CrowdStrike Query Language Grammar Subset

CrowdStrike

CrowdStrike Query Language Grammar Subset

CrowdStrike

Abstract

This manual provides the Backus-Naur Form for the CrowdStrike Query Language (CQL).

Build date: 2026-04-29 (44acd1050a)

Table of Contents

Chapter 1. CrowdStrike Query Language Grammar Subset

The following grammar represents a subset of the CrowdStrike Query Language (formerly known as the LogScale Query Language (LQL)). The full grammar is what is implemented by the parser in LogScale, and it contains several quirks that have been elided from this subset.

This guide is intended for programmatically generating LogScale queries — not for parsing them. If you follow the rules in this grammar, the generated queries should be parsed by the LogScale parser.

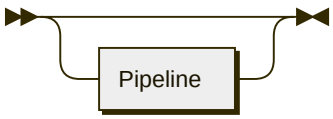
- See Chapter 2, *Grammar Subset* for an overview of the items in this guide.
- See Appendix A, *Quirks* for a discussion of quirks and lessons learned.
- See Appendix B, *Notation* for an overview of the notation used.
- See Appendix C, *Character Table* for a list of the characters supported in the character set ISO/IEC 8859-1.
- See Appendix D, *Appendix D - Reserved Words* for a list of reserved words that should be quoted if they want to be used for filtering.

Chapter 2. Grammar Subset

This subset is not sufficient for parsing LogScale queries. See Appendix A, *Quirks* for more information.

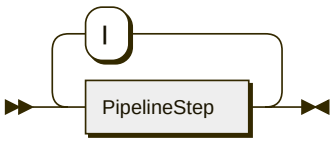
2.1. Query

```
Query ::= Pipeline?
```



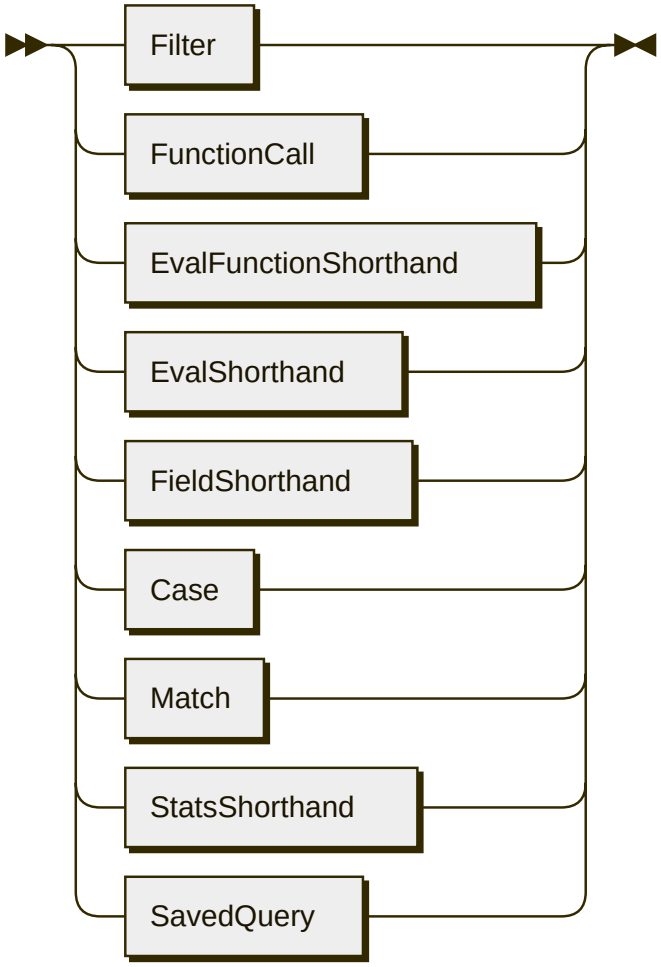
2.2. Pipeline

```
Pipeline ::= PipelineStep ( '|' PipelineStep )*
```



2.3. PipelineStep

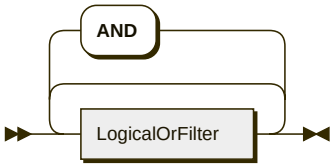
```
PipelineStep ::=
  Filter
  | FunctionCall
  | EvalFunctionShorthand
  | EvalShorthand
  | FieldShorthand
  | Case
  | Match
  | StatsShorthand
  | SavedQuery
```



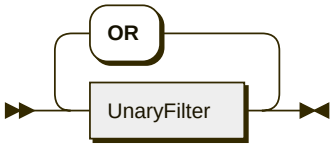
2.4. Filters

```
Filter ::= LogicalAndFilter
```

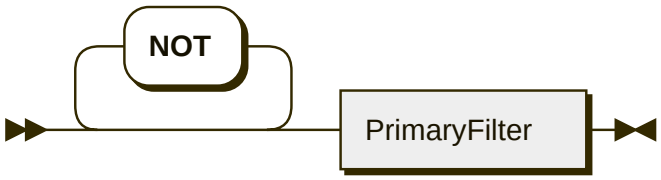

```
LogicalAndFilter ::=
  LogicalOrFilter ('AND'? LogicalOrFilter)*
```



