The value of "pi" is pi.
  The value of "pi" is 3.14159.

> #Expand  This is a computer program.
  This is software.

> #Expand  %MyConstant = 123
  public const int MyConstant = 123;

> #Expand  If x > 1 Then y = z
  if (x > 1) y = z;

> #Expand  orange AND orange
  Two oranges are better than one orange.

> #Expand  Orange AND Apple
  "Orange" is not the same as "Apple".

> #Expand  <a href="http://www.ucalc.com" class="abc" title="Home
page">
  <a href="http://www.ucalc.com">

> #Expand  A total of CountThem(key lime, orange, lemon) fruits were
found.
  A total of 3 fruits were found.
```

Parameters

Parameters in a uCalc pattern are the parts that are in between curly braces. Unlike tokens that match an exact predefined sequence of characters, a parameter matches one or more tokens of almost any kind at the given location. The following is a list containing the parameters found in the uCalc patterns from the previous example, each parameter on a separate line:

```
{const}
{value}
{condition}
{DoIt}
{tag:3}
{etc}
{tag}
{this}
{that}
{same}
{x}
{y}
```

By the way rather than manually create the above list in a tedious cut-and-paste session, the example was pasted into [uCalc Search/Replace] and the list was generated by clicking Filter (with Unique set to True). It was filtered with this pattern: "{" {param} "}"

Tokens

Everything else in a uCalc pattern that is not a parameter is a token. Unlike a parameter that can match any token, a token in a uCalc pattern only matches a token in the text that contains the exact same sequence of characters (case sensitivity configuration can determine if upper/lower casing must also match). Some tokens in the example include (one per line):

```
pi
computer
%
=
<
a
>
AND
CountThem
(
,
)
```

What determines the actual grouping of text that forms a token depends on what have been defined using token definitions. The above example is based on the token definitions in the uCalc Interpreter, as found in the file named Tokens.uc. These are similar to the token definitions found in the ucInitialize() function of the Interface.Bas file for uCalc Fast Math Parser. See Token Properties for more details.

Other items from the above example that would be parsed as a token, even if they are not in the main patterns are:

```
software
3.14159
;
~Eval
"http://www.ucalc.com"
```

~Eval is defined in Tokens.uc with a special property that causes the parser to evaluate the expression found within parenthesis, and substitute the result back into the text. Based on definitions from the same Tokens.uc file, items within a pair of double quotes, such as "*http://www.ucalc.com*" are parsed as one token (with literal string properties), so are *3.14159* (numeric), *;* (statement separator), and *software* (alphanumeric). Whitespace is also defined there as a token. Tokens defined as white space have unique properties.

Special characters

Curly braces and square brackets have special meaning in a uCalc pattern. In order to use them as literal characters, enclose them within a pair of quotes, as done in the previously mentioned pattern: "{" {param} "}" , where we wanted to match items found in between curly braces in our text.

Order of definitions (ranking)

The sequence in which similar uCalc patterns are defined may play a role in the way they match text. Ranking of individual patterns is determined either by the order in which they were defined, or by the explicit use of the Rank keyword (See Rank). By default, the more recent a definition, the higher ranking it has. When multiple patterns starting with a similar anchor point match a token in the text, the parser attempts to match the rest of the text with the highest ranking pattern. The next similar pattern is considered only if the current pattern is not an exact match. The parser cycles like this through similar patterns until it finds the first one that matches perfectly.

The starting anchor point is the first token found in a uCalc pattern. Patterns are grouped and ranked against each other if they have the same first token.

Earlier, we dealt with the following two definitions:

```
#Def Syntax: CountThem({x})       ::= 1
#Def Syntax: CountThem({x}, {y}) ::= ~Eval(1 + CountThem({y}))
```

They share the same starting anchor point, which is the token *CountThem*. If an occurrence of CountThem has one or more commas (with two or more arguments) inside the parenthesis, then the

second definition will be chosen as the correct match. If there is no comma, it will still start with the second definition. When that doesn't match, it will try the first definition. In the second definition, the pattern match is replaced with ~Eval(). Unless the PassOnce command is used, when a match is modified it is reparsed in search for more matches. In this situation here, we have a form of recursion. To help us see the steps that are taken, we'll use the #*Steps* command in the interpreter like this:

```
#Steps CountThem(a, b, c, d)
```

which leads to:

