Contents

Tremor Language Reference **6**

High Level Overview . . . 7
UNIX philosophy . . . 7
Scripting . . . 7
Querying . . . 7
Deploying . . . 7
Running . . . 9
Modularity . . . 9
Module system . . . 9

Full Grammar **10**

Rule Use . . . 10
TREMOR_PATH . . . 10
Modules . . . 10
Constraints . . . 11
Rule ConfigDirectives . . . 11
Rule ConfigDirective . . . 11
Providing a metrics internal via a config directive . . . 11
Rule ArgsWithEnd . . . 11
Rule DefinitionArgs . . . 12
Rule ArgsClause . . . 12
Rule ArgsExprs . . . 12
Rule ArgsExpr . . . 12
Rule CreationWithEnd . . . 12
Rule CreationWith . . . 13
Rule WithClause . . . 13
Rule WithEndClause . . . 13
Rule WithExprs . . . 13
Rule WithExpr . . . 13
Form . . . 13
Example . . . 14
Rule ModuleBody . . . 14
Rule ModuleFile . . . 14
Rule ModuleStmts . . . 14
Rule ModuleStmt . . . 14
Rule ModularTarget . . . 15
Examples . . . 15
Rule DocComment . . . 16
Rule DocComment_ . . . 16
Rule ModComment . . . 16
Example . . . 16
Rule ModComment_ . . . 17
Rule Deploy . . . 17
Deployment Language Entrypoint . . . 17

Rule DeployStmts . . . 17
Rule DeployStmt . . . 18
Rule DeployFlowStmt . . . 18
Rule ConnectorKind . . . 18
Examples . . . 18
Rule FlowStmts . . . 18
Rule FlowStmts_ . . . 19
Rule CreateKind . . . 19
Rule FlowStmtInner . . . 19
Rule Define . . . 19
Rule Create . . . 19
Rule Connect . . . 20
Rule ConnectFromConnector . . . 20
Rule ConnectFromPipeline . . . 20
Rule ConnectToPipeline . . . 20
Rule ConnectToConnector . . . 20
Rule DefineConnector . . . 20
Rule DefineFlow . . . 21
Rule Query . . . 21
Query Language Entrypoint . . . 21
Rule Stmts . . . 21
Rule Stmt . . . 21
Rule DefineWindow . . . 22
Rule DefineOperator . . . 22
Rule DefineScript . . . 22
Rule DefinePipeline . . . 23
Rule CreateScript . . . 23
Rule CreateOperator . . . 23
Rule CreatePipeline . . . 24
Rule MaybePort . . . 24
Rule StreamPort . . . 24
Rule WindowKind . . . 24
Tumbling . . . 24
Sliding . . . 25
Conditioning . . . 25
Rule WindowClause . . . 25
Rule Windows . . . 25
Rule Windows_ . . . 25
Rule Window . . . 25
Rule WindowDefn . . . 26
Rule WhereClause . . . 26
Rule HavingClause . . . 26
Rule GroupByClause . . . 26
Rule GroupDef . . . 27
Rule GroupDefs . . . 27
Rule GroupDefs_ . . . 27

Rule EmbeddedScriptImut . . . 27
Rule EmbeddedScriptContent . . . 27
Rule Ports . . . 28
Rule OperatorKind . . . 28
Rule EmbeddedScript . . . 28
Rule Pipeline . . . 28
Rule PipelineCreateInner . . . 28
Rule Script . . . 29
    Type system . . . 29
    Asymmetric . . . 29
    Computations . . . 30
    Loops . . . 30
    Expression oriented . . . 30
    Event oriented . . . 30
    Illustrative example . . . 30
Rule TopLevelExprs . . . 31
Rule InnerExprs . . . 31
Rule TopLevelExpr . . . 31
    Example . . . 31
Rule Const . . . 32
    Example . . . 32
Rule Expr . . . 32
Rule SimpleExpr . . . 32
Rule AlwaysImutExpr . . . 33
Rule Recur . . . 33
Rule ExprImut . . . 33
Rule OrExprImut . . . 33
Rule XorExprImut . . . 34
Rule AndExprImut . . . 34
Rule BitOrExprImut . . . 34
Rule BitXorExprImut . . . 34
Rule BitAndExprImut . . . 35
Rule EqExprImut . . . 35
Rule CmpExprImut . . . 35
Rule BitShiftExprImut . . . 35
Rule AddExprImut . . . 36
Rule MulExprImut . . . 36
Rule UnaryExprImut . . . 36
Rule UnarySimpleExprImut . . . 36
Rule PresenceSimplExprImut . . . 37
Rule ComplexExprImut . . . 37
Rule Intrinsic . . . 37
    Example . . . 38
Rule FnDefn . . . 38
Rule FnCases . . . 38
Rule FnCaseDefault . . . 38

Rule FnCase . . . 39
Rule FnCaseClauses . . . 39
Rule FnArgs . . . 39
Rule SimpleExprImut . . . 39
Rule Literal . . . 39
Rule Nil . . . 40
Example . . . 40
Rule Bool . . . 40
Example . . . 40
Rule Int . . . 40
Rule Float . . . 40
Rule StringLiteral . . . 40
Rule StrLitElements . . . 41
Rule StringPart . . . 41
Rule List . . . 41
Rule ListElements . . . 41
Rule ListElements_ . . . 42
Rule Record . . . 42
Rule Field . . . 42
Rule Path . . . 42
Rule ExprPathRoot . . . 43
Rule ExprPath . . . 43
Rule MetaPath . . . 43
Rule AggrPath . . . 44
Rule ArgsPath . . . 44
Rule LocalPath . . . 44
Rule ConstPath . . . 44
Rule StatePath . . . 45
Rule EventPath . . . 45
Rule PathSegments . . . 45
Rule Selector . . . 46
Rule Invoke . . . 46
Rule FunctionName . . . 46
Rule ModPath . . . 46
Rule InvokeArgs . . . 46
Rule InvokeArgs_ . . . 47
Rule Drop . . . 47
Constraints . . . 47
Rule Emit . . . 47
. . . 47
Rule Let . . . 48
Rule Assignment . . . 48
Rule Patch . . . 48
Rule PatchOperations . . . 48
Rule PatchField . . . 49
Rule PatchOperationClause . . . 49

Rule Merge . . . 49
Rule For . . . 49
Rule ForCaseClauses . . . 50
Rule ForCaseClause . . . 50
Rule ForImut . . . 50
Rule ForCaseClausesImut . . . 50
Rule ForCaseClauseImut . . . 50
Record Comprehension . . . 51
Array Comprehension . . . 51
Rule Match . . . 51
Rule Predicates . . . 51
Rule PredicateClause . . . 51
Rule Effectors . . . 51
Rule Block . . . 52
Rule MatchImut . . . 52
Rule PredicatesImut . . . 52
Rule CasePattern . . . 52
Rule PredicateClauseImut . . . 52
Rule EffectorsImut . . . 53
Rule BlockImut . . . 53
Rule WhenClause . . . 53
Rule PredicateFieldPattern . . . 53
Rule TestExpr . . . 54
Rule RecordPattern . . . 54
Rule ArrayPattern . . . 54
Rule TuplePattern . . . 55
Rule OpenTuple . . . 55
Rule TuplePredicatePatterns . . . 55
Rule TuplePredicatePattern . . . 56
Rule ArrayPredicatePattern . . . 56
Rule ArrayPredicatePatterns . . . 56
Rule PatternFields . . . 56
Rule PatternFields_ . . . 56
Rule Fields . . . 57
Rule Fields_ . . . 57
Rule Ident . . . 57
Examples of identifiers . . . 57
Keyword escaping . . . 57
Emoji . . . 57
Rule TestLiteral . . . 58
Extracting JSON embedded within strings . . . 58
Decoding base64 embedded within strings . . . 58
Wrap and Extract . . . 58
Rule BytesLiteral . . . 59
Examples . . . 59
Rule Bytes . . . 60

Example: How do I encode a TCP packet? 60
Rule BytesPart 61
Form 61
Size constraints 61
Encoding Hints 61
Rule Sep 62
Rule BinOp 62
Considerations 62
Rule BinCmpEq 63
Rule BinOr 63
Rule BinXor 63
Rule BinAnd 63
Rule BinBitXor 64
Rule BinBitAnd 64
Rule BinEq 64
Rule BinCmp 65
Rule BinBitShift 65
Rule BinAdd 65
Rule BinMul 66
EBNF Grammar 66

Tremor Language Reference

Tremor contains a number of related domain specific languages that are designed to simplify development and operations of event based production systems.

Tremor provides an expression oriented scripting language that is optimized for transforming nested heirarchic data structures with a rich suite of data transformation operations on nominal (named) record (object), array (list, set) and primitive (string, integral, floating point, boolean) data types.

Tremor provides a statement oriented query like language that embeds the scripting language that is flow oriented. Tremor queries are compiled to directed-acyclic graphs represent streaming transformations based on builtin operations such as `select` queries, or custom user defined scripted operations via the `script` operator.

Tremor provides a statement oriented deployment language that embeds the query and scripting langauges allows complex flows composed of query pipelines, and connectors to external data sources and streams to be interconnected and deployed into the tremor runtime.

High Level Overview

```
graph LR
    D[troy] -->|Embeds| Q(snot)
    Q[trickle] --> |Embeds| S{Let me think}
    S[tremor]
```

UNIX philosophy

Tremor follows UNIX philosophy. The scripting langauge encapsulates the computation and manipulation of events. The query language composes multiple streams of events into event processing graphs. The deployment language connects the outside world to event flow applications in units of deployment called a flow.

Scripting

The simplest useful operation in an event based system is to pass an inbound event in real time from some stream of origin to some target stream preserving the event data.

In the scripting language this is a 1 line program:

```
event
```

The `event` keyword in tremor represents the current event being processed.

Querying

In the query language, this is also a 1 line program:

```
select event from in into out;
```

Visually, this might render as follows:

```
graph LR
    input[in] -->|from| process(event)
    process[event] --> |into| output
    output[out]
```

The event originates at a standard builtin stream called `in` and is distributed to a standard builtin stream called `out`.

Deploying

In the deployment language, this is slightly longer:

```
###
### A simple console echo application
### Given json input line by line on stdin
```

```
### Produces json input line by lin on stdout
### Preserving order of events in distribution order
###

define flow main
flow
  # Define a pipeline with our passthrough logic
  define pipeline passthrough
  pipeline
    select event from in into out;
  end;

  # Define a connector that can read from stdin, write to stdio
  # and expects line delimited json messages
  define connector console from stdio
  with
    codec = "json",
    preprocessors = ["lines"],
    postprocessors = ["lines"],
  end;

  create connector out from console;       # Our output to `stdout`
  create connector in from console;        # Our input from `stdin`
  create pipeline main from passthrough;   # Our application

  connect /connector/in to /pipeline/main;  # Connect stdout to the app
  connect /pipeline/main to /connector/out; # Connect the app to stdout

end;

# The deploy command does all the work
deploy flow main;
```

Although the command that instructs tremor to deploy our 2 instances of the `stdio` connector and our query application pipeline is a single line, the flow `main` is a reusable template. So we can store our flow in a separate file and reuse the definitions.

In fact, we have much more flexibility than this. We could further modularise by separating the definitions of `pipeline`s and `connector`s from their use in `flow` definitions.

This would enable us to have the same logic with different connectivity. Perhaps instead of the console (useful for developing and debugging) we might wish to use kafka connectors. Perhaps our kafka configuration will be different depending on whether we're in a staging, development or production environment. Perhaps we are migrating from a legacy kafka cluster to a high performance

`redpanda` Kafka compatible cluster.

We can compose many different variants and reuse the parts as appropriate.

All of these possible `flow` variants have a similar structure:

```
graph LR
    input[in] -->|connect| pipeline(main)
    pipeline[events] --> |connect| output
    output[out]
```

Running

```
$ tremor server run echo.troy
> tremor version: 0.12
> tremor instance: tremor
> rd_kafka version: 0x000002ff, 1.8.2
> allocator: snmalloc
> Listening at: http://0.0.0.0:9898
> 1
< 1
> {}
< {}
> "snot"
< "snot"
```

Modularity

All tremor DSLs share a common set of compiler and runtime infrastructure.

The module system is itself defined as a tremor-module DSL.