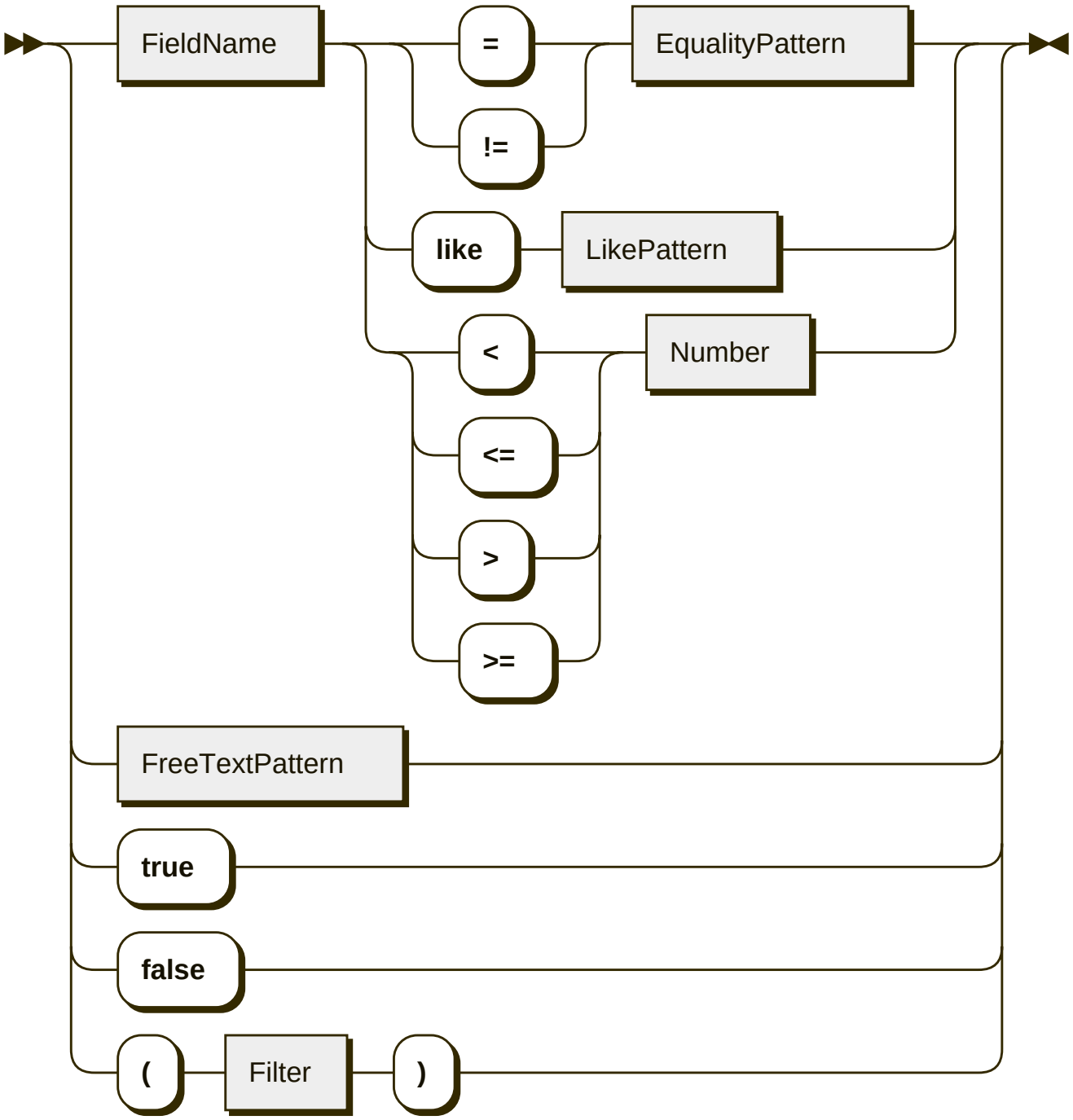

```
LogicalOrFilter ::=
  UnaryFilter ('OR' UnaryFilter)*
```



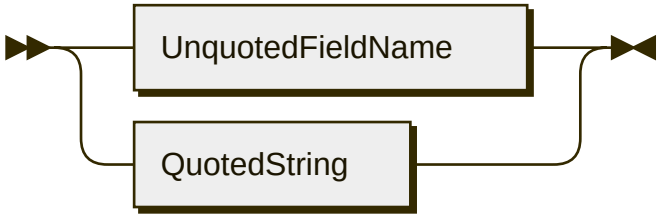
```
UnaryFilter ::=
  'NOT'* PrimaryFilter
```




```
PrimaryFilter ::=
  FieldName '=' EqualityPattern |
  FieldName 'like' LikePattern |
  FieldName '!=' EqualityPattern |
  FieldName '<' Number |
  FieldName '<=' Number |
  FieldName '>' Number |
  FieldName '>=' Number |
  FreeTextPattern |
  'true' |
  'false' |
  '(' Filter ')'
```




```
FieldName ::= UnquotedFieldName | QuotedString
```



A `Filter` (filter expression) can be a step in a pipeline. Filter is a less general kind of expression compared to an expression. Neither is a subset of the other, but `Filter` is particularly quirky.

Implicit `AND` is supported in the `Filter` production so be aware that this:

```
foo < 42 + 3
```

means:

```
(foo < 42) AND "*+*" AND "*3*"
```

`LogicalOrFilter1 LogicalOrFilter2` is shorthand for `LogicalOrFilter1 AND LogicalOrFilter2`.

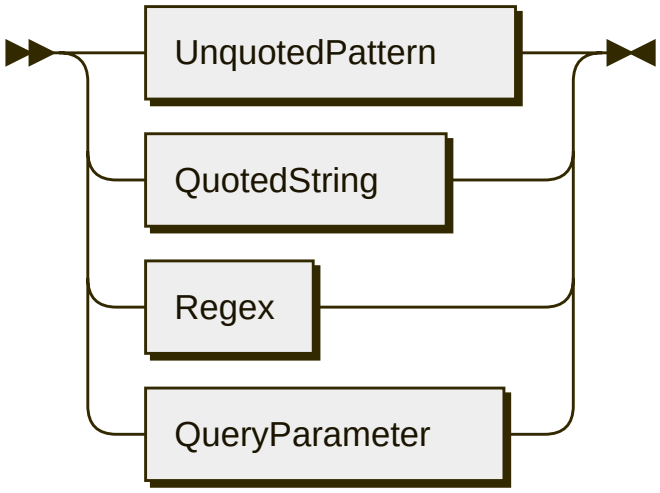
`LogicalOrFilter1 AND LogicalOrFilter2` is shorthand for `LogicalOrFilter1 | LogicalOrFilter2`.

`UnaryFilter1 OR UnaryFilter2` is shorthand for `case { UnaryFilter1; UnaryFilter2 }`.