```
CountThem(a, b, c, d)
 ~Eval(1 + CountThem(b, c, d))
 ~Eval(1 + ~Eval(1 + CountThem(c, d)))
 ~Eval(1 + ~Eval(1 + ~Eval(1 + CountThem(d))))
 ~Eval(1 + ~Eval(1 + ~Eval(1 + 1)))
 ~Eval(1 + ~Eval(1 + ~Eval(2)))
 ~Eval(1 + ~Eval(3))
 ~Eval(4)
```

Note that patterns that are anchored with *a regular expression* are ranked with token definitions and have a higher priority than patterns that are anchored with a token. Patterns anchored with a regular expression should be used only if there is no other way, since they can be tricky to deal with and can slow the parser down if their ranking is not carefully selected; they should generally be ranked lower than frequently occurring tokens if speed is a concern.

Number of tokens to match

To specify the number of tokens that a parameter should retrieve, place a colon and a numeric value next to the parameter name. Earlier we had the following example:

```
#Def Syntax: <a {tag:3} {etc}>   ::= <a {tag}>
```

This tells the {tag} parameter to retrieve exactly 3 tokens following <a . We don't care how many other tokens come after that. So for the rest we have the {etc} parameter to capture everything else (up to >). Therefore in the following expansion, {tag:3} captures these 3 tokens: href, =, and "http://www.ucalc.com" .

```
#Expand  <a href="http://www.ucalc.com" class="abc" title="Home page">
```

results in:

```
<a href="http://www.ucalc.com">
```

Optional parts

To make part of a pattern optional, enclose it within square brackets. A uCalc pattern may have any number of optional parts, and optional parts can be nested. The following example defines a BASIC-like Dim statement, that can work with or without the optional *As {type}* part.

```
#Def Syntax: Dim {VariableName} [As {type}] ::= name: {VariableName}
~~ DataType: {type}
```

This is similar to making the following two definitions (in that order):

```
#Def Syntax: Dim {VariableName} ::= name: {VariableName} ~~ DataType:
#Def Syntax: Dim {VariableName} As {type} ::= name: {VariableName} ~~
DataType: {type}
```

We can test the following in the interpreter:

```
#Expand Dim Length
#Expand Dim Width As Double
```

Results:

```
> #Expand Dim Length
  name: Length ~~ DataType:

> #Expand Dim Width As Double
  name: Width ~~ DataType: Double
```

Default parameter value

An optional part (same holds true for an alternative part) can have a default value. So taking the above example a step further, we will set the data type to Integer by default, if no data type is specified, by adding an equal sign:

```
#Def Syntax: Dim {VariableName} [As {type=Integer}] ::= name:
{VariableName} ~~ DataType: {type}
```

Using the same test as before we get:

```
> #Expand Dim Length
  name: Length ~~ DataType: Integer

> #Expand Dim Width As Double
  name: Width ~~ DataType: Double
```

Conditional parameter

A parameter can be conditional in the replacement pattern, based on whether the optional parameter was able to match some text. Taking the earlier example a step further, but in a different direction than the previous example, we have:

```
#Def Syntax: Dim {VarName} [As {Type}] ::= name: {VarName} {Type: ~~
DataType: {Type}}
```

In this situation, if no type was specified, then the DataType section will not be featured at all in the replacement:

```
> #Expand Dim Length
 name: Length

> #Expand Dim Width As Double
 name: Width  ~~ DataType: Double
```

Alternative part

[Come back for later update]

Identical parameters

Sometimes you may want a parameter to match only if the match is identical to another parameter. If the same parameter name is featured several times in a pattern, then the pattern will match only if each parameter match is identical. The following is a very practical example derived from the uCalc FMP

auto-generator for the VB header. It finds only function declarations where the function name is identical to the Alias name, in which case the redundant Alias part is removed. Consider the following code:

```
Public Declare Function ucArg Lib "ucFMP315.dll" Alias "ucArg" (ByVal
Expr As Integer, ByVal index As Integer) As Double

Public Declare Function ucParam Lib "ucFMP315.dll" Alias "ucArg"
(ByVal Expr As Integer, ByVal index As Integer) As Double
```

Here is the search/replace pattern:

```
Find: Function {SameName} Lib {DLL} Alias {Q}{SameName}{Q}

Replace: Function {SameName} Lib {DLL}
```

With this search replace operation, the first line is modified, because the function name *ucArg* is the same as the alias. The second line remains the same, because *ucParam* is different from the alias name *ucArg*. So with the search/replace operation, you end up with:

```
Public Declare Function ucArg Lib "ucFMP315.dll" (ByVal Expr As
Integer, ByVal index As Integer) As Double

Public Declare Function ucParam Lib "ucFMP315.dll" Alias "ucArg"
(ByVal Expr As Integer, ByVal index As Integer) As Double
```

Regular expressions

An ordinary parameter retrieves tokens one after the other until it reaches a delimiter. However, it also can instead be defined using a regular expression. A regular expression is denoted either by a colon and pair of quotes following the parameter name, or just a quoted regular expression within curly braces. The following interpreter example defines a binary notation syntax similar to that of the BASIC programming language, and a Test routine followed by required spacing, which is not treated as whitespace:

```
#Def Syntax: {'&b'}{Number:"[01]+"} ::= ~Eval(BaseConvert('{Number}',
2))
#Def Syntax: Test{" +"}.  ::= Testing

#Expand &b101
#Expand Test.
#Expand Test .
```

The results are:

```
> #Expand &b101
  5

> #Expand Test.
  Test.