The primary domain specific languages are:

Guide	Description	Extension
[tremor-module	- The tremor module system.	`none`
tremor-deploy	- The tremor deployment language.	troy
tremor-query	- The tremor query language.	trickle
tremor-script	- the tremor scripting language.	script

Module system

The DSLs in tremor share a common module system. The module system allows multiple modular scripts to be loaded via one or many mount points. Each mount point provides a heirarchy of modules where nesting and namespacing is indicated by the relative folder structure, including the file's basename.

Scripts can load other scripts. Queries can load scripts, and other queries. Deployments can load queries, can load scripts and can load other deployments.

```
graph LR
    A[troy] -->|Uses| D(module system)
    B[trickle] --> |Uses| D
    C[tremor] --> |Uses| D
    A[troy] -->|Embeds| B
    B[trickle] --> |Embeds| C
```

Full Grammar

Rule Use

Imports definitions from an external source for use in the current source file.

The contents of a source file form a module.

TREMOR_PATH

The `TREMOR_PATH` environment path variable is a `:` delimited set of paths.

Each path is an absolute or relative path to a directory.

When using relative paths - these are relative to the working directory where the `tremor` executable is executed from.

The tremor standard library MUST be added to the path to be accessible to scripts.

```
rule Use ::=
     'use' ModularTarget
  |  'use' ModularTarget  'as' Ident
  ;
```

Modules

Modules can be scripts. Scripts can store function and constant definitions.

Scripts are stored in `.tremor` files.

Modules can be queries. Queries can store window, pipeline, script and operator definitions.

Scripts are stored in `.trickle` files.

Modules can be deployments. Deployments can store connector, pipeline and flow definitions.

Deployments are stored in `.troy` files.

Conditioning Modules in tremor are resolved via the `TREMOR_PATH` environment variable. The variable can refer to multiple directory paths, each separated by a `:` colon. The relative directory structure and base file name of the source file form the relative module path.

Constraints

It is not recommended to have overlapping or shared directories across the set of paths provided in the tremor path.

It is not recommended to have multiple definitions mapping to the same identifier.

Rule ConfigDirectives

The `ConfigDirectives` rule allows line delimited compiler, interpreter or runtime hints to be specified.

```
rule ConfigDirectives ::=
    ConfigDirective ConfigDirectives
  | ConfigDirective
  ;
```

Rule ConfigDirective

A `ConfigDirective` is a directive to the tremor runtime.

Directives MUST begin on a new line with the `#!config` shebang config token.

```
rule ConfigDirective ::=
     '#!config' WithExpr
  ;
```

Providing a metrics internal via a config directive

```
# Enable metrics with a 10 second interval
#!config metrics_interval_s = 10
```

Rule ArgsWithEnd

The `ArgsWithEnd` rule defines an arguments block with an `end` block.

```
rule ArgsWithEnd ::=
    ArgsClause ?  WithEndClause
  |
  ;
```

Rule DefinitionArgs

The `DefinitionArgs` rule defines an arguments block without an `end` block.

```
rule DefinitionArgs ::=
    ArgsClause ?
  ;
```

Rule ArgsClause

The `ArgsClause` rule marks the beginning of an arguments block.

A valid clause has one or many argument expressions delimited by a ‘,’ comma.

```
rule ArgsClause ::=
     'args' ArgsExprs
  ;
```

Rule ArgsExprs

The `ArgsExpr` rule is a macro rule invocation based on the `Sep` separator macro rule.

An args expression is a comma delimited set of argument expressions.

```
rule ArgsExprs ::=
    Sep!(ArgsExprs, ArgsExpr, ",")
  ;
```

Rule ArgsExpr

```
rule ArgsExpr ::=
    Ident  '=' ExprImut
  | Ident
  ;
```

Rule CreationWithEnd

The `CreationWithEnd` rule defines a `with` block of expressions with a terminal `end` keyword.

```
rule CreationWithEnd ::=
    WithEndClause
  |
  ;
```

Rule CreationWith

The `CreationWit` rule defines an optional `with` block of expressions without a terminal `end` keyword.

```
rule CreationWith ::=
    WithClause
  |
  ;
```

Rule WithClause

The `WithClause` rule defines a `with` block with a `,` comma delimited set of `WithExpr` rules.

```
rule WithClause ::=
     'with' WithExprs
  ;
```

Rule WithEndClause

```
rule WithEndClause ::=
    WithClause  'end'
  ;
```

Rule WithExprs

The `WithExprs` rule defines a `,` comma delimited set of `WithExpr` rules.

```
rule WithExprs ::=
    Sep!(WithExprs, WithExpr, ",")
  ;
```

Rule WithExpr

The `WithExpr` rule defines a name value binding.

```
rule WithExpr ::=
    Ident  '=' ExprImut
  ;
```

Form

```
name = <value>
```

Where:

- `name` is an identifier.

- `<value>` is any valid immutable expression.

Example

```
snot = "badger"
```

Rule ModuleBody

The `ModuleBody` rule defines the structure of a valid module in tremor.

Modules begin with optional module comments.

Modules MUST define at least one statement, but may define many.

Statements are `;` semi-colon delimited.

```
rule ModuleBody ::=
    ModComment ModuleStmts
  ;
```

Rule ModuleFile

The `ModuleFile` rule defines a module in tremor.

A module is a unit of compilation.

```
rule ModuleFile ::=
    ModuleBody  '<end-of-stream>'
  ;
```

Rule ModuleStmts

The `ModuleStmts` rule defines a set of module statements.

Module statements are a `;` semi-colon delimited set of `ModuleStmt` rules

```
rule ModuleStmts ::=
    ModuleStmt  ';' ModuleStmts
  | ModuleStmt  ';' ?
  ;
```

Rule ModuleStmt

The `ModuleStmt` rule defines the statement types that are valid in a tremor module.

```
rule ModuleStmt ::=
    Use
  | Const
```

```
  | FnDefn
  | Intrinsic
  | DefineWindow
  | DefineOperator
  | DefineScript
  | DefinePipeline
  | DefineConnector
  | DefineFlow
  ;
```

Rule ModularTarget

A `ModularTarget` indexes into tremor's module path.

In tremor a `module` is a file on the file system.

A `module` is also a unit of compilation.

A `ModularTarget` is a `::` double-colon delimited set of identifiers.

Leading `::` are not supported in a modular target..

Trailing `::` are not supported in a modular target.

```
rule ModularTarget ::=
    Ident
  | ModPath  '::' Ident
  ;
```

Examples

Loading and using a builtin function

```
# Load the base64 utilities
use std::base64;

# Base64 encode the current `event`.
base64::encode(event)
```

Loading and using a builtin function with an alias

```
# Load the base64 utilities
use std::base64 as snot;

# Base64 encode the current `event`.
snot::encode(event)
```

Rule DocComment

The `DocComment` rule specifies documentation comments in tremor.

Documentation comments are optional.

A documentation comment begins with a `##` double-hash and they are line delimited.

Muliple successive comments are coalesced together to form a complete comment.

The content of a documentation comment is markdown syntax.

```
rule DocComment ::=
    ( DocComment_ ) ?
  ;
```

Rule DocComment_

The `DocComment_` rule is an internal part of the `DocComment` rule

```
rule DocComment_ ::=
     '<doc-comment>'
  | DocComment_  '<doc-comment>'
  ;
```

Rule ModComment

The `ModComment` rule specifies module comments in tremor.

Documentation comments for modules are optional.

A module documentation comment begins with a `###` triple-hash and they are line delimited.

Muliple successive comments are coalesced together to form a complete comment.

The content of a module documentation comment is markdown syntax.

```
rule ModComment ::=
    ( ModComment_ ) ?
  ;
```

Example

Module level comments are used throughput the tremor standard library and used as part of our document generation process.

Here is a modified snippet from the standard library to illustrate

```
### The tremor language standard library it provides the following modules:
###
### * [array](std/array.md) - functions to deal with arrays (`[]`)
### * [base64](std/base64.md) - functions for base64 en and decoding
### * [binary](std/base64.md) - functions to deal with binary data (`<< 1, 2, 3 >>`)
### * [float](std/float.md) - functions to deal with floating point numbers
### * [integer](std/integer.md) - functions to deal with integer numbers
### * [json](std/json.md) - functions to deal with JSON
...
```

Rule ModComment_

The `ModComment_` rule is an internal part of the `ModComment` rule

```
rule ModComment_ ::=
     '<mod-comment>'
  | ModComment_  '<mod-comment>'
  ;
```

Rule Deploy

Deployment Language Entrypoint

This is the top level rule of the tremor deployment language `troy`

```
rule Deploy ::=
    ConfigDirectives ModComment DeployStmts  '<end-of-stream>' ?
  | ModComment DeployStmts  '<end-of-stream>' ?
  ;
```

Rule DeployStmts

The `DeployStmts` rule defines the statements that are legal in a deployment module.

Statements in a deployment modules are ; semi-colon delimited.

There MUST be at least one.

There MAY be more than one.

```
rule DeployStmts ::=
    DeployStmt  ';' DeployStmts
  | DeployStmt  ';' ?
  ;
```

Rule DeployStmt

The `DeployStmt` rule constrains the statements that are legal in a `.troy` deployment module.

Importing modules via the `use` clause is allowed.

Flow definitions and `deploy` commands are allowed.

```
rule DeployStmt ::=
    DefineFlow
  | DeployFlowStmt
  | Use
  ;
```

Rule DeployFlowStmt

```
rule DeployFlowStmt ::=
    DocComment  'deploy'  'flow' Ident  'from' ModularTarget CreationWithEnd
  | DocComment  'deploy'  'flow' Ident CreationWithEnd
  ;
```

Rule ConnectorKind

The `ConnectorKind` rule identifies a builtin connector in tremor.

Connectors in tremor are provided by the runtime and builtin. They can be resolved through an identifier.

Examples

The `http_server` identifies a HTTP server connector.

The `metronome` identifies a periodic metronome.

```
rule ConnectorKind ::=
    Ident
  ;
```

Rule FlowStmts

The `FlowStmts` rule defines a mandatory `;` semi-colon delimited sequence of `FlowStmtInner` rules.

```
rule FlowStmts ::=
    FlowStmts_
  ;
```

Rule FlowStmts_

The `FlowStmts_` rule defines a `;` semi-colon delimited sequence of `FlowStmtInner` rules.

```
rule FlowStmts_ ::=
    Sep!(FlowStmts_, FlowStmtInner, ";")
  ;
```

Rule CreateKind

The `CreateKind` rule encapsulates the artefact types that can be created in the tremor deploymant language.

```
rule CreateKind ::=
     'connector'
  |  'pipeline'
  ;
```

Rule FlowStmtInner

The `FlowStmtInner` rule defines the body of a flow definition.

```
rule FlowStmtInner ::=
    Define
  | Create
  | Connect
  | Use
  ;
```

Rule Define

The `Define` rule allows connectors and pipelines to be specified.

```
rule Define ::=
    DefinePipeline
  | DefineConnector
  ;
```

Rule Create

The `Create` rule creates instances of connectors and pipelines in a flow.

```
rule Create ::=
     'create' CreateKind Ident  'from' ModularTarget CreationWithEnd
  |  'create' CreateKind Ident CreationWithEnd
  ;
```

Rule Connect

The `Connect` rule defines routes between connectors and pipelines running in a flow.

```
rule Connect ::=
     'connect'  '/' ConnectFromConnector  'to'  '/' ConnectToPipeline
  |  'connect'  '/' ConnectFromPipeline  'to'  '/' ConnectToConnector
  |  'connect'  '/' ConnectFromPipeline  'to'  '/' ConnectToPipeline
  ;
```

Rule ConnectFromConnector

The `ConnectFromConnector` rule defines a route from a connector instance.

```
rule ConnectFromConnector ::=
     'connector'  '/' Ident MaybePort
  ;
```

Rule ConnectFromPipeline

The `ConnectFromPipeline` rule defines route from a pipeline instance.

```
rule ConnectFromPipeline ::=
     'pipeline'  '/' Ident MaybePort
  ;
```

Rule ConnectToPipeline

The `ConnectToPipeline` rule defines route to a pipeline instance.

```
rule ConnectToPipeline ::=
     'pipeline'  '/' Ident MaybePort
  ;
```

Rule ConnectToConnector

The `ConnectToConnector` rule defines a route to a connector instance.

```
rule ConnectToConnector ::=
     'connector'  '/' Ident MaybePort
  ;
```

Rule DefineConnector

The `DefineConnector` rule defines a connector.

A connector is a runtime artefact that allows tremor to connect to the outside world, or for the outside connector to connect to tremor to send and/or receive data.