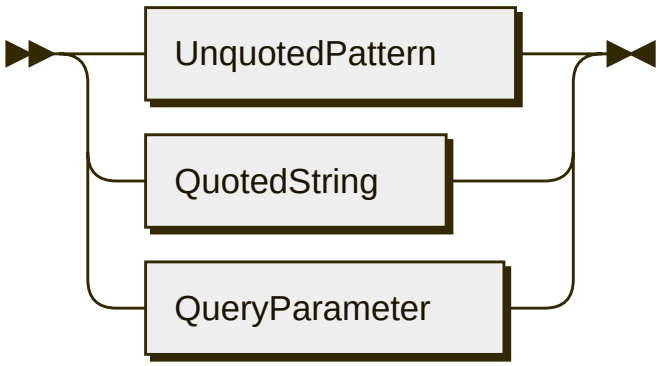
However, `UnaryFilter1` and `UnaryFilter2` cannot be a `Regex` with named groups.

2.5. Patterns

```
FreeTextPattern ::=
  UnquotedPattern | QuotedString | Regex | QueryParameter
```

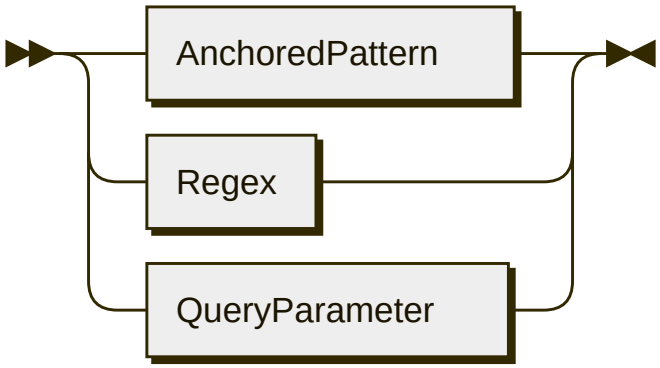



```
LikePattern ::=
  UnquotedPattern | QuotedString | QueryParameter
```

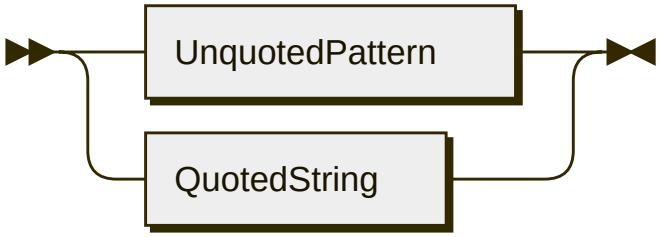



```
EqualityPattern ::=
```

```
  AnchoredPattern | Regex | QueryParameter
```




```
AnchoredPattern ::=
  UnquotedPattern | QuotedString
```



When `UnquotedPattern` or `QuotedString` are used as patterns, the asterisk character (`*`) is a wildcard, meaning that it matches anything. The wildcard cannot be escaped. Regular expressions can be used to search for `*`.

Patterns are sensitive to case. For example, `MyClass` matches `MyClass`, but not `myclass`. Regular expressions can be used to disable case sensitivity in matches using the `i` flag.

2.5.1. Anchored Patterns

We say that a pattern is anchored if it must match the entire string. For example, if the pattern `bar` is anchored, it matches `bar`, but not `foobar`, `barbaz`, or `foobarbaz`.

Regular expressions are anchored only if explicit anchors are used in the regular expression, but unquoted and quoted strings can be anchored or not depending on how they're used. For example, the regular expression `/bar/` matches `bar`, `foobar`, `barbaz`, and `foobarbaz`. The regular expression `/^bar/` matches `bar`, and `barbaz`, but not `foobar`, and `foobarbaz`.

The patterns `AnchoredPattern`, and `EqualityPattern` are always anchored (except for `Regex`, see Section 2.5, “Patterns”).

The patterns `FreeTextPattern`, `LikePattern` are not anchored (except for `Regex`, see Section 2.5, “Patterns”).

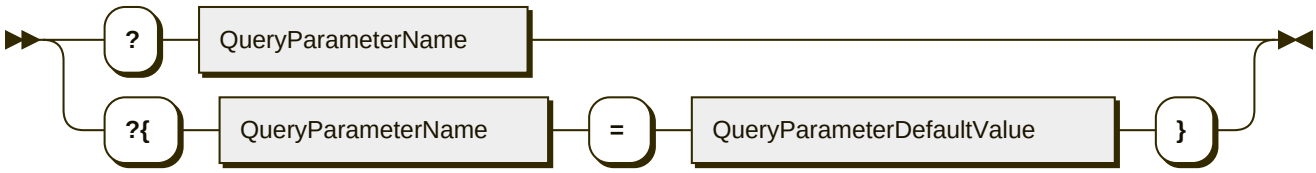
When `UnquotedPattern` or `QuotedString` are not anchored, they are equivalent to having `*` prepended and appended. For example, the pattern `"foo*"` is equivalent to `"*foo**"` or `"*foo*"` when not anchored.

The `FreeTextPattern` `*` is equivalent to `true`, this means that it will match any event — even one without any fields. Also, any number of `*` means the same as `*`, so `""`, `***`, and `"***"` are all equivalent to `true`.

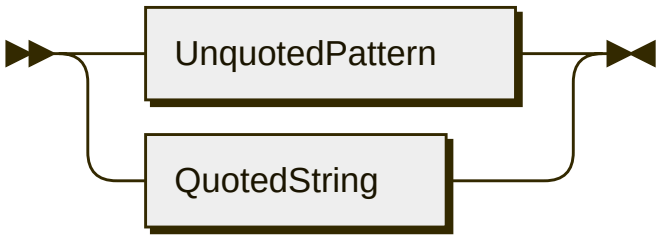
The `EqualityPattern` `*` is not equivalent to `true`, instead it can be used to look for the presence of a field. For example, `my_field = *` matches all events that have a field named my_field, and `my_field != *` matches any event that doesn't. To look for empty (non-empty) fields, use `my_field = ""` (`my_field != ""`). Since `""` is anchored here, it is not equivalent to `*`; however, `"*"`, `"***"`, `**`, all are.