> #Expand Test .
  Testing
```

Directives

A parameter may include special directives that add flexibility to the way parameter behaves. A parameter may be defined with multiple directives.

Immediate argument substitution (%)

This causes a pattern argument to be expanded immediately so that all substitutions in the argument take place prior to substition for the entire pattern match as a whole. Consider for example the following example, where we first define a syntax construct that changes "Citrus limon" to "Lemon". Now consider definitions for QuoteA and QuoteB that defer only by the presence %.

```
#Def Syntax: Citrus limon ::= Lemon
#Def Syntax: QuoteA({arg})  ::= "{arg}"
#Def Syntax: QuoteB({arg%}) ::= "{arg}"

#Expand QuoteA(Citrus limon)
#Expand QuoteB(Citrus limon)
```

QuoteA will return "Citrus limon", while QuoteB will return "Lemon".

Ignore Statement Separators (+)

Tokens defined as statement separators serve as pattern boundaries. Pattern arguments do not cross over the boundary unless this directive is used. In some programming languages a semicolon is a statement separator while in others like BASIC, the end-of-line or colon serve this purpose. You may define some constructs that deal with individual statements, while you may define others that take a block of code spanning multiple statements. In the following example the + directive indicates that {Statements} can cross over statement separates before reaching End If.

```
If {condition} Then
   {Statements+}
End If
```

Syntax Stop (-)

This indicates that the pattern match stops at, but doesn't include this argument. This means that although text must match the syntax stop parameter, it will will not be included as part of the match. Any substitution that takes place will happen between the start of the match up to but not including the stop parttern. If it's a search, text will be highlighted up to, but not including this part. A syntax stop is valid only at the very end of a pattern.

Skip code execution while parsing (!)

This causes the parser to suppress the evaluation/expansion of inline code while parsing. This refers specifically to source code embeded in text using tokens that have any of the following properties: ucDefineNow, ucEvalDuringParse, ucEvalInsert, ucExpandInsert, ucFileInclude, ucLocalVar, ucStaticVar, ucTempDef. This means that in the interpreter where ~Eval is a token with the ucEvalInsert property, ~Eval(1+1) in the designated section would be skipped over, rather than be replaced with 2 in the text.

Match an expression unit (#)

This causes a pattern to match the maximum number of tokens that form a complete expression. The maximum expression unit in the example is highlighted in yellow:

```
#Def Syntax: Test {MyExpr#} ::= Expression: {MyExpr}, Other:
#Expand Test 3 + 4 * 10 123, 456
```

Result:

```
Expression: 3 + 4 * 10, Other: 123, 456
```

Maximal match (>)

By default a pattern parameter settles for the first occurence of matching text. With maximal matching, it reaches for the last possible match. Consider the following example, where the <end> mark is positioned at the first occurence of a comma, whereas for the maximal parameter, the marker is positioned at the last occurence of a comma:

```
#Def Syntax: Normal   {xyz},  ::= NormalResult= {xyz}<end>
#Def Syntax: Maximal  {xyz>}, ::= MaximalResult= {xyz}<end>

#Expand Normal  a, b, c, d, e
#Expand Maximal a, b, c, d, e
```

Result:

```
> #Expand Normal  a, b, c, d, e
  NormalResult= a<end> b, c, d, e

> #Expand Maximal a, b, c, d, e
  MaximalResult= a, b, c, d<end> e
```

This concept is used in the interpreter Basic.uc file for the PRINT statement, which matches the line up to the last occurence of comma and/or semi-colon.

Ignore patterns (~)

By default when a pattern parameter finds a match, it is later parsed for further patterns. This directive causes the parser to ignore further patterns within a match.

```
#Def Syntax: Bannana ::= fruit
#Def PassOnce ~~ Syntax: Change({text})  ::= Changed: This is a {text}
#Def PassOnce ~~ Syntax: Ignore({text~}) ::= Ignored: This is a {text}
#Expand Change(Bannana)
#Expand Ignore(Bannana)
```

Result:

WhiteSpaceCounts ($)

By default, parameter matches are stripped of leading and trailing whitespace when substitution takes place. This option prevents any whitespace from being stripped.