The named connector can be parameterized and instanciated via the `Create` rule

```
rule DefineConnector ::=
    DocComment  'define'  'connector' Ident  'from' ConnectorKind ArgsWithEnd
  ;
```

Rule DefineFlow

```
rule DefineFlow ::=
    DocComment  'define'  'flow' Ident DefinitionArgs  'flow' FlowStmts  'end'
  ;
```

Rule Query

Query Language Entrypoint

This is the top level rule of the tremor query language `trickle`

```
rule Query ::=
    ConfigDirectives Stmts  '<end-of-stream>' ?
  | Stmts  '<end-of-stream>' ?
  ;
```

Rule Stmts

The `Stmts` rule defines a `;` semi-colon delimited sequence of `Stmt` rules.

```
rule Stmts ::=
    Stmt  ';' Stmts
  | Stmt  ';' ?
  ;
```

Rule Stmt

The `Stmt` rule defines the legal statements in a query script.

Queries in tremor support: * Defining named `window`, `operator`, `script` and `pipeline` definitions. * Creating node instances of `stream`, `pipeline`, `operator` and `script` operations. * Linking nodes togther to form an execution graph via the `select` operation.

```
rule Stmt ::=
    Use
```

```
  | DefineWindow
  | DefineOperator
  | DefineScript
  | DefinePipeline
  | CreateOperator
  | CreateScript
  | CreatePipeline
  |  'create'  'stream' Ident
  |  'select' ComplexExprImut  'from' StreamPort WindowClause WhereClause GroupByClause  'in
  ;
```

Rule DefineWindow

The `DefineWindow` rule defines a temporal window specification.

A window is a mechanism that caches, stores or buffers events for processing over a finite temporal range. The time range can be based on the number of events, the wall clock or other defined parameters.

The named window can be instanciated via operations that support windows such as the `select` operation.

```
rule DefineWindow ::=
    DocComment  'define'  'window' Ident  'from' WindowKind CreationWith EmbeddedScriptImut
  ;
```

Rule DefineOperator

The `DefineOperator` rule defines an operator.

An operator is a query operation composed using the builtin operators provided by tremor written in the rust programming language.

The named operator can be parameterized and instanciated via the `CreateOperator` rule

```
rule DefineOperator ::=
    DocComment  'define'  'operator' Ident  'from' OperatorKind ArgsWithEnd
  ;
```

Rule DefineScript

The `DefineScript` rule defines a named operator based on a tremor script.

A script operator is a query operation composed using the scripting language DSL rather than the builtin operators provided by tremor written in the rust programming language.

The named script can be parameterized and instanciated via the `CreateScript` rule

```
rule DefineScript ::=
    DocComment  'define'  'script' Ident DefinitionArgs EmbeddedScript
  ;
```

Rule DefinePipeline

The `DefinePipeline` rule creates a named pipeline.

A pipeline is a query operation composed using the query langauge DSL instead of a builtin operation provided by tremor written in the rust programming language.

The named pipeline can be parameterized and instanciated via the `CreatePipeline` rule

```
rule DefinePipeline ::=
    DocComment  'define'  'pipeline' Ident (  'from' Ports ) ?  (  'into' Ports ) ?  Definit
  ;
```

Rule CreateScript

The `CreateScript` rule creates an operator based on a tremor script.

A script operator is a query operation composed using the scripting language DSL rather than the builtin operators provided by tremor written in the rust programming language.

The rule causes an instance of the referenced script definition to be created an inserted into the query processing execution graph.

```
rule CreateScript ::=
     'create'  'script' Ident CreationWithEnd
  |  'create'  'script' Ident  'from' ModularTarget CreationWithEnd
  ;
```

Rule CreateOperator

The `CreateOperator` rule creates an operator.

An operator is a query operation composed using the builtin operators provided by tremor written in the rust programming language.

The rule causes an instance of the referenced operator definition to be created an inserted into the query processing execution graph.

```
rule CreateOperator ::=
     'create'  'operator' Ident CreationWithEnd
```

```
  |  'create'  'operator' Ident  'from' ModularTarget CreationWithEnd
  ;
```

Rule CreatePipeline

The `CreatePipeline` rule creates a pipeline.

A pipeline is a query operation composed using the query langauge DSL instead of a builtin operation provided by tremor written in the rust programming language.

The rule causes an instance of the referenced pipeline definition to be created an inserted into the query processing execution graph.

```
rule CreatePipeline ::=
     'create'  'pipeline' Ident CreationWithEnd
  |  'create'  'pipeline' Ident  'from' ModularTarget CreationWithEnd
  ;
```

Rule MaybePort

The `MaybePort` rule defines an optional `Port`.

```
rule MaybePort ::=
    (  '/' Ident ) ?
  ;
```

Rule StreamPort

The `StreamPort` rule defines a stream by name with an optional named `Port`.

When the `Port` is omitted, tremor will internally default the `Port` to the appropriate `in` or `out` port. Where the `err` or user defined `Port`s are preferred, the optional `Port` specification SHOULD be provided.

```
rule StreamPort ::=
    Ident MaybePort
  ;
```

Rule WindowKind

Tumbling

A `tumbling` window defines a wall-clock-bound or data-bound window of non-overlapping time for storing events. The windows can not overlap, and there are no gaps between windows permissible.

Sliding

A `sliding` window defines a wall-clock-bound or data-bound window of events that captures an intervalic window of events whose extent derives from the size of the window. A sliding window of size 2 captures up to to events. Every subsequent event will evict the oldest and retain the newest event with the previous (now oldest) event.

Conditioning

Both kinds of window store events in arrival order

```
rule WindowKind ::=
     'sliding'
  |  'tumbling'
  ;
```

Rule WindowClause

The `WindowClause` rule defines an optional window definition for a supporting operation.

```
rule WindowClause ::=
    ( WindowDefn ) ?
  ;
```

Rule Windows

The `Windows` rule defines a sequence of window definitions that are , comma delimited.

```
rule Windows ::=
    Windows_
  ;
```

Rule Windows_

The `Windows_` rule defines a sequence of window definitions that are , comma delimited.

```
rule Windows_ ::=
    Sep!(Windows_, Window, ",")
  ;
```

Rule Window

The `Window` rule defines a modular target to a window definition.

```
rule Window ::=
    ModularTarget
  ;
```

Rule WindowDefn

The `WindowDefn` defines a temporal basis over which a stream of events is applicable.

```
rule WindowDefn ::=
     '[' Windows  ']'
  ;
```

Rule WhereClause

The `WhereClause` defines a predicate expression used to filter (forward or discard) events in an operation.

The `where` clause is executed before a operation processes an event.

```
rule WhereClause ::=
    (  'where' ComplexExprImut ) ?
  ;
```

Rule HavingClause

The `HavingClause` defines a predicate expression used to filter (forward or discard) events in an operation.

The `having` clause is executed after an operation has processed an event.

```
rule HavingClause ::=
    (  'having' ComplexExprImut ) ?
  ;
```

Rule GroupByClause

The `GroupByClause` defines the group by clause of a supporting operation in tremor.

An operator that uses a group by clause maintains the operation for each group captured by the grouping dimensions specified in this clause.

```
rule GroupByClause ::=
    (  'group'  'by' GroupDef ) ?
  ;
```

Rule GroupDef

The `GroupDef` rule defines the parts of a grouping dimension.

Group segments can be derived from: * Expressions - for which their serialized values are used. * Set expressions - which computes a set based on an expression. * Each expressions - which iterates an expression to compute a set.

```
rule GroupDef ::=
    ExprImut
  |  'set'  '(' GroupDefs  ')'
  |  'each'  '(' ExprImut  ')'
  ;
```

Rule GroupDefs

The `GroupDefs` rule defines a , comma delimited set of `GroupDef` rules.

```
rule GroupDefs ::=
    GroupDefs_
  ;
```

Rule GroupDefs_

The `GroupDefs_` rule defines a , comma delimited set of `GroupDef` rules.

```
rule GroupDefs_ ::=
    Sep!(GroupDefs_, GroupDef, ",")
  ;
```

Rule EmbeddedScriptImut

The `EmbeddedScriptImut` rule defines an optional embedded `script`.

```
rule EmbeddedScriptImut ::=
    (  'script' EmbeddedScriptContent ) ?
  ;
```

Rule EmbeddedScriptContent

The `EmbeddedScriptContent` rule defines an embedded script expression.

```
rule EmbeddedScriptContent ::=
    ExprImut
  ;
```

Rule Ports

The `Ports` rule defines a `,` comma delimited set of stream ports.

```
rule Ports ::=
    Sep!(Ports, <Ident>, ",")
  ;
```

Rule OperatorKind

The `OperatorKind` rule defines a modular path like reference to a builtin tremor operator.

Operators are programmed in rust native code and referenced via a virtual module path.

```
rule OperatorKind ::=
    Ident  '::' Ident
  ;
```

Rule EmbeddedScript

The `EmbeddedScript` rule defines a script using the Script DSL [Full].

The script is enclosed in `script` .. `end` blocks.

```
rule EmbeddedScript ::=
     'script' TopLevelExprs  'end'
  ;
```

Rule Pipeline

The `Pipeline` rule defines a block of statements in a `pipeline` .. `end` block.

The block MAY begin with an optional set of `ConfigDirectives`.

```
rule Pipeline ::=
     'pipeline' ConfigDirectives ?  PipelineCreateInner  'end'
  ;
```

Rule PipelineCreateInner

The `PipelineCreateInner` is an internal rule of the `Pipeline` rule.

The rule defines a `;` semi-colon delimited set of one or many `Stmt`s.

```
rule PipelineCreateInner ::=
    Stmt  ';' Stmts
  | Stmt  ';' ?
  ;
```

Rule Script

The `Script` rule defines the logical entry point into Tremor's expression oriented scripting language. The scripting langauge can be embedded into queries via the `script` operator. The scripting language is also used to specify configuration of connectors, pipelines, flows, and operators in the query language.

A legal script is composed of: * An optional set of module comments * A sequence of top level expressions. There must be at least one defined. * An optional end of stream token

```
rule Script ::=
    ModComment TopLevelExprs  '<end-of-stream>' ?
  ;
```

Type system

Tremor supports a data oriented or value based type system with a syntax that is backwards compatible with JSON.

Any well-formed and legal JSON document is a valid literal in tremor.

Tremor literals for `null`, `boolean`, `string` (utf-8), integer (64-bit unsigned), float (64-bit ieee), arrays, and records are equivalent to their JSON counterparts.

Tremor also supports a binary literal for transporting and processing opaque binary data.

Asymmetric

JSON literals are valid tremor value literals.

Tremor literals MAY NOT always be valid JSON literal.

```
# The following literal is valid JSON and valid Tremor
[1, "snot", {}];

# The following literal is valid in tremor only
[1, "snot", {}, << data/binary >>, ];
```

Tremor supports comments, JSON does not. Tremor supports trailing commas in arrays and records, JSON does not. Tremor supports binary literal data, JSON does not.

Note: By default, most connectors in tremor serialize to and from `json` via a codec. The type system in tremor however is agnostic to the wire format of data that flows through tremor. So data originate as `json`, as `msgpack`.

Computations

Tremor also supports a rich expression language with the same support for additive, mutliplicate, comparitive, and logical unary and binary expressions as languages like `rust` and `java`.

As most of the data that flows through tremor is heirarchically structured or JSON-like tremor also has rich primitives for structural pattern matching, structural comprehension or iterating over data structures.

Loops

Tremor does not support `while` `loop` or other primitives that can loop, recurse or iterate indefinitely.

In an event based system, events are streaming continuously - so infinite loops that can block streams from making forward progress are considered harmful.

There are no loops.

We do support iteration over finite arrays.

We do support depth-limited tail recursive functional programming.

Expression oriented

The script processing is expression oriented. This is to say that every structural form supported by tremor returns a data structure as a result.

Event oriented

Scripts in tremor can `emit` or `drop` an ‘event that is being processed.

The `event` keyword is the subject. It identifies the value currently being processed.

The `emit` keyword halts processing succesfully with a value.

The `drop` keyword halts processing by discarding the current event.

Illustrative example

```
# Propagate events marked as important and convert them to system alerts
match event of
  case %{ present important } => { "alert": event.message }
  default => drop
end;
```

Rule TopLevelExprs

The `TopLevelExprs` rule defines semi-colon separated sequence of top level tremor expressions with an optional terminating semi-colon

```
rule TopLevelExprs ::=
    TopLevelExpr  ';' TopLevelExprs
  | TopLevelExpr  ';' ?
  ;
```

Rule InnerExprs

The `InnerExprs` rule defines the expression forms permissible within another containing scope. Like `TopLevelExprs`, inner expressions are separated by semi-colons. The semi-colon is optional for the last expression in a set of expressions.