2.6. Query Parameters

```
QueryParameter ::=
  '?' QueryParameterName
  | '?{' QueryParameterName '=' QueryParameterDefaultValue '}'
```




```
QueryParameterName ::= UnquotedPattern
| QuotedString
```




```
QueryParameterDefaultValue ::= UnquotedPattern
| QuotedString
```

A query can be parameterized so that some patterns can be specified later. This is useful, for example, in dashboard widgets: see some Multi-value Parameters.

Arguments to queries are never interpreted as regular expressions. Consider, for example, this query:

```
class = ?my_class
```

This query has one query parameter `my_class`. If the value for `my_class` is set to `/MyClass/`, it is equivalent to:

```
class = "/MyClass/"
```

Not to:

```
class = /MyClass/
```

2.7. Unquoted Strings

```
UnquotedPattern

UnquotedFieldName
```

An `UnquotedPattern` is a non-empty sequence of characters.

An `UnquotedFieldName` is a non-empty sequence of characters.

The exact characters supported is beyond the scope of this document — see Appendix C, *Character Table* for a table of which characters are supported in the character set ISO/IEC 8859-1. LogScale accepts the full Unicode character set, but not all details are included here. We recommend limiting to Unicode identifiers for `UnquotedPattern` and `UnquotedFieldName`.

The reserved character `/` is also supported in `UnquotedPattern`. However, an `UnquotedPattern` cannot start with `/` (one slash) and cannot end with `//` (two slashes). See Section A.1, “Slashes” for more information on slashes.

- `UnquotedPattern` supports the `*` wildcard character, but it does not always mean a wildcard. `UnquotedPattern` does not support the JSON array index syntax.

- `UnquotedFieldName` supports the JSON array index syntax, for example, `foo.bar[42]`, but not the `*` character (wildcard).

2.8. Quoted Strings

```
QuotedString
```

A `QuotedString` is a sequence of characters surrounded by `"`. A `QuotedString` cannot span multiple lines, but `\n` can be used to include the newline character. `\"` can be used to include `"` in the string. `\\` can be used to include the character `\`, and we recommend not including any other sequences of `\` in quoted strings.

2.9. Regular Expressions

```
Regex
```

A Regex is a sequence of characters surrounded by `/`.

A Regex cannot span multiple lines, but `\n` can be used to include the newline character.

`\/` can be used to include `\` in the regular expression. Other characters can also be preceeded by `\` in regular expressions.

For more information, see Regular Expression Syntax Patterns.

By default, regular expressions are not anchored, see Section 2.5.1, “Anchored Patterns”.

A regular expression can be followed by flags. The supported flags are `d`, `m`, and `i`.

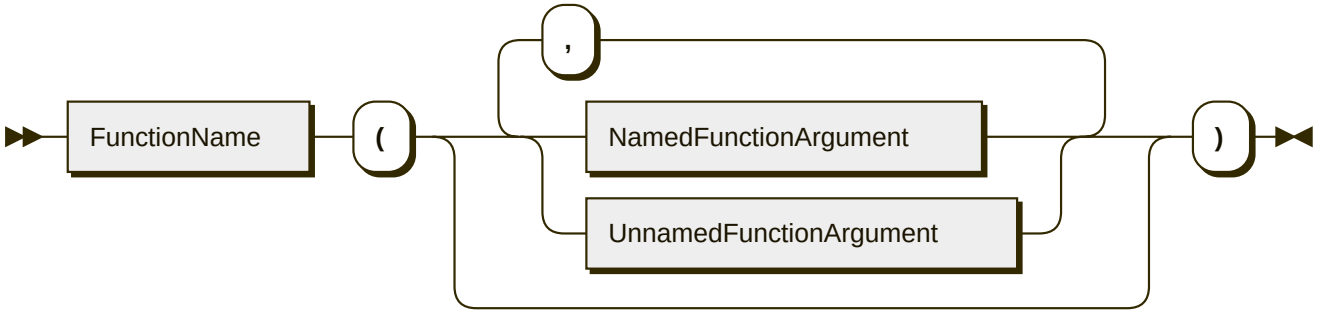
2.10. Numbers

```
Number
```

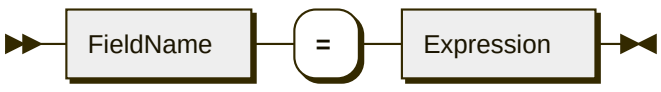
Numbers in LogScale are decimals with optional floating point or scientific notation.

2.11. Function Call

```
FunctionCall ::= FunctionName '(' FunctionArguments? ')'

FunctionArguments ::=
    NamedFunctionArgument (',' FunctionArguments)? |
    UnnamedFunctionArgument (',' FunctionArguments)?
```




```
NamedFunctionArgument ::=
    FieldName '=' Expression
```




```
UnnamedFunctionArgument ::=
    Expression
```

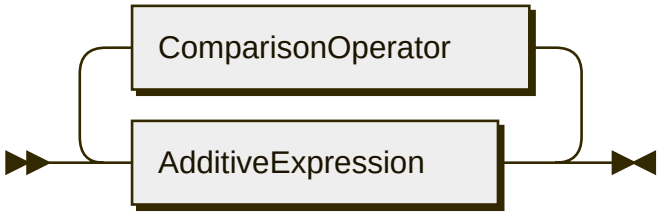
One occurrence of `UnnamedFunctionArgument` at most is supported in `FunctionArguments`.

