Apama EPL Reference

5.2.0

August 2014

Document ID: PAM-Apama_EPL_Reference-5.2.0-20140808@233876

Table of Contents

Preface

About this documentation

Apama® Event Processing Language (EPL), which is the new name for Apama MonitorScript, is the native language of the Apama event correlator. You use EPL to write programs that process events in the correlator. *Apama EPL Reference* is a companion to the Apama Studio EPL tutorials and *Developing Apama Applications in EPL*. Use the tutorials and *Developing Apama Applications in EPL* to learn how to write programs in EPL. Use this reference to answer questions and obtain complete details about a particular construct.

Note: Within the product, both EPL and MonitorScript are used and should be treated as synonymous.

Preface

How this book is organized

The information in this book is organized as follows:

Preface

Documentation roadmap

On Windows platforms, the specific set of documentation provided with Apama depends on whether you choose the Developer, Server, or User installation option. On UNIX platforms, only the Server option is available.

Apama provides documentation in three formats:

- HTML viewable in a Web browser
- PDF
- Eclipse Help (if you select the Apama Developer installation option)

On Windows, to access the documentation, select Start > All Programs > Software AG > Apama 5.2 > Apama Documentation . On UNIX, display the `index.html` file, which is in the `doc` directory of your Apama installation directory.

The following table describes the PDF documents that are available when you install the Apama Developer option. A subset of these documents is provided with the Server and User options.

Title	Contents
What's New in Apama	Describes new features and changes since the previous release.
Installing Apama	Instructions for installing the Developer, Server, or User Apama installation options.
Introduction to Apama	Introduction to developing Apama applications, discussions of Apama architecture and concepts, and pointers to sources of information outside the documentation set.
Using Apama Studio	Instructions for using Apama Studio to create and test Apama projects; write, profile, and debug EPL programs; write JMon programs; develop custom blocks; and store, retrieve and playback data.
Developing Apama Applications in Event Modeler	Instructions for using Apama Studio's Event Modeler editor to develop scenarios. Includes information about using standard functions, standard blocks, and blocks generated from scenarios.
Developing Apama Applications in EPL	Introduces Apama's Event Processing Language (EPL) and provides user guide type information for how to write EPL programs. EPL is the native interface to the correlator. This document also provides information for using the standard correlator plug-ins.
Apama EPL Reference	Reference information for EPL: lexical elements, syntax, types, variables, event definitions, expressions, statements.

Title	Contents
Developing Apama Applications in Java	Introduces the Apama in-process API for Java, referred to as JMon, and provides user guide type information for how to write Java programs that run on the correlator. Reference information in Javadoc format is also available.
Building Dashboards	Describes how to create dashboards, which are the end-user interfaces to running scenario instances and data view items.
Dashboard Property Reference	Reference information on the properties of the visualization objects that you can include in your dashboards.
Dashboard Function Reference	Reference information on dashboard functions, which allow you to operate on correlator data before you attach it to visualization objects.
Developing Adapters	Describes how to create adapters, which are components that translate events from non-Apama format to Apama format.
Developing Clients	Describes how to develop C, C++, Java, or .NET clients that can communicate with and interact with the correlator.
Writing Correlator Plug-ins	Describes how to develop formatted libraries of C, C++ or Java functions that can be called from EPL.
Deploying and Managing Apama Applications	Describes how to: • Use the Management & Monitoring console to configure, start, stop, and monitor the correlator and adapters across multiple hosts. • Deploy dashboards over wide area networks, including the internet, and provide dashboards with effective authorization and authentication. • Improve Apama application performance by using multiple correlators, and saving and reusing a snapshot of a correlator's state. • Use the Apama ADBC adapter to store and retrieve data in JDBC, ODBC, and Apama Sim databases. • Use the Apama Web Services Client adapter to invoke Web Services. • Use correlator-integrated messaging for JMS to reliably send and receive JMS messages in Apama applications. • Use Universal Messaging to connect correlators.
Using the Dashboard Viewer	Describes how to view and interact with dashboards that are receiving run-time data from the correlator.