At least one expression MUST be provided.

```
rule InnerExprs ::=
    Expr  ';' InnerExprs
  | Expr  ';' ?
  ;
```

Rule TopLevelExpr

The `TopLevelExpr` rule specifies the expression forms that are legal at the outer most scope of a tremor script definition.

The legal forms are: * Use declarations - these allow external modules to be referenced. * Constant expressions - these are immutable compile time constants. * Function definitions - these are user defined functions. * Intrinsic function definitions - these are builtin funtions provided by the runtime.

```
rule TopLevelExpr ::=
    Const
  | FnDefn
  | Intrinsic
  | Expr
  | Use
  ;
```

Example

In the tremor standard library many of the top level expressions are `use` definitions importing sub modules from the module path.

```
use std::array;     # Import the std array utilities
use std::base64     # Import the std base64 utilities;
```

```
use std::binary;    # ...
use std::float;
use std::integer;
use std::json;
```

Rule Const

The `Const` rule defines a rule that binds an immutable expression to an identifier.

As the value cannot be changed at runtime.

```
rule Const ::=
    DocComment  'const' Ident  '=' ComplexExprImut
  ;
```

Example

```
use std::base64;
const snot = "snot";
const badger = "badger";
const snot_badger = { "#{snot}": "#{base64::encode(badger)}" };
```

Rule Expr

The `Expr` rule aliases the `SimpleExpr` rule.

The alias allows higher levels of the DSL such as the rules in the deployment or query language to avoid some of the internal complexity in the scripting language.

Within the scripting DSLs grammar the different forms and variations of expression are significant.

Hoewver, in the higher level we limit exposure to a subset of these forms. This is done for convenience, and for consistency of usage, and ease of learning the language.

```
rule Expr ::=
    SimpleExpr
  ;
```

Rule SimpleExpr

The `SimpleExpr` rule defines all the structural and simple expressions and literals in tremor.

```
rule SimpleExpr ::=
    Match
```

```
  | For
  | Let
  | Drop
  | Emit
  | ExprImut
  ;
```

Rule AlwaysImutExpr

The `AlwaysImutExpr` defines the immutable expression forms in tremor.

Immutable expressions can be reduced at compile time and folded into literals.

```
rule AlwaysImutExpr ::=
    Patch
  | Merge
  | Invoke
  | Literal
  | Path
  | Record
  | List
  | StringLiteral
  | BytesLiteral
  | Recur
  ;
```

Rule Recur

The `Recur` rule defines stack-depth-limited tail-recursion in tremor functions.

```
rule Recur ::=
     'recur'  '('  ')'
  |  'recur'  '(' InvokeArgs  ')'
  ;
```

Rule ExprImut

The `ExprImut` is the root of immutable expressions in tremor.

```
rule ExprImut ::=
    OrExprImut
  ;
```

Rule OrExprImut

The `OrExprImut` rule supports logical or expressions in tremor.

Binary logical or expressions take precedence over logical exclusive or expressions.

```
rule OrExprImut ::=
    BinOp!(BinOr, ExprImut, XorExprImut)
  | XorExprImut
  ;
```

Rule XorExprImut

The `XorExprImut` rule supports logical exclusive or expressions in tremor.

Binary logical exclusive or expressions take precedence over logical and expressions.

```
rule XorExprImut ::=
    BinOp!(BinXor, XorExprImut, AndExprImut)
  | AndExprImut
  ;
```

Rule AndExprImut

The `AndExprImut` rule supports logical and expressions in tremor.

Binary logical and expressions take precedence over bitwise or expressions.

```
rule AndExprImut ::=
    BinOp!(BinAnd, AndExprImut, BitOrExprImut)
  | BitOrExprImut
  ;
```

Rule BitOrExprImut

The `BitOrExprImut` rule supports bitwise or expressions in tremor.

Binary bitwise or expressions take precedence over bitwise exclusive or expressions.

```
rule BitOrExprImut ::=
    BitXorExprImut
  ;
```

Rule BitXorExprImut

The `BitXorExprImut` rule supports bitwise exclusive or expressions in tremor.

Binary bitwise exclusive or expressions take precedence over bitwise and expressions.

```
rule BitXorExprImut ::=
    BinOp!(BinBitXor, BitXorExprImut, BitAndExprImut)
  | BitAndExprImut
  ;
```

Rule BitAndExprImut

The `BitAndExprImut` rule supports bitwise and expressions in tremor.

Binary bitwise and expressions take precedence over equality expressions.

```
rule BitAndExprImut ::=
    BinOp!(BinBitAnd, BitAndExprImut, EqExprImut)
  | EqExprImut
  ;
```

Rule EqExprImut

The `EqExprImut` rule supports equality expressions in tremor.

Binary equality expressions take precedence over comparitive expressions.

```
rule EqExprImut ::=
    BinOp!(BinEq, EqExprImut, CmpExprImut)
  | CmpExprImut
  ;
```

Rule CmpExprImut

The `CmpExprImut` rule supports comparative expressions in tremor.

Binary comparative expressions take precedence over bit shift expressions.

```
rule CmpExprImut ::=
    BinOp!(BinCmp, CmpExprImut, BitShiftExprImut)
  | BitShiftExprImut
  ;
```

Rule BitShiftExprImut

The `BitShiftExprImut` rule supports bit shift expressions in tremor.

Binary bit shift expressions take precedence over bitwise additive expressions.

```
rule BitShiftExprImut ::=
    BinOp!(BinBitShift, BitShiftExprImut, AddExprImut)
  | AddExprImut
  ;
```

Rule AddExprImut

The `AddExprImut` rule supports additive expressions in tremor.

Binary additive expressions take precedence over multiplicative expressions.

```
rule AddExprImut ::=
    BinOp!(BinAdd, AddExprImut, MulExprImut)
  | MulExprImut
  ;
```

Rule MulExprImut

The `MulExprImut` rule supports multiplicative expressions in tremor.

Binary multiplicative expressions take precedence over unary expressions.

```
rule MulExprImut ::=
    BinOp!(BinMul, MulExprImut, UnaryExprImut)
  | UnaryExprImut
  ;
```

Rule UnaryExprImut

The `UnaryExprImut` rule specifies unary expression operations.

Expressions can be marked as + positive, - negative explicitly when needed.

Otherwise, the expression reduces to a simple unary expression.

The simple unary expression has lower precedence.

```
rule UnaryExprImut ::=
     '+' UnaryExprImut
  |  '-' UnaryExprImut
  | UnarySimpleExprImut
  ;
```

Rule UnarySimpleExprImut

The `UnarySimpleExprImut` rule specifies predicate unary expression operations.

Expressions can be marked explicitly with `not` or `!` to negate the target simple presence expression.

Otherwise, the expression reduces to a simple presence expression.

The simple presence expression has lower precedence.

```
rule UnarySimpleExprImut ::=
     'not' UnarySimpleExprImut
  |  '!' UnarySimpleExprImut
  | PresenceSimplExprImut
  ;
```

Rule PresenceSimplExprImut

The `PresenceSimplExprImut` rule specifies presence and simple expressions

Expressions path predicate tests based on the `present` and `absent` predicate test expressions, or a simple expression.

Otherwise, the expression reduces to a simple expression.

The simple expression has lower precedence.

```
rule PresenceSimplExprImut ::=
     'present' Path
  |  'absent' Path
  | SimpleExprImut
  ;
```

Rule ComplexExprImut

The `ComplexExprImut` rule defines complex immutable expression in tremor.

```
rule ComplexExprImut ::=
    MatchImut
  | ForImut
  | ExprImut
  ;
```

Rule Intrinsic

The `intrinsic` rule defines intrinsic function signatures.

This rule allows tremor maintainers to document the builtin functions implemented as native rust code. The facility also allows document generation tools to document builtin intrinsic functions in the same way as user defined functions.

In short, these can be thought of as runtime provided.

For information on how to define user defined functions see the function rule.

```
rule Intrinsic ::=
    DocComment  'intrinsic'  'fn' Ident  '('  ')'  'as' ModularTarget
  | DocComment  'intrinsic'  'fn' Ident  '(' FnArgs  ')'  'as' ModularTarget
  | DocComment  'intrinsic'  'fn' Ident  '(' FnArgs  ','  '.'  '.'  '.'  ')'  'as' ModularTa
```

```
  | DocComment  'intrinsic'  'fn' Ident  '('  '.'  '.'  '.'  ')'  'as' ModularTarget
  ;
```

Example

From our standard library generated documentation, we can see that the base64 encode function is an intrinsic function.

```
## Encodes a `binary` as a base64 encoded string
##
## Returns a `string`
intrinsic fn encode(input) as base64::encode;
```

Rule FnDefn

```
rule FnDefn ::=
    DocComment  'fn' Ident  '('  '.'  '.'  '.'  ')'  'with' InnerExprs  'end'
  | DocComment  'fn' Ident  '(' FnArgs  ','  '.'  '.'  '.'  ')'  'with' InnerExprs  'end'
  | DocComment  'fn' Ident  '('  ')'  'with' InnerExprs  'end'
  | DocComment  'fn' Ident  '(' FnArgs  ')'  'with' InnerExprs  'end'
  | DocComment  'fn' Ident  '('  ')'  'of' FnCases  'end'
  | DocComment  'fn' Ident  '(' FnArgs  ')'  'of' FnCases  'end'
  ;
```

Rule FnCases

The `FnCases` rule defines a sequence of cases for structural pattern matching in tremor pattern functions.

```
rule FnCases ::=
    FnCaseClauses FnCaseDefault
  | FnCaseDefault
  ;
```

Rule FnCaseDefault

The `FnCaseDefines` rule defines a default match clause for use in pattern match function signatures in tremor.

```
rule FnCaseDefault ::=
     'default' Effectors
  ;
```

Rule FnCase

The `FnCase` rule defines an array predicate pattern supporting match clause for use in pattern match function signatures in tremor.

```
rule FnCase ::=
     'case'  '(' ArrayPredicatePatterns  ')' WhenClause Effectors
  ;
```

Rule FnCaseClauses

The `FnCaseClauses` defines the case syntax to structurally matched function signatures in tremor.

```
rule FnCaseClauses ::=
    FnCase
  | FnCaseClauses FnCase
  ;
```

Rule FnArgs

The `FnArgs` rule defines , comma delimited arguments to a tremor function.

```
rule FnArgs ::=
    Ident
  | FnArgs  ',' Ident
  ;
```

Rule SimpleExprImut

The `SimpleExprImut` rule defines optionally parenthesized simple immutable expressions in tremor.

```
rule SimpleExprImut ::=
     '(' ComplexExprImut  ')'
  | AlwaysImutExpr
  ;
```

Rule Literal

The `Literal` rule defines the set of primitive literals supported in tremor.

```
rule Literal ::=
    Nil
  | Bool
  | Int
  | Float
  ;
```

Rule Nil

```
rule Nil ::=
     'nil'
  ;
```

Example

```
null # The `null` literal value
```

Rule Bool

The `Bool` rule defines the syntax of boolean literal in tremor.

```
rule Bool ::=
     'bool'
  ;
```

Example

```
true # The boolean `true` literal
```

```
false # The boolean `false` literal
```

Rule Int

The `Int` rule literal specifes the syntax of integer literals in tremor.

```
rule Int ::=
     'int'
  ;
```

Rule Float

The `Float` rule literal specifes the syntax of IEEE float literals in tremor.

```
rule Float ::=
     'float'
  ;
```

Rule StringLiteral

The `StringLiteral` rule defines a string literal in tremor.

Strings are `"` single-quote or `"""` triple-quote delimited blocks of UTF-8 text.

A single-quote string is a single line string, supporting sting interpolation.

A triple-quote string is a multi-line string, supporting sting interpolation.

```
rule StringLiteral ::=
     'heredoc_start' StrLitElements  'heredoc_end'
  |  '\\' StrLitElements  '\\'
  |  '\\'  '\\'
  ;
```

Rule StrLitElements

The `StrLitElements` rule defines the internal structure of a string literal in tremor.