2.12. Expression

```
Expression ::=
  Expression ComparisonOperator AdditiveExpression |
  AdditiveExpression
```

From v1.192 the expression syntax has been updated to support expression attributes:

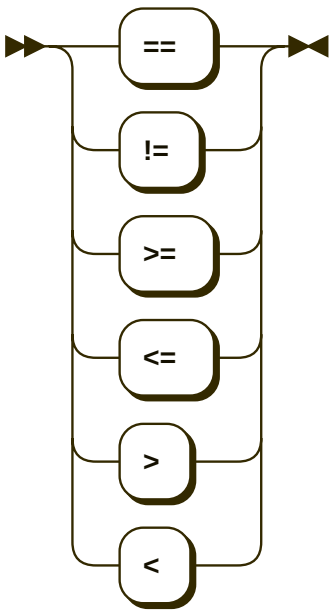
```
Expression ::=
  ComparativeExpression ExpressionAttribute*

ExpressionAttribute ::=
  Identifier ':' ComparativeExpression

ComparativeExpression ::=
  Expression ComparisonOperator AdditiveExpression |
  AdditiveExpression
```



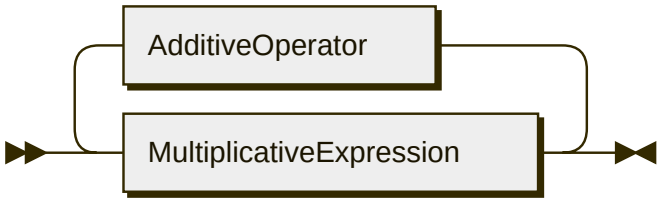

```
ComparisonOperator ::=
  '==' | '!=' | '>=' | '<=' | '>' | '<'
```

From v1.192, a modified format of this syntax is available to support the link operator:

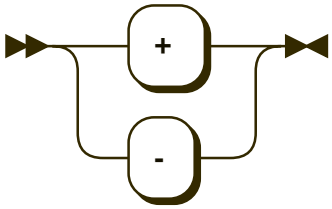
```
ComparisonOperator ::=
  '==' | '!=' | '>=' | '<=' | '>' | '<' | '<=>'
```



```
AdditiveExpression ::=
  AdditiveExpression AdditiveOperator MultiplicativeExpression |
  MultiplicativeExpression
```

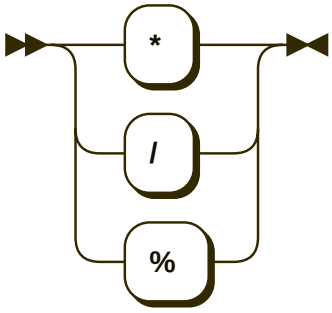



```
AdditiveOperator ::=
  '+' | '-'
```

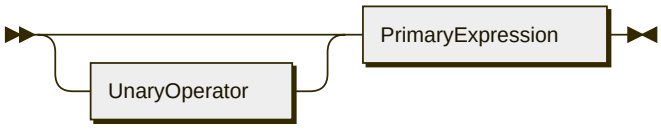



```
MultiplicativeExpression ::=
  MultiplicativeExpression MultiplicativeOperator UnaryExpression |
  UnaryExpression
```

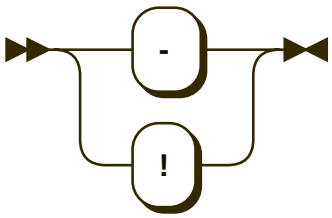

```
MultiplicativeOperator ::=
  '*' | '/' | '%'
```



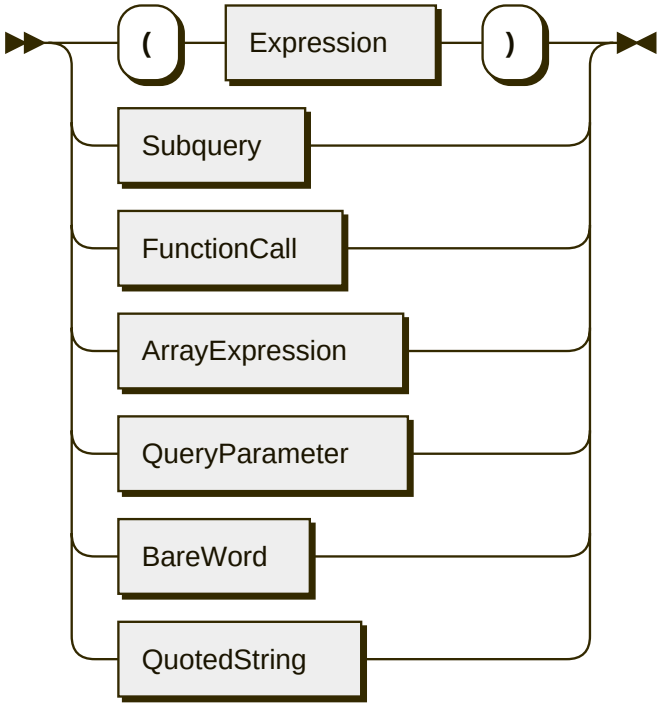

```
UnaryExpression ::=
  UnaryOperator? PrimaryExpression
```



```
UnaryOperator ::=
  '-' | '!'
```




```
PrimaryExpression ::=
  '(' Expression ')' |
  Subquery |
  FunctionCall |
  ArrayExpression |
  QueryParameter |
  BareWord |
  QuotedString
```




```
Subquery ::= '{' Pipeline '}'
```

From v1.192, a modified format of this syntax is available to support an identifier within the subquery clause:

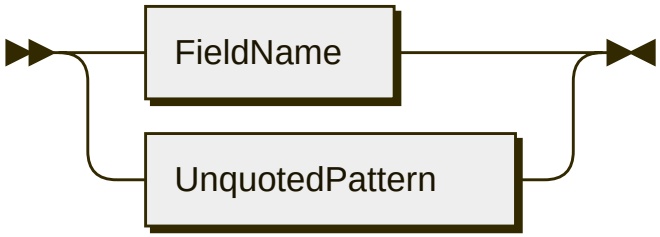
```
Subquery ::= ( Identifier ':' )? '{' Pipeline '}'
```

Where *`Identifier`* is a regular expression matching `/[a-z][a-z0-9_]*/i`.

An `Expression` is similar to an expression from general purpose languages, such as Java or C. It is used as an argument to functions or the right-hand side of `:=` (Section 2.15, “Eval Shorthand”).

2.13. Bare Words

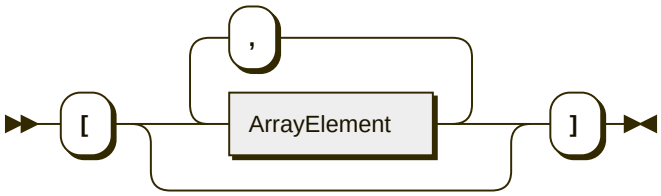
```
BareWord ::=
  FieldName |
  UnquotedPattern
```



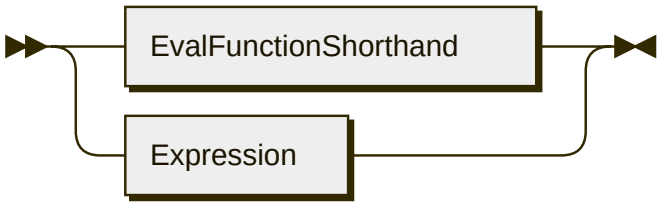
Caveat: `UnquotedPattern` is only used if it starts with a character that cannot start a `FieldName`. For example, `*/*/*https://www.example.com/` is a `BareWord`, but `https:*/*/*https://www.example.com/` isn't.

2.14. Array Expression

```
ArrayExpression ::=
  '[' (ArrayElement (',' ArrayElement)* )? ']'
```

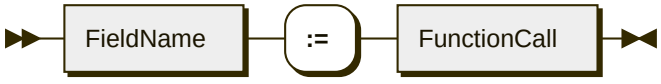



```
ArrayElement ::=
  EvalFunctionShorthand | Expression
```



2.15. Eval Shorthand

```
EvalFunctionShorthand ::=
 FieldName ':=' FunctionCall
```



Shorthand for adding `as=FieldName` to argument list of `FunctionCall`.

```
EvalShorthand ::=
```

```
  FieldName ':=' Expression
```

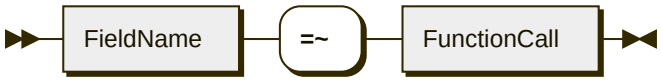
Shorthand for:

```
eval(FieldName = Expression)
```

Note that `eval()` does not support `ArrayExpression` and `Subquery` expressions.

2.16. Field Shorthand

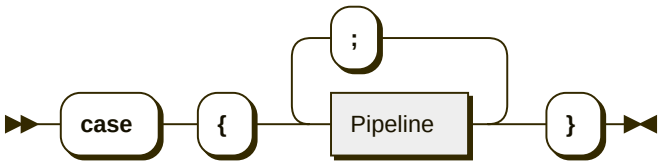
```
FieldShorthand ::= FieldName '=~' FunctionCall
```



Shorthand for adding `field=FieldName` to argument list of `FunctionCall`.

2.17. Case

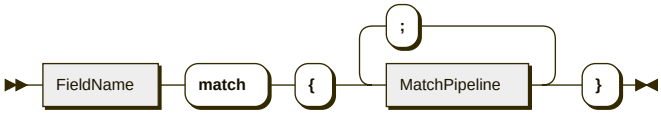
```
Case ::=
  'case' '{' Pipeline (';' Pipeline)* '}'
```



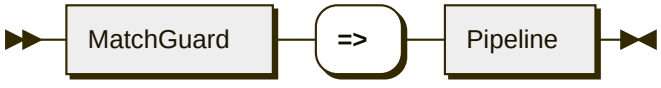
Runs each pipeline in sequence until one matches an event.

2.18. Match

```
Match ::=
  FieldName 'match' '{' MatchPipeline (';' MatchPipeline)* '}'
```

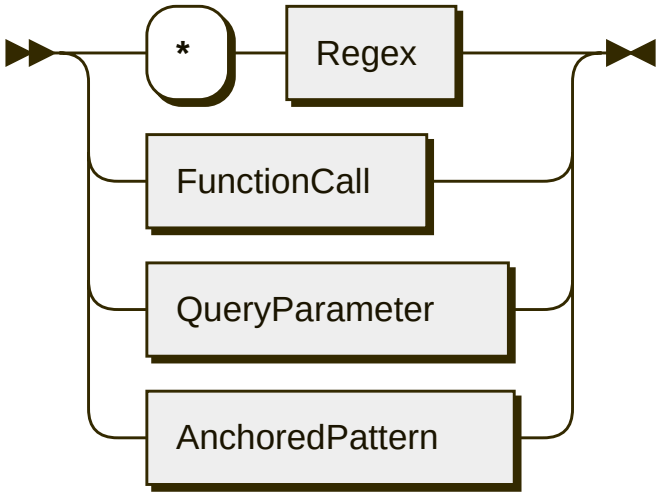



```
MatchPipeline ::=
  MatchGuard => Pipeline
```




```
MatchGuard ::=
  '*'
  Regex |
  FunctionCall |
  QueryParameter |
```

```
  AnchoredPattern
```



`Match` evaluates the first `Pipeline` whose `MatchGuard` matches the current event. `Match` is not a shorthand for `Case`.

A `MatchGuard` is evaluated as follows:

If `MatchGuard` is on the form `'*'`, the current event is matched.

Surprisingly enough, this is equivalent to `*`, not `MyField = *`. In other words, not a field existence test.

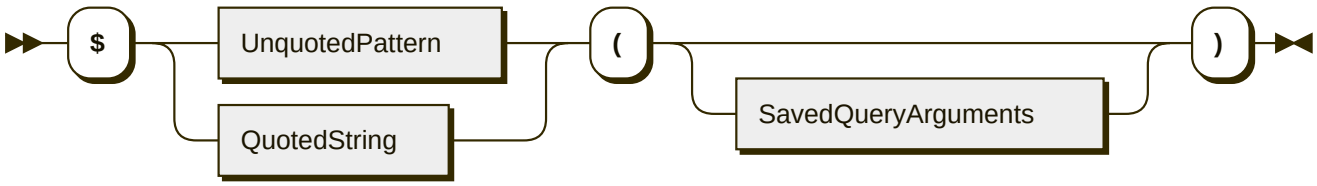
Otherwise, if `MatchGuard` is on the form `Regex`, the current event is matched if this matches: `FieldName = Regex`.

Otherwise, if `MatchGuard` is on the form `FunctionCall`, the current event is matched if this matches: `FieldName =~ FunctionCall`.