String literal in tremor support string interpolation via the `#{` and `}` escape sequence. Content within the escape sequence can be any legal and valid tremor expression.

```
rule StrLitElements ::=
    StringPart StrLitElements
  |  '\\\\#' StrLitElements
  |  '#{' ExprImut  '}' StrLitElements
  | StringPart
  |  '\\\\#'
  |  '#{' ExprImut  '}'
  ;
```

Rule StringPart

The `StringPart` rule defines a simple or heredoc style string part.

```
rule StringPart ::=
     'string'
  |  'heredoc'
  ;
```

Rule List

The `List` rule defines a `[` and `]` square bracket delimited sequence of zero or many ‘,’ delimited expressions.

```
rule List ::=
     '[' ListElements  ']'
  |  '['  ']'
  ;
```

Rule ListElements

The `ListElements` rule defines a `,` comma delimited sequence of expression elements.

```
rule ListElements ::=
    ListElements_
  ;
```

Rule ListElements_

The `ListElements_` rule is internal to the `ListElements` rule.

The rule defines a sequence of `,` comma delimited expression elements using the `Sep` macro rule.

```
rule ListElements_ ::=
    Sep!(ListElements_, ComplexExprImut, ",")
  ;
```

Rule Record

The `Record` rule defines a set of name-value pairs delimited by `,` a comma.

Records are enclosed in `{` and `}` curly braces.

The record structure in tremor is backwards compatible with JSON.

All JSON records can be read by tremor.

Not all tremor records can be read by a JSON reader as tremor supports computations, comments and trailiing `,` commas in its record and array structures.

```
rule Record ::=
     '{' Fields  '}'
  |  '{'  '}'
  ;
```

Rule Field

The `Field` rule defines a `:` colon delimited name value pair for a record literal.

The name is a string literal.

The value is an expression.

```
rule Field ::=
    StringLiteral  ':' ComplexExprImut
  ;
```

Rule Path

The `Path` rule defines path operations over expressions.

Path operations structures to be tersely indexed in a path like structure.

Path operations are supported on * A subset of expressions (record, array, function) * Meta keywords like `$`, `args`, `state`, `event`, `group`, `window`

```
rule Path ::=
    MetaPath
  | EventPath
  | StatePath
  | LocalPath
  | ConstPath
  | AggrPath
  | ArgsPath
  | ExprPath
  ;
```

Rule ExprPathRoot

The `ExprPathRoot` rule defines a subset of expressions where path operations are supported.

These are: * Record literals or references to records. * Array literals or references to arrays. * The result of function invocations. * The result of Parenthetic expressions.

```
rule ExprPathRoot ::=
     '(' ComplexExprImut  ')'
  | Invoke
  | Record
  | List
  ;
```

Rule ExprPath

The `ExprPath` rule defines path operations for expressions.

```
rule ExprPath ::=
    ExprPathRoot PathSegments
  ;
```

Rule MetaPath

The `MetaPath` rule defines path operations for event metadata references.

In the context of a streaming event, allows metadata generated by the runtime to be accessed via path operations.

It is also possible to write to metadata to hint at the runtime to perform certain functions on the event data being forwarded. Tremor operators and connectors can read and write metadata.

```
rule MetaPath ::=
     '$' Ident PathSegments
  |  '$' Ident
  |  '$'
  ;
```

Rule AggrPath

The `AggrPath` rule defines path operations for `group` and `window` references.

In the context of a windowed operation, enables the `group` and `window` meta keywords to partipcate in path operations.

```
rule AggrPath ::=
     'group' PathSegments
  |  'group'
  |  'window' PathSegments
  |  'window'
  ;
```

Rule ArgsPath

The `ArgsPath` rule defines path operations for `args` references.

```
rule ArgsPath ::=
     'args' PathSegments
  |  'args'
  ;
```

Rule LocalPath

The `LocalPath` rule enables path operations on locally scoped identifiers.

```
rule LocalPath ::=
    Ident PathSegments
  | Ident
  ;
```

Rule ConstPath

The `ConstPath` rule enables path operations on module scoped references.

```
rule ConstPath ::=
    ModPath  '::' LocalPath
  ;
```

Rule StatePath

The `StatePath` rule defines path operations for user defined in memory state in tremor.

Allows the `state` value to be dereferenced via path operations.

```
rule StatePath ::=
     'state' PathSegments
  |  'state'
  ;
```

Rule EventPath

The `EventPath` rule defines path operations for streaming events in tremor.

Allows the current streaming `event` to be dereferenced via path operations.

```
rule EventPath ::=
     'event' PathSegments
  |  'event'
  ;
```

Rule PathSegments

The `PathSegments` rule specifies the continuation of a path rule.

Form Variation	Description
`.<Ident>`	A terminal segment dereferencing a record field
`<Ident><PathSegments>`	A non-terminal segment dereferencing a record field
`[<Selector>]`	A range or index segment dereferencing an array
`[<Selector>]`	A terminal range or index segment dereferencing an array
`[<Selector>]<PathSegments>`	A non-terminal range or index segment dereferencing an array

```
rule PathSegments ::=
     '.' Ident PathSegments
  |  '[' Selector  ']' PathSegments
  |  '[' Selector  ']'
  |  '.' Ident
  ;
```

Rule Selector

The `Selector` rule specifies an index or range of an array.

A range is a `:` colon separated pair of expressions.

An index is a single expression.

```
rule Selector ::=
    ComplexExprImut  ':' ComplexExprImut
  | ComplexExprImut
  ;
```

Rule Invoke

The `Invoke` rule specifies the syntax of a function invocation.

```
rule Invoke ::=
    FunctionName  '(' InvokeArgs  ')'
  | FunctionName  '('  ')'
  ;
```

Rule FunctionName

The `FunctionName` rule defines a path to a function in tremor.

It can be an `Ident` for functions defined in local scope.

It can be a `ModPath` for functions in a modular scope.

```
rule FunctionName ::=
    Ident
  | ModPath  '::' Ident
  ;
```

Rule ModPath

The `ModPath` rule defines a modular path.

A modular path is a sequence of `Ident`s separated by a `::` double-colon.

```
rule ModPath ::=
    ModPath  '::' Ident
  | Ident
  ;
```

Rule InvokeArgs

The `InvokeArgs` rule defines a sequence of expression statements.

```
rule InvokeArgs ::=
    InvokeArgs_
  ;
```

Rule InvokeArgs_

The `InvokeArgs_` rule is an internal rule of the `InvokeArgs` rule.

The rule specifies a `;` semi-colon delimited sequence of expression statements.

```
rule InvokeArgs_ ::=
    Sep!(InvokeArgs_, ComplexExprImut, ",")
  ;
```

Rule Drop

Drop halts event processing for the current event being processed returning control to the tremor runtime, dropping the event.

Constraints

The `drop` operation should be used with care as the in-flight event is discarded by the runtime. Where circuit breakers, guaranteed delivery and quality of service operations are being managed by the engine downstream these should be carefully programmed so that `drop` operations have no side-effects on non-functional behaviours of the tremor runtime.

Here be dragons!

```
rule Drop ::=
     'drop'
  ;
```

Rule Emit

Emit halts event processing for the current event being processed returning control to the tremor runtime, emitting a synthetic event as output.

By default, the emit operation will emit events to the standard output port `out`.

The operation can be redirected to an alternate output port.

```
rule Emit ::=
     'emit' ComplexExprImut  '=>' StringLiteral
  |  'emit' ComplexExprImut
  |  'emit'  '=>' StringLiteral
```

```
  |  'emit'
  ;
```

Rule Let

The `Let` rule allows an expression to be bound to a `Path`.

The `Path` references the subject of the assignment based on tremor's `Path` rules.

The bound `Path` is mutable.

```
rule Let ::=
     'let' Assignment
  ;
```

Rule Assignment

The `Assignment` rule allows an expression to be bound to a `Path`.

The `Path` references the subject of the assignment based on tremor's `Path` rules.

```
rule Assignment ::=
    Path  '=' SimpleExpr
  ;
```

Rule Patch

The `Patch` rule defines the `patch` statement in tremor.

```
rule Patch ::=
     'patch' ComplexExprImut  'of' PatchOperations  'end'
  ;
```

Rule PatchOperations

The `PatchOperations` rule defines a sequence of semi-colon delimited patch operations.

```
rule PatchOperations ::=
    PatchOperationClause
  | PatchOperations  ';' PatchOperationClause
  ;
```

Rule PatchField

The `PatchField` is a string literal identifying a the field of a record to which a `PatchOperationClause` is being applied.

```
rule PatchField ::=
    StringLiteral
  ;
```

Rule PatchOperationClause

The `PatchOperationClause` rule defines operations of a `patch` statement.

A patch operation can: * Insert, update, copy (clone), move (rename), merge or erase fields in a record. * Apply a default operation on a field or on the whole input record.

```
rule PatchOperationClause ::=
     'insert' PatchField  '=>' ComplexExprImut
  |  'upsert' PatchField  '=>' ComplexExprImut
  |  'update' PatchField  '=>' ComplexExprImut
  |  'erase' PatchField
  |  'move' PatchField  '=>' PatchField
  |  'copy' PatchField  '=>' PatchField
  |  'merge' PatchField  '=>' ComplexExprImut
  |  'merge'  '=>' ComplexExprImut
  |  'default' PatchField  '=>' ComplexExprImut
  |  'default'  '=>' ComplexExprImut
  ;
```

Rule Merge

The `Merge` rule defines a merge operation of two complex immutable expressions.

```
rule Merge ::=
     'merge' ComplexExprImut  'of' ComplexExprImut  'end'
  ;
```

Rule For

The `For` rule defines an mutable `for` comprehension.

```
rule For ::=
     'for' ComplexExprImut  'of' ForCaseClauses  'end'
  ;
```

Rule ForCaseClauses

The `ForCaseClausest` defines a sequence of case clauses in an mutable `for` comprehension.

```
rule ForCaseClauses ::=
    ForCaseClause
  | ForCaseClauses ForCaseClause
  ;
```

Rule ForCaseClause

The `ForCaseClause` defines the case clause for mutable `for` comprehensions.

```
rule ForCaseClause ::=
     'case'  '(' Ident  ',' Ident  ')' WhenClause Effectors
  ;
```

Rule ForImut

The `ForImut` rule defines an immutable `for` comprehension.

```
rule ForImut ::=
     'for' ComplexExprImut  'of' ForCaseClausesImut  'end'
  ;
```

Rule ForCaseClausesImut

The `ForCaseClausesImut` defines a sequence of case clauses in an immutable `for` comprehension.

```
rule ForCaseClausesImut ::=
    ForCaseClauseImut
  | ForCaseClausesImut ForCaseClauseImut
  ;
```

Rule ForCaseClauseImut

The `ForCaseClauseImut` defines the case clause for immutable `for` comprehensions.

```
rule ForCaseClauseImut ::=
     'case'  '(' Ident  ',' Ident  ')' WhenClause EffectorsImut
  ;
```

Record Comprehension

```
for { "snot": "badger" } of
  case (name, value) => value
end;
```

Array Comprehension

```
for [1, "foo", 2, "bar"] of
  case (index, value) => value
end;
```

Rule Match

The `Match` rule defines a mutable match statement in tremor.

```
rule Match ::=
     'match' ComplexExprImut  'of' Predicates  'end'
  ;
```

Rule Predicates

The `Predicates` rule defines a sequence of mutable `PredicateClause` rules in tremor.

```
rule Predicates ::=
    PredicateClause
  | Predicates PredicateClause
  ;
```

Rule PredicateClause

The `PredicateClause` rule defines the forms of a mutable match statement in tremor.

```
rule PredicateClause ::=
     'case' CasePattern WhenClause Effectors
  |  'default' Effectors
  ;
```

Rule Effectors

The `Effectors` rule defines an effect block.

```
rule Effectors ::=
     '=>' Block
  ;
```

Rule Block

The `Block` rule defines a semi-colon delimited set of `Expr` rules.

```
rule Block ::=
    Expr
  | Block  ';' Expr
  ;
```

Rule MatchImut

The `MatchImut` rule defines a `match` statement in tremor.

```
rule MatchImut ::=
     'match' ComplexExprImut  'of' PredicatesImut  'end'
  ;
```

Rule PredicatesImut

The `PredicatesImut` rule defines a sequence of `PredicateClauseImut` rules.

```
rule PredicatesImut ::=
    PredicateClauseImut
  | PredicatesImut PredicateClauseImut
  ;
```

Rule CasePattern

The `CasePattern` rule defines the valid structural pattern matching forms available in a match statement's `case` clause.

```
rule CasePattern ::=
    RecordPattern
  | ArrayPattern
  | TuplePattern
  | ComplexExprImut
  |  '_'
  |  '~' TestExpr
  | Ident  '=' CasePattern
  ;
```

Rule PredicateClauseImut

The `PredicateClauseImut` rule defines valid clauses of a match statement.