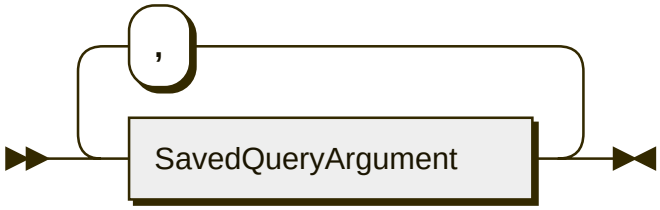
Otherwise, if `MatchGuard` is on the form `AnchoredPattern`, the current event is matched if this matches: `FieldName=~ AnchoredPattern`.

2.19. Saved Query

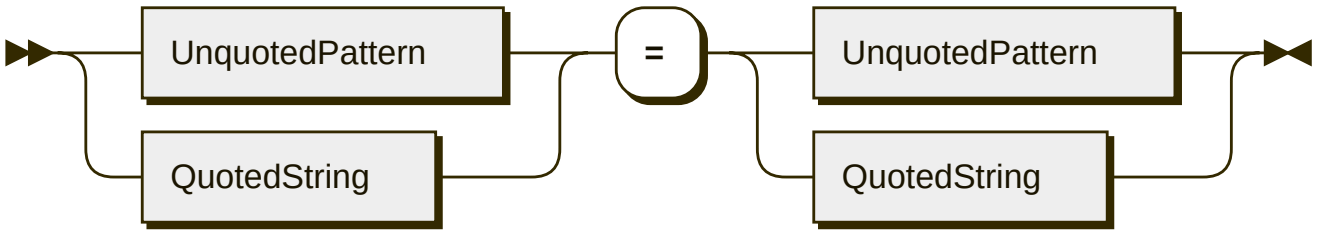
```
SavedQuery ::=
  '$' (UnquotedPattern | QuotedString) '(' SavedQueryArguments? ')'
```




```
SavedQueryArguments ::=
  SavedQueryArgument (',' SavedQueryArgument)*
```




```
SavedQueryArgument ::=
  (UnquotedPattern | QuotedString) '=' (UnquotedPattern | QuotedString)
```



2.20. Stats Shorthand

```
StatsShorthand ::= ArrayExpression
```

`[ e1 , e2 , … , e3 ]` is shorthand for `stats([ e1 , e2 , … , e3 ])`.

Appendix A. Quirks

Below, we elaborate on most of the quirks in the CrowdStrike Query Language in the hopes that this will discourage anyone from attempting to write their own parser.

A.1. Slashes

Slash (`/`) is used for multiple features that makes tokenization hard. These features include comments, regular expression literals, and division. We recommend using LSP to tokenize queries rather than clients attempting to duplicate these quirks. In particular, we do not believe that regular expressions can be used to identify comments or regular expression literals.

Examples:

`https://www.example.com/` is a valid query that is equivalent to searching for `"*https://www.example.com/*"`.

`https://` is a syntax error.

`https: //www.example.com/` is a valid query that contains a comment. It is equivalent to searching for `"*https:*"`,

`/fisk/i` is a valid query that is similar to the query `regex("fisk", flags="i")`.

`a:=m/fisk/i` is a valid query that assigns the field `a` the value of dividing the value of the field `m` by the values of the fields `fisk` and `i`.

`a:=/fisk/i` is a syntax error.

`a:=foo==/fisk/i` is a syntax error.

`foo=m/fisk/i` is a query that is equivalent to searching for `foo="m/fisk/i"`.

A.2. Irregularity of Comparison

A common source of confusion is, that the left-hand side of comparison in the `PrimaryFilter` is always a field name, and the right-hand side never is. In particular, the operators `=` and `!=` are confusing, for example:

```
myField = myOtherField
```

This checks if myField holds the value myOtherField. It does not check if the two fields hold the same value.

A.2.1. Using `test()` to Compare Fields

Fields can be compared using `test()`, for example:

```
test(myField == myOtherField)
```

This checks that the fields contain the same value. But `test()` can also be used to check if a field contains a particular value, for example:

```
test(myField == "myOtherField")
```

This checks if myField holds the value myOtherField, just as myField = myOtherField does.

Unfortunately, this syntax does not support field names with spaces.

A.3. Unconventional Precedence of AND/OR Operators

The precedence of `AND` versus `OR` is reversed compared to general programming languages such as C and Java.

Presumably, this is due to Section A.4, “Implicit AND”.

A.4. Implicit AND

The combination of implicit `AND` between filter expressions and parenthesized filter expressions means that one can write a filter expression that looks like a Section 2.11, “Function Call”. For example:

```
hest fisk(field)
```

is a valid query that is equivalent to:

```
"*hest*" AND "*fisk*" AND "*field*"
```

You could write something like this:

```
ERROR // Search for errors
groupBy(host) // Then grouped by host
```

So function names were turned into reserved words.

A.5. Function Calls and Reserved Words

Function names are reserved words. For example, when the function `test()` was introduced, some previously legal queries became syntax errors.

However, reserved words are not enforced consistently.

Examples:

- `test` is a syntax error.
- `test=fisk` is a valid query.
- `fisk(field)` is a syntax error (`fisk` is not a known function).

For a list of reserved words based on the list of current functions, see Appendix D, *Appendix D - Reserved Words*.

A.6. Alternatives

Making function names reserved words is a source of breaking changes each time a new function is added. An alternative would have been to have a rule that if it looks like a Section 2.11, “Function Call”, it is a function call. In this case, an error like this could be correctly diagnosed:

```
ERROR // Search for errors
groupedBy(host) // Then grouped by host
```

No such function as “groupedBy”.

With this approach, new functions can be added without breaking existing queries.

Regarding slashes (`/`) there's probably little value in allowing them in unquoted strings.

Using `'` or `` ` `` for field names could probably mitigate some of the confusion around the comparison operators. However, to completely eliminate the confusion, a name without any quotes should always be interpreted as field name as is the case for `eval()` and `test()`.

A.7. Recommendations for Generating Queries

Avoid using Section A.4, “Implicit AND”.