Two forms are supported:

- A `case` expression with optional guard expression and mandatory effector block.
- A `default` case expression with effector block.

```
rule PredicateClauseImut ::=
     'case' CasePattern WhenClause EffectorsImut
  |  'default' EffectorsImut
  ;
```

Rule EffectorsImut

The `EffectorsImut` rule defines the result value block sequence of pattern rule.

The effectors block provides the result value of `case` and `default` clauses in match statements, for comprehensions.

```
rule EffectorsImut ::=
     '=>' BlockImut
  ;
```

Rule BlockImut

The `BlockImut` rule defines a comma delimited sequence of complex immutable expressions.

```
rule BlockImut ::=
    ComplexExprImut
  | BlockImut  ',' ComplexExprImut
  ;
```

Rule WhenClause

The `WhenClause` rule defines an optional guard expression.

```
rule WhenClause ::=
    (  'when' ComplexExprImut ) ?
  ;
```

Rule PredicateFieldPattern

The `PredicateFieldPattern` rule defines the legal predicate tests available within record patterns.

Record patterns can use: * Extractor test expressions against fields. * Record, array and tuple patterns against fields. * Equality and comparison predicate patterns against fields. * Presence patterns against fields.

```
rule PredicateFieldPattern ::=
    Ident  '~=' TestExpr
  | Ident  '=' Ident  '~=' TestExpr
  | Ident  '~=' RecordPattern
  | Ident  '~=' ArrayPattern
  | Ident  '~=' TuplePattern
  |  'present' Ident
  |  'absent' Ident
  | Ident BinCmpEq ComplexExprImut
  ;
```

Rule TestExpr

The `TestExpr` defines an extractor with an optional microformat body.

A test expression has a predicate component. The `Ident` defines the expected microformat the value being tested in a structural pattern match should conform to.

If this validates, then an optional microformat expression that is specific to the extractor named by the `Ident` is employed to extract content from the value into a value that tremor can process.

```
rule TestExpr ::=
    Ident TestLiteral
  ;
```

Rule RecordPattern

The `RecordPattern` defines structural patterns against record values.

Record patterns start with the `%{` operator and end with '}'.

Patterns may be empty `%{}`, or a sequence of record pattern fields.

Record patterns are search oriented based on predicate matching.

Ordinal, order or position based matching in records is not defined.

```
rule RecordPattern ::=
     '%{' PatternFields  '}'
  |  '%{'  '}'
  ;
```

Rule ArrayPattern

The `ArrayPattern` defines structural patterns against array values.

Array patterns start with the `%[` operator and end with `]`.

Patterns may be empty `%[]`, or a sequence of array predicate patterns.

Array patterns are search oriented based on predicate matching.

Where ordinal matching is needed then a `TuplePattern` may be preferential.

```
rule ArrayPattern ::=
     '%[' ArrayPredicatePatterns  ']'
  |  '%['  ']'
  ;
```

Rule TuplePattern

The `TuplePattern` defines structural patterns against tuple values.

Tuple patterns start with the `%(` operator and end with `)`.

Patterns may be empty `%()`, `%(...)` any, or a sequence of tuple patterns followed by an optional open tuple `...` match.

Tuple patterns are ordinal patterns defined against arrays.

Where search like predicate filters are preferential the `ArrayPattern` may be a better choice.

```
rule TuplePattern ::=
     '%(' TuplePredicatePatterns OpenTuple  ')'
  |  '%('  ')'
  |  '%('  '.'  '.'  '.'  ')'
  ;
```

Rule OpenTuple

The `OpenTuple` rule defines a tuple pattern that matches any element in a tuple from the position it is used and subseuent elements.

It can only be used as an optional final predicate in a `TuplePattern`.

```
rule OpenTuple ::=
    (  ','  '.'  '.'  '.' ) ?
  ;
```

Rule TuplePredicatePatterns

The `TuplePredicatePatterns` rule defines a set of comma delimited `TuplePredicatePattern` rules.

```
rule TuplePredicatePatterns ::=
    TuplePredicatePatterns  ',' TuplePredicatePattern
  | TuplePredicatePattern
  ;
```

Rule TuplePredicatePattern

The syntax of the `TuplePredicatePattern` is the same as that of the `ArrayPredicatePattern`.

```
rule TuplePredicatePattern ::=
    ArrayPredicatePattern
  ;
```

Rule ArrayPredicatePattern

The `ArrayPredicatePattern` rule defines predicate patterns for structural pattern matching against array values.

```
rule ArrayPredicatePattern ::=
     '~' TestExpr
  |  '_'
  | ComplexExprImut
  | RecordPattern
  ;
```

Rule ArrayPredicatePatterns

The `ArrayPredicatePatterns` rule defines a set of comma delimited `ArrayPredicatePattern` rules.

```
rule ArrayPredicatePatterns ::=
    ArrayPredicatePatterns  ',' ArrayPredicatePattern
  | ArrayPredicatePattern
  ;
```

Rule PatternFields

The `PatternFields` rule defines a set of comma delimited `PredicateFieldPattern` rules.

```
rule PatternFields ::=
    PatternFields_
  ;
```

Rule PatternFields_

The `PatternFields_` rule is a rule that defines a comma separated set of `PatternField` definitions.

The rule follows the semantics defined in the `Sep` macro.

```
rule PatternFields_ ::=
    Sep!(PatternFields_, PredicateFieldPattern, ",")
  ;
```

Rule Fields

The `Fields` rule defines a set of comma delimited `Field` rules.

```
rule Fields ::=
    Fields_
  ;
```

Rule Fields_

The `Fields_` rule is a rule that defines a comma separated set of field definitions.

The rule follows the semantics defined in the `Sep` macro.

```
rule Fields_ ::=
    Sep!(Fields_, Field, ",")
  ;
```

Rule Ident

An `Ident` is an identifier - a user defined name for a tremor value.

```
rule Ident ::=
     '<ident>'
  ;
```

Examples of identifiers

```
let snot = { "snot": "badger" };
```

Keyword escaping

Surrounding an identifier with a tick “’ allows keywords in tremor’s DSLs to be escaped

```
let `let` = 1234.5;
```

Emoji

You can even use emoji as identifiers via the escaping mechanism.

```
let ` ` = "rocket";
```

But we cannot think of any good reason to do so!

Rule TestLiteral

The `TestLiteral` rule specifies an extractor microformat block.

An extractor takes the general form:

```
Ident '|' MicroFormat '|'
```

Where

The `ident` is the name of a builtin extractor such as `json` or `base64`.

The `Microformat` content depends on the extractor being used

```
rule TestLiteral ::=
     '<extractor>'
  ;
```

Extracting JSON embedded within strings

```
let example = { "snot": "{\"snot\": \"badger\"" };
match example of
  case extraction=%{ snot ~= json|| } => extraction.snot
  default => "no match"
end;
```

When executed this will result in:

```
"badger"
```

Decoding base64 embedded within strings

```
let example = { "snot": "eyJzbm90IjogImJhZGdlciJ9Cg==" };
match example of
  case extraction=%{ snot ~= base64|| } => extraction.snot
  default => "no match"
end;
```

When executed this will result in:

```
"{\"snot\": \"badger\"}
```

Wrap and Extract

We can decode the base64 decoded string through composition:

```
let example = { "snot": "eyJzbm90IjogImJhZGdlciJ9Cg==" };
match example of
  case decoded = %{ snot ~= base64|| } =>
    match { "snot": decoded.snot } of
      case json = %{ snot ~= json|| } => json.snot.snot
```

```
      default => "no match - json"
    end
  default => "no match - base64"
end;
```

Rule BytesLiteral

The `BytesLiteral` is a representation of opaque binary data literals in tremor

The syntax is a subset of the bit syntax representation in the Erlang Programming Language.

We □ Erlang.

We □ bit syntax!

```
rule BytesLiteral ::=
     '<<'  '>>'
  |  '<<' Bytes  '>>'
  ;
```

Examples

```
# Import standard tremor binary utility functions
use std::binary;

# Structure of a TCP packet header
#  0                   1                   2                   3
#  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
# +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
# |          Source Port          |        Destination Port       |
# +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
# |                        Sequence Number                        |
# +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
# |                     Acknowledgment Number                     |
# +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
# | Offset|  Res. |     Flags     |             Window            |
# +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
# |            Checksum           |         Urgent Pointer        |
# +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
# |                    Options                    |    Padding    | IGNORED
# +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

# Record representation of a TCP packet
let event = {
  "src": {"port": 1234},
  "dst": {"port": 2345},
```

```
  "seq": event,
  "ack": 4567,
  "offset": 1,
  "res": 2,
  "flags": 3,
  "win": 4,
  "checksum": 5,
  "urgent": 6,
  "data": "snot badger!"
};

# Convert the record into a binary encoded TCP packet
binary::into_bytes(<<
  # Header segment
  event.src.port:16,  event.dst.port:16,
  event.seq:32,
  event.ack:32,
  event.offset:4, event.res:4, event.flags:8, event.win:16,
  event.checksum:16, event.urgent:16,
  # Data segment
  event.data/binary
>>)
```

Rule Bytes

The `Bytes` rule defines a sequence of bit syntax patterns in a binary tremor literal representation.

A legal sequence of bytes MUST contain at least one byte part segment.

Byte part segments are comma (‘,’) delimited.

```
rule Bytes ::=
    BytesPart
  | Bytes  ',' BytesPart
  ;
```

Example: How do I encode a TCP packet?

```
# Convert the record into a binary encoded TCP packet
binary::into_bytes(<<
  # Encode source and destination TCP ports, each 16 bits wide
  event.src.port:16,  event.dst.port:16,
  # Encode sequence, 32 bits wide
  event.seq:32,
  # Encode acknowldgement, 32 bits wide
  event.ack:32,
```

```
  # Encode TCP conditioning and flags fields
  event.offset:4, event.res:4, event.flags:8, event.win:16,
  # Encode checksum; and urgent bytes from first byte
  event.checksum:16, event.urgent:16,
  # Encode data using the encoded length of another binary literal
  event.data/binary
>>)
```

Rule BytesPart

The `BytesPart` rule represents sub segment of a binary encoded literal

If the part is the last segment in a bytes literal, it can be of arbitrary length.

If the part is not the last segment, it must specify its length in bits.

```
rule BytesPart ::=
    SimpleExprImut
  | SimpleExprImut  ':'  'int'
  | SimpleExprImut  '/' Ident
  | SimpleExprImut  ':'  'int'  '/' Ident
  ;
```

Form

The part may take the following general form

```
SimpleExprImut  ':'  'int'  '/' Ident
```

Where: * The ‘SimpleExprImut can be a literal or identifier to the data being encoded. * A optional size in bits, or defaulted based on the data being encoded. * An optional encoding hint as an identifier

Size constraints

The size must be zero or greater, up to and including but no larger than 64 bits.

Encoding Hints

Ident	Description
`binary`	Encoded in binary, using network (big) endian
`big-unsigned-integer`	Unsigned integer encoding, big endian
`little-unsigned-integer`	Unsigned integer encoding, little endian
`big-signed-integer`	Signed integer encoding, big endian
`little-signed-integer`	Signed integer encoding, little endian

Rule Sep

The `Sep` rule is a LALRPOP convenience that allows defining a macro rule template for a common sub rule sequence.

The `Sep` macro rule definition in tremor DSLs allows lists or sequences of expressions to be separated by a specified delimiter. The delimiter is optional for the final item in a list or sequence.

Argument	Description
T	The term rule - specifies what is to be separated
D	The delimiter rule - specifies how elements are separated
L	A list of accumulated terms

```
macro Sep<L, T, D> ::=
    T D L
  | T D ?
  ;
```

Rule BinOp

The `BinOp` rule is a LALRPOP convenience that allows defining a macro rule template for a common sub rule sequence.

The `BinOp` macro rule definition in tremor DSLs allows binary operations to be defined tersely

Argument	Description
Current	The current rule permissible for the LHS of the expression
Operation	The operation to be performeed
Next	The current rule permissible for the RHS of the expression

The macro imposes rule precedence where the left hand side expression takes higher precedence relative to the right hand side expression when interpreted by tremor.