Prefer `|` over `AND` to avoid confusion around precedence.

Prefer parenthesis around logical expressions to avoid confusion around precedence.

Avoid unquoted strings. In filter expressions, all unquoted strings can be quoted without changing the meaning. However, in `eval()` and `test()` unquoted strings are interpreted as field names, so care must be taken. One approach is to use:

```
tmp := rename("sus field name")
  | test(host==tmp)
  | drop(tmp).
```

Appendix B. Notation

```
Name ::= ...
```

Defines a grammar rule with name `Name`.

```
Rule1 | Rule2
```

Alternative: either `Rule1` or `Rule2`.

```
Rule*
```

Zero or more occurrences of `Rule`.

```
Rule+
```

One or more occurrences of `Rule`.

```
Rule?
```

Zero or one occurrences of `Rule`.

```
'xyz'
```

An occurrence of `xyz` with proper tokenization. For example, `'case'` does not match `cases`.

Appendix C. Character Table

```
U+0000 (?) - U+0008 (?): ReservedCharacter
U+0009 (?) - U+000A (?): Whitespace
U+000B (?)             : ReservedCharacter
U+000C (?) - U+000D (?): Whitespace
U+000E (?) - U+001F (?): ReservedCharacter
U+0020 (?)             : Whitespace
U+0021 (!) - U+0022 ("): ReservedCharacter
U+0023 (#)             : UnquotedPattern, UnquotedFieldName
U+0024 ($)             : ReservedCharacter
U+0025 (%) - U+0026 (&): UnquotedPattern, UnquotedFieldName
U+0027 (') - U+0029 ()): ReservedCharacter
U+002A (*) - U+002B (+): UnquotedPattern
U+002C (,)             : ReservedCharacter
U+002D (-)             : UnquotedPattern
U+002E (.)             : UnquotedPattern, UnquotedFieldName
U+002F (/)             : ReservedCharacter
U+0030 (0) - U+0039 (9): UnquotedPattern, UnquotedFieldName
U+003A (:)             : UnquotedPattern
U+003B (;) - U+003F (?): ReservedCharacter
U+0040 (@) - U+005A (Z): UnquotedPattern, UnquotedFieldName
U+005B ([)             : ReservedCharacter
U+005C (\)             : UnquotedPattern, UnquotedFieldName
U+005D (])             : ReservedCharacter
U+005E (^) - U+005F (_): UnquotedPattern, UnquotedFieldName
U+0060 (`)             : ReservedCharacter
U+0061 (a) - U+007A (z): UnquotedPattern, UnquotedFieldName
U+007B ({) - U+007D (}): ReservedCharacter
U+007E (~)             : UnquotedPattern, UnquotedFieldName
U+007F (?) - U+00A0 (?): ReservedCharacter
U+00A1 (¡) - U+00AA (ª): UnquotedPattern, UnquotedFieldName
U+00AB («)             : ReservedCharacter
U+00AC (¬)             : UnquotedPattern, UnquotedFieldName
U+00AD (?)             : ReservedCharacter
U+00AE (®) - U+00BA (º): UnquotedPattern, UnquotedFieldName
U+00BB (»)             : ReservedCharacter
U+00BC (¼) - U+00FF (ÿ): UnquotedPattern, UnquotedFieldName
```

Appendix D. Appendix D - Reserved Words

The following words are reserved within LogScale; to use a reserved word as a string within a query, the word must be quoted, for example:

```
"groupBy"
```

accumulate	array:append	array:contains	array:dedup
array:drop	array:eval	array:exists	array:filter
array:intersection	array:length	array:reduceAll	array:reduceColumn
array:reduceRow	array:regex	array:rename	array:sort
array:union	asn	avg	base64Decode
base64Encode	beta:parse	bitfield:extractFlags	bitfield:extractFlagsAsArray
bitfield:extractFlagsAsString	bucket	callFunction	case
cidr	coalesce	collect	communityId
concat	concatArray	copyEvent	correlate
count	counterAsRate	createEvents	crypto:md5
crypto:sha1	crypto:sha256	default	defineTable
drop	dropEvent	duration	end
eval	eventFieldCount	eventInternals	eventSize
explain:asTable	fieldstats	findTimestamp	format
formatDuration	formatTime	geography	geohash
getField	groupBy		

phy:distance			
hash	hashMatch	hashRewrite	head
if	in	ioc:lookup	ipLocation
join	json:prettyPrint	kvParse	length
like	linReg	lower	lowercase
match	matchArray	math:abs	math:arccos
math:arcsin	math:arctan	math:arctan2	math:ceil
math:cos	math:cosh	math:degrees	math:exp
math:exp1m	math:floor	math:log	math:log10
math:log1p	math:log2	math:mod	math:pow
math:radians	math:sin	math:sinh	math:spherical2cartesian
math:sqrt	math:tan	math:tanh	max
min	neighbor	now	objectArray:eval
objectArray:exists	parseCEF	parseCsv	parseFixedWidth
parseHexString	parseInt	parseJson	parseLEEF
parseTimestamp	parseUri	parseUrl	parseXml
partition	percentage	percentile	range
rdns	readFile	regex	rename
replace	reverseDns	round	sample
sankey	select	selectFromMax	selectFromMin
selectLast	selfJoin	selfJoinFilter	series
session	setField	setTimeInterval	shannonEntropy

		ter-val	
sliding-TimeWindow	sliding-Window	sort	split
splitString	string	stats	stdDev
stripAnsiCodes	subnet	sum	table
tail	test	text:contains	text:edit-Distance
text:edit-DistanceAsArray	text:endsWith	text:indexOf	text:positionOf
text:startsWith	text:substring	text:trim	time:dayOfMonth
time:dayOfWeek	time:dayOfWeekName	time:dayOfYear	time:hour
time:millisecond	time:minute	time:month	time:monthName
time:second	time:weekOfYear	time:year	timeChart
tokenHash	top	transpose	unit:convert
upper	urlDecode	urlEncode	wildcard
window	worldMap	writeJson	xml:prettyPrint