Considerations

Tremor performs compile time optimizations such as constant folding. So literal expressions of the form `1 + 2` may compile to a constant (`3` in this case) and have no runtime cost.

```
macro BinOp<Op, Current, Next> ::=
    ( Current ) ( Op ) Next
```

```
  ;
```

Rule BinCmpEq

The `BinCmpEq` rule allows binary or comparative operations

Comparitive and Equality operations have the same precedence.

```
rule BinCmpEq ::=
    BinEq
  | BinCmp
  ;
```

Rule BinOr

The `BinOr` rule defines binary or operation

Operator	Description
`xor`	Binary or

```
rule BinOr ::=
     'or'
  ;
```

Rule BinXor

The `BinXor` rule defines binary exclusive or operation

Operator	Description
`xor`	Binary exlusive or

```
rule BinXor ::=
     'xor'
  ;
```

Rule BinAnd

The `BinAnd` rule defines binary and operation

Operator	Description
`and`	Binary and

```
rule BinAnd ::=
     'and'
  ;
```

Rule BinBitXor

The `BinBitXor` rule defines binary bitwise exlusive-or operation

Operator	Description
`^`	Binary logical `xor` exclusive or

```
rule BinBitXor ::=
     '^'
  ;
```

Rule BinBitAnd

The `BinBitAnd` rule defines binary bitwise and operation

Operator	Description
`&`	Binary logical `and`

```
rule BinBitAnd ::=
     '&'
  ;
```

Rule BinEq

The `BinEq` rule defines binary equality operations

Operator	Description
`==`	Binary equality
`!=`	Binary non-equality

```
rule BinEq ::=
     '=='
  |  '!='
  ;
```

Rule BinCmp

The `BinCmp` rule defines binary comparitive operations

Operator	Description
>=	Binary greater than or equal to
>	Binary greater than
<=	Binary less than or equal to
<	Binary less than

```
rule BinCmp ::=
     '>='
  |  '>'
  |  '<='
  |  '<'
  ;
```

Rule BinBitShift

The `BinBitShift` rule defines bit shift operations

Operator	Description
>>>	Binary bit shift right, with `1` injected
>>	Binary bit shift right, with `0` injected
<<	Binary bit shift left, with `0` injected

```
rule BinBitShift ::=
     '>>'
  |  '>>>'
  |  '<<'
  ;
```

Rule BinAdd

The `BinAdd` rule defines additive operations

Operator	Description
+	Binary addition
-	Binary subtraction

Note that the + binary operation is also used for string concatenation.

```
rule BinAdd ::=
     '+'
  |  '-'
  ;
```

Rule BinMul

The `BinMul` rule defines multiplicative operations

Operator	Description
*	Binary multiplication
/	Binary division
%	Binary modulo

```
rule BinMul ::=
     '*'
  |  '/'
  |  '%'
  ;
```

EBNF Grammar

This EBNF grammar was generated from: “/Users/dennis/code/oss/tremor-rs/tremor-www-idle/tremor-runtime/tremor-script/src/grammar.lalrpop”

```
rule Use ::=
     'use' ModularTarget
  |  'use' ModularTarget  'as' Ident
  ;

rule ConfigDirectives ::=
    ConfigDirective ConfigDirectives
  | ConfigDirective
  ;

rule ConfigDirective ::=
     '#!config' WithExpr
  ;

rule ArgsWithEnd ::=
    ArgsClause ?  WithEndClause
  |
  ;
```

```
rule DefinitionArgs ::=
    ArgsClause ?
  ;

rule ArgsClause ::=
     'args' ArgsExprs
  ;

rule ArgsExprs ::=
    Sep!(ArgsExprs, ArgsExpr, ",")
  ;

rule ArgsExpr ::=
    Ident  '=' ExprImut
  | Ident
  ;

rule CreationWithEnd ::=
    WithEndClause
  |
  ;

rule CreationWith ::=
    WithClause
  |
  ;

rule WithClause ::=
     'with' WithExprs
  ;

rule WithEndClause ::=
    WithClause  'end'
  ;

rule WithExprs ::=
    Sep!(WithExprs, WithExpr, ",")
  ;

rule WithExpr ::=
    Ident  '=' ExprImut
  ;

rule ModuleBody ::=
    ModComment ModuleStmts
```

```
  ;

rule ModuleFile ::=
    ModuleBody  '<end-of-stream>'
  ;

rule ModuleStmts ::=
    ModuleStmt  ';' ModuleStmts
  | ModuleStmt  ';' ?
  ;

rule ModuleStmt ::=
    Use
  | Const
  | FnDefn
  | Intrinsic
  | DefineWindow
  | DefineOperator
  | DefineScript
  | DefinePipeline
  | DefineConnector
  | DefineFlow
  ;

rule ModularTarget ::=
    Ident
  | ModPath  '::' Ident
  ;

rule DocComment ::=
    ( DocComment_ ) ?
  ;

rule DocComment_ ::=
     '<doc-comment>'
  | DocComment_  '<doc-comment>'
  ;

rule ModComment ::=
    ( ModComment_ ) ?
  ;

rule ModComment_ ::=
     '<mod-comment>'
  | ModComment_  '<mod-comment>'
  ;
```

```
rule Deploy ::=
    ConfigDirectives ModComment DeployStmts  '<end-of-stream>' ?
  | ModComment DeployStmts  '<end-of-stream>' ?
  ;

rule DeployStmts ::=
    DeployStmt  ';' DeployStmts
  | DeployStmt  ';' ?
  ;

rule DeployStmt ::=
    DefineFlow
  | DeployFlowStmt
  | Use
  ;

rule DeployFlowStmt ::=
    DocComment  'deploy'  'flow' Ident  'from' ModularTarget CreationWithEnd
  | DocComment  'deploy'  'flow' Ident CreationWithEnd
  ;

rule ConnectorKind ::=
    Ident
  ;

rule FlowStmts ::=
    FlowStmts_
  ;

rule FlowStmts_ ::=
    Sep!(FlowStmts_, FlowStmtInner, ";")
  ;

rule CreateKind ::=
     'connector'
  |  'pipeline'
  ;

rule FlowStmtInner ::=
    Define
  | Create
  | Connect
  | Use
  ;
```

```
rule Define ::=
    DefinePipeline
  | DefineConnector
  ;

rule Create ::=
     'create' CreateKind Ident  'from' ModularTarget CreationWithEnd
  |  'create' CreateKind Ident CreationWithEnd
  ;

rule Connect ::=
     'connect'  '/' ConnectFromConnector  'to'  '/' ConnectToPipeline
  |  'connect'  '/' ConnectFromPipeline  'to'  '/' ConnectToConnector
  |  'connect'  '/' ConnectFromPipeline  'to'  '/' ConnectToPipeline
  ;

rule ConnectFromConnector ::=
     'connector'  '/' Ident MaybePort
  ;

rule ConnectFromPipeline ::=
     'pipeline'  '/' Ident MaybePort
  ;

rule ConnectToPipeline ::=
     'pipeline'  '/' Ident MaybePort
  ;

rule ConnectToConnector ::=
     'connector'  '/' Ident MaybePort
  ;

rule DefineConnector ::=
    DocComment  'define'  'connector' Ident  'from' ConnectorKind ArgsWithEnd
  ;

rule DefineFlow ::=
    DocComment  'define'  'flow' Ident DefinitionArgs  'flow' FlowStmts  'end'
  ;

rule Query ::=
    ConfigDirectives Stmts  '<end-of-stream>' ?
  | Stmts  '<end-of-stream>' ?
  ;

rule Stmts ::=
```

```
    Stmt  ';' Stmts
  | Stmt  ';' ?
  ;

rule Stmt ::=
    Use
  | DefineWindow
  | DefineOperator
  | DefineScript
  | DefinePipeline
  | CreateOperator
  | CreateScript
  | CreatePipeline
  |  'create'  'stream' Ident
  |  'select' ComplexExprImut  'from' StreamPort WindowClause WhereClause GroupByClause  'in
  ;

rule DefineWindow ::=
    DocComment  'define'  'window' Ident  'from' WindowKind CreationWith EmbeddedScriptImut
  ;

rule DefineOperator ::=
    DocComment  'define'  'operator' Ident  'from' OperatorKind ArgsWithEnd
  ;

rule DefineScript ::=
    DocComment  'define'  'script' Ident DefinitionArgs EmbeddedScript
  ;

rule DefinePipeline ::=
    DocComment  'define'  'pipeline' Ident (  'from' Ports ) ?  (  'into' Ports ) ?  Definit
  ;

rule CreateScript ::=
     'create'  'script' Ident CreationWithEnd
  |  'create'  'script' Ident  'from' ModularTarget CreationWithEnd
  ;

rule CreateOperator ::=
     'create'  'operator' Ident CreationWithEnd
  |  'create'  'operator' Ident  'from' ModularTarget CreationWithEnd
  ;

rule CreatePipeline ::=
     'create'  'pipeline' Ident CreationWithEnd
  |  'create'  'pipeline' Ident  'from' ModularTarget CreationWithEnd
```

```
  ;

rule MaybePort ::=
    (  '/' Ident ) ?
  ;

rule StreamPort ::=
    Ident MaybePort
  ;

rule WindowKind ::=
     'sliding'
  |  'tumbling'
  ;

rule WindowClause ::=
    ( WindowDefn ) ?
  ;

rule Windows ::=
    Windows_
  ;

rule Windows_ ::=
    Sep!(Windows_, Window, ",")
  ;

rule Window ::=
    ModularTarget
  ;

rule WindowDefn ::=
     '[' Windows  ']'
  ;

rule WhereClause ::=
    (  'where' ComplexExprImut ) ?
  ;

rule HavingClause ::=
    (  'having' ComplexExprImut ) ?
  ;

rule GroupByClause ::=
    (  'group'  'by' GroupDef ) ?
  ;
```

```
rule GroupDef ::=
    ExprImut
  |  'set'  '(' GroupDefs  ')'
  |  'each'  '(' ExprImut  ')'
  ;

rule GroupDefs ::=
    GroupDefs_
  ;

rule GroupDefs_ ::=
    Sep!(GroupDefs_, GroupDef, ",")
  ;

rule EmbeddedScriptImut ::=
    (  'script' EmbeddedScriptContent ) ?
  ;

rule EmbeddedScriptContent ::=
    ExprImut
  ;

rule Ports ::=
    Sep!(Ports, <Ident>, ",")
  ;

rule OperatorKind ::=
    Ident  '::' Ident
  ;

rule EmbeddedScript ::=
     'script' TopLevelExprs  'end'
  ;

rule Pipeline ::=
     'pipeline' ConfigDirectives ?  PipelineCreateInner  'end'
  ;

rule PipelineCreateInner ::=
    Stmt  ';' Stmts
  | Stmt  ';' ?
  ;

rule Script ::=
    ModComment TopLevelExprs  '<end-of-stream>' ?
```

```
  ;

rule TopLevelExprs ::=
    TopLevelExpr  ';' TopLevelExprs
  | TopLevelExpr  ';' ?
  ;

rule InnerExprs ::=
    Expr  ';' InnerExprs
  | Expr  ';' ?
  ;

rule TopLevelExpr ::=
    Const
  | FnDefn
  | Intrinsic
  | Expr
  | Use
  ;

rule Const ::=
    DocComment  'const' Ident  '=' ComplexExprImut
  ;

rule Expr ::=
    SimpleExpr
  ;

rule SimpleExpr ::=
    Match
  | For
  | Let
  | Drop
  | Emit
  | ExprImut
  ;

rule AlwaysImutExpr ::=
    Patch
  | Merge
  | Invoke
  | Literal
  | Path
  | Record
  | List
  | StringLiteral
```

```
  | BytesLiteral
  | Recur
  ;

rule Recur ::=
     'recur'  '('  ')'
  |  'recur'  '(' InvokeArgs  ')'
  ;

rule ExprImut ::=
    OrExprImut
  ;

rule OrExprImut ::=
    BinOp!(BinOr, ExprImut, XorExprImut)
  | XorExprImut
  ;

rule XorExprImut ::=
    BinOp!(BinXor, XorExprImut, AndExprImut)
  | AndExprImut
  ;

rule AndExprImut ::=
    BinOp!(BinAnd, AndExprImut, BitOrExprImut)
  | BitOrExprImut
  ;

rule BitOrExprImut ::=
    BitXorExprImut
  ;

rule BitXorExprImut ::=
    BinOp!(BinBitXor, BitXorExprImut, BitAndExprImut)
  | BitAndExprImut
  ;

rule BitAndExprImut ::=
    BinOp!(BinBitAnd, BitAndExprImut, EqExprImut)
  | EqExprImut
  ;

rule EqExprImut ::=
    BinOp!(BinEq, EqExprImut, CmpExprImut)
  | CmpExprImut
  ;
```

```
rule CmpExprImut ::=
    BinOp!(BinCmp, CmpExprImut, BitShiftExprImut)
  | BitShiftExprImut
  ;

rule BitShiftExprImut ::=
    BinOp!(BinBitShift, BitShiftExprImut, AddExprImut)
  | AddExprImut
  ;

rule AddExprImut ::=
    BinOp!(BinAdd, AddExprImut, MulExprImut)
  | MulExprImut
  ;

rule MulExprImut ::=
    BinOp!(BinMul, MulExprImut, UnaryExprImut)
  | UnaryExprImut
  ;

rule UnaryExprImut ::=
     '+' UnaryExprImut
  |  '-' UnaryExprImut
  | UnarySimpleExprImut
  ;

rule UnarySimpleExprImut ::=
     'not' UnarySimpleExprImut
  |  '!' UnarySimpleExprImut
  | PresenceSimplExprImut
  ;

rule PresenceSimplExprImut ::=
     'present' Path
  |  'absent' Path
  | SimpleExprImut
  ;

rule ComplexExprImut ::=
    MatchImut
  | ForImut
  | ExprImut
  ;

rule Intrinsic ::=
```

```
    DocComment  'intrinsic'  'fn' Ident  '('  ')'  'as' ModularTarget
  | DocComment  'intrinsic'  'fn' Ident  '(' FnArgs  ')'  'as' ModularTarget
  | DocComment  'intrinsic'  'fn' Ident  '(' FnArgs  ','  '.'  '.'  '.'  ')'  'as' ModularTa
  | DocComment  'intrinsic'  'fn' Ident  '('  '.'  '.'  '.'  ')'  'as' ModularTarget
  ;

rule FnDefn ::=
    DocComment  'fn' Ident  '('  '.'  '.'  '.'  ')'  'with' InnerExprs  'end'
  | DocComment  'fn' Ident  '(' FnArgs  ','  '.'  '.'  '.'  ')'  'with' InnerExprs  'end'
  | DocComment  'fn' Ident  '('  ')'  'with' InnerExprs  'end'
  | DocComment  'fn' Ident  '(' FnArgs  ')'  'with' InnerExprs  'end'
  | DocComment  'fn' Ident  '('  ')'  'of' FnCases  'end'
  | DocComment  'fn' Ident  '(' FnArgs  ')'  'of' FnCases  'end'
  ;

rule FnCases ::=
    FnCaseClauses FnCaseDefault
  | FnCaseDefault
  ;

rule FnCaseDefault ::=
     'default' Effectors
  ;

rule FnCase ::=
     'case'  '(' ArrayPredicatePatterns  ')' WhenClause Effectors
  ;

rule FnCaseClauses ::=
    FnCase
  | FnCaseClauses FnCase
  ;

rule FnArgs ::=
    Ident
  | FnArgs  ',' Ident
  ;

rule SimpleExprImut ::=
     '(' ComplexExprImut  ')'
  | AlwaysImutExpr
  ;

rule Literal ::=
    Nil
  | Bool
```

```
  | Int
  | Float
  ;

rule Nil ::=
     'nil'
  ;

rule Bool ::=
     'bool'
  ;

rule Int ::=
     'int'
  ;

rule Float ::=
     'float'
  ;

rule StringLiteral ::=
     'heredoc_start' StrLitElements  'heredoc_end'
  |  '\\' StrLitElements  '\\'
  |  '\\'  '\\'
  ;

rule StrLitElements ::=
    StringPart StrLitElements
  |  '\\\\#' StrLitElements
  |  '#{' ExprImut  '}' StrLitElements
  | StringPart
  |  '\\\\#'
  |  '#{' ExprImut  '}'
  ;

rule StringPart ::=
     'string'
  |  'heredoc'
  ;

rule List ::=
     '[' ListElements  ']'
  |  '['  ']'
  ;

rule ListElements ::=
```

```
    ListElements_
  ;

rule ListElements_ ::=
    Sep!(ListElements_, ComplexExprImut, ",")
  ;

rule Record ::=
     '{' Fields  '}'
  |  '{'  '}'
  ;

rule Field ::=
    StringLiteral  ':' ComplexExprImut
  ;

rule Path ::=
    MetaPath
  | EventPath
  | StatePath
  | LocalPath
  | ConstPath
  | AggrPath
  | ArgsPath
  | ExprPath
  ;

rule ExprPathRoot ::=
     '(' ComplexExprImut  ')'
  | Invoke
  | Record
  | List
  ;

rule ExprPath ::=
    ExprPathRoot PathSegments
  ;

rule MetaPath ::=
     '$' Ident PathSegments
  |  '$' Ident
  |  '$'
  ;

rule AggrPath ::=
     'group' PathSegments
```

```
  |  'group'
  |  'window' PathSegments
  |  'window'
  ;

rule ArgsPath ::=
     'args' PathSegments
  |  'args'
  ;

rule LocalPath ::=
    Ident PathSegments
  | Ident
  ;

rule ConstPath ::=
    ModPath  '::' LocalPath
  ;

rule StatePath ::=
     'state' PathSegments
  |  'state'
  ;

rule EventPath ::=
     'event' PathSegments
  |  'event'
  ;

rule PathSegments ::=
     '.' Ident PathSegments
  |  '[' Selector  ']' PathSegments
  |  '[' Selector  ']'
  |  '.' Ident
  ;

rule Selector ::=
    ComplexExprImut  ':' ComplexExprImut
  | ComplexExprImut
  ;

rule Invoke ::=
    FunctionName  '(' InvokeArgs  ')'
  | FunctionName  '('  ')'
  ;
```

```
rule FunctionName ::=
    Ident
  | ModPath  '::' Ident
  ;

rule ModPath ::=
    ModPath  '::' Ident
  | Ident
  ;

rule InvokeArgs ::=
    InvokeArgs_
  ;

rule InvokeArgs_ ::=
    Sep!(InvokeArgs_, ComplexExprImut, ",")
  ;

rule Drop ::=
     'drop'
  ;

rule Emit ::=
     'emit' ComplexExprImut  '=>' StringLiteral
  |  'emit' ComplexExprImut
  |  'emit'  '=>' StringLiteral
  |  'emit'
  ;

rule Let ::=
     'let' Assignment
  ;

rule Assignment ::=
    Path  '=' SimpleExpr
  ;

rule Patch ::=
     'patch' ComplexExprImut  'of' PatchOperations  'end'
  ;

rule PatchOperations ::=
    PatchOperationClause
  | PatchOperations  ';' PatchOperationClause
  ;
```

```
rule PatchField ::=
    StringLiteral
  ;

rule PatchOperationClause ::=
     'insert' PatchField  '=>' ComplexExprImut
  |  'upsert' PatchField  '=>' ComplexExprImut
  |  'update' PatchField  '=>' ComplexExprImut
  |  'erase' PatchField
  |  'move' PatchField  '=>' PatchField
  |  'copy' PatchField  '=>' PatchField
  |  'merge' PatchField  '=>' ComplexExprImut
  |  'merge'  '=>' ComplexExprImut
  |  'default' PatchField  '=>' ComplexExprImut
  |  'default'  '=>' ComplexExprImut
  ;

rule Merge ::=
     'merge' ComplexExprImut  'of' ComplexExprImut  'end'
  ;

rule For ::=
     'for' ComplexExprImut  'of' ForCaseClauses  'end'
  ;

rule ForCaseClauses ::=
    ForCaseClause
  | ForCaseClauses ForCaseClause
  ;

rule ForCaseClause ::=
     'case'  '(' Ident  ',' Ident  ')' WhenClause Effectors
  ;

rule ForImut ::=
     'for' ComplexExprImut  'of' ForCaseClausesImut  'end'
  ;

rule ForCaseClausesImut ::=
    ForCaseClauseImut
  | ForCaseClausesImut ForCaseClauseImut
  ;

rule ForCaseClauseImut ::=
     'case'  '(' Ident  ',' Ident  ')' WhenClause EffectorsImut
  ;
```

```
rule Match ::=
     'match' ComplexExprImut  'of' Predicates  'end'
  ;

rule Predicates ::=
    PredicateClause
  | Predicates PredicateClause
  ;

rule PredicateClause ::=
     'case' CasePattern WhenClause Effectors
  |  'default' Effectors
  ;

rule Effectors ::=
     '=>' Block
  ;

rule Block ::=
    Expr
  | Block  ';' Expr
  ;

rule MatchImut ::=
     'match' ComplexExprImut  'of' PredicatesImut  'end'
  ;

rule PredicatesImut ::=
    PredicateClauseImut
  | PredicatesImut PredicateClauseImut
  ;

rule CasePattern ::=
    RecordPattern
  | ArrayPattern
  | TuplePattern
  | ComplexExprImut
  |  '_'
  |  '~' TestExpr
  | Ident  '=' CasePattern
  ;

rule PredicateClauseImut ::=
     'case' CasePattern WhenClause EffectorsImut
  |  'default' EffectorsImut
```

```
  ;

rule EffectorsImut ::=
     '=>' BlockImut
  ;

rule BlockImut ::=
    ComplexExprImut
  | BlockImut  ',' ComplexExprImut
  ;

rule WhenClause ::=
    (  'when' ComplexExprImut ) ?
  ;

rule PredicateFieldPattern ::=
    Ident  '~=' TestExpr
  | Ident  '=' Ident  '~=' TestExpr
  | Ident  '~=' RecordPattern
  | Ident  '~=' ArrayPattern
  | Ident  '~=' TuplePattern
  |  'present' Ident
  |  'absent' Ident
  | Ident BinCmpEq ComplexExprImut
  ;

rule TestExpr ::=
    Ident TestLiteral
  ;

rule RecordPattern ::=
     '%{' PatternFields  '}'
  |  '%{'  '}'
  ;

rule ArrayPattern ::=
     '%[' ArrayPredicatePatterns  ']'
  |  '%['  ']'
  ;

rule TuplePattern ::=
     '%(' TuplePredicatePatterns OpenTuple  ')'
  |  '%('  ')'
  |  '%('  '.'  '.'  '.'  ')'
  ;
```

```
rule OpenTuple ::=
    (  ','  '.'  '.'  '.' ) ?
  ;

rule TuplePredicatePatterns ::=
    TuplePredicatePatterns  ',' TuplePredicatePattern
  | TuplePredicatePattern
  ;

rule TuplePredicatePattern ::=
    ArrayPredicatePattern
  ;

rule ArrayPredicatePattern ::=
     '~' TestExpr
  |  '_'
  | ComplexExprImut
  | RecordPattern
  ;

rule ArrayPredicatePatterns ::=
    ArrayPredicatePatterns  ',' ArrayPredicatePattern
  | ArrayPredicatePattern
  ;

rule PatternFields ::=
    PatternFields_
  ;

rule PatternFields_ ::=
    Sep!(PatternFields_, PredicateFieldPattern, ",")
  ;

rule Fields ::=
    Fields_
  ;

rule Fields_ ::=
    Sep!(Fields_, Field, ",")
  ;

rule Ident ::=
     '<ident>'
  ;

rule TestLiteral ::=
```

```
      '<extractor>'
  ;

rule BytesLiteral ::=
      '<<'  '>>'
  |  '<<' Bytes  '>>'
  ;

rule Bytes ::=
    BytesPart
  | Bytes  ',' BytesPart
  ;

rule BytesPart ::=
    SimpleExprImut
  | SimpleExprImut  ':'  'int'
  | SimpleExprImut  '/' Ident
  | SimpleExprImut  ':'  'int'  '/' Ident
  ;

macro Sep<L, T, D> ::=
    T D L
  | T D ?
  ;

macro BinOp<Op, Current, Next> ::=
    ( Current ) ( Op ) Next
  ;

rule BinCmpEq ::=
    BinEq
  | BinCmp
  ;

rule BinOr ::=
      'or'
  ;

rule BinXor ::=
      'xor'
  ;

rule BinAnd ::=
      'and'
  ;
```

```
rule BinBitXor ::=
     '^'
  ;

rule BinBitAnd ::=
     '&'
  ;

rule BinEq ::=
     '=='
  |  '!='
  ;

rule BinCmp ::=
     '>='
  |  '>'
  |  '<='
  |  '<'
  ;

rule BinBitShift ::=
     '>>'
  |  '>>>'
  |  '<<'
  ;

rule BinAdd ::=
     '+'
  |  '-'
  ;

rule BinMul ::=
     '*'
  |  '/'
  |  '%'
  ;
```