Preface

Contacting customer support

You may open Apama Support Incidents online via the eService section of Empower at http://empower.softwareag.com. If you are new to Empower, send an email to `empower@softwareag.com` with your name, company, and company email address to request an account.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Directory at https://empower.softwareag.com/public_directory.asp and give us a call.

Preface

Chapter 1: Introduction and Notation Conventions

EPL is a flexible and powerful curly-brace, domain-specific, language designed for writing programs that process events.

In EPL, an event is a data object that contains a notification of something that has happened, such as a customer order was shipped, a shipment was delivered, a sensor state change occurred, a stock trade took place, or myriad other things. Each kind of event has an event type name and one or more data elements (called event fields) associated with it. External events are received by one or more adapters, which receive events from an event source and translate them from a source-specific format into Apama's internal canonical format. Derived events can be created as needed by EPL programs.

Hello World example

Though it contains many of the familiar constructs and features found in general-purpose programming languages like Python or Java, EPL also has special features to make it easy to aggregate, filter, correlate, transform, act on, and create events in a concise manner. Here is the canonical "hello world" example written in EPL:

```
monitor HelloWorld
{
   action onload()
   {
      print "Hello world!";
   }
}
```

The Apama event processor, called the correlator, receives events of various types from external sources and routes them to one or more active EPL programs, called monitors. Monitors have registered event handlers, called listeners, for events of particular types with specific combinations of data values or ranges of values. When the correlator detects an event of interest, it calls the appropriate event handlers. If there are no handlers for an event, the correlator discards it or passes it to an event handler specifically for events that have no handler.

Event handlers in EPL are conceptually similar to methods or functions used for handling user-interface events in other languages, such as Java Swing or SWT applications. In EPL, code is executed only in response to events. Except, that is, for the special EPL `onload()`, `ondie()`, and `onunload()` actions. See "Monitor lifecycle" on page 97 for information about these actions.

Introduction and Notation Conventions

Notation conventions

The *Apama EPL Reference* describes the EPL language grammar using a pictorial form of Backus-Naur Form (BNF) notation informally termed *railroad diagrams*. Each diagram is a drawing that represents one rule or *production* of the language grammar. The diagrams show

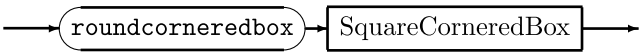
- Lexical rules — how sequences of characters are composed to form the basic elements of the language, called tokens or symbols.
- Syntactic rules — how symbols are used to compose statements and other language constructs such as event definitions.

In addition to the lexical and syntactic rules, both of which have to do with a program's structure, there are semantic rules, which have to do with meaning. For example, the operands of the mulitiplication operator (*) must be of type `decimal`, `float` or `integer` — this is a semantic rule. The semantic rules are described in the text that accompanies the syntax rules.

Each railroad diagram drawing is preceded by a rule name in italic text. The rule can then be referred to by its name in other diagrams.

The railroad diagram drawings consist of sequences and combinations of round and square cornered boxes connected together by "tracks" represented as lines with arrowheads that indicate the direction of travel. For example:

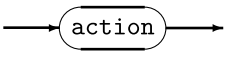
Example



Each diagram represents one language construct or grammar production rule and is read from left to right, following the lines as if you were on a train moving along its track. Branches can occur at various points to allow you to choose amongst several possible alternatives, to skip over optional constructions, and to go back to an earlier point to repeat certain constructions (for example, the comma separated parameter lists in action calls).

Round-cornered boxes represent sequences of characters that form the terminal symbols of the language. Terminal symbols must be taken literally. They are not defined in terms of any other symbol. Terminal symbols are things such as keywords, separators, punctuation, and operators. For convenience, several characters are usually elided into a single box rather than making them separate boxes. For example:

Action



Square-cornered boxes represent named nonterminal symbols or production rules, each of which is further elaborated by a rule in another diagram. At the point where a nonterminal symbol appears, you insert a construct that follows the rules for that symbol and then continue. All nonterminal symbols are defined in terms of a combination of other nonterminal and terminal symbols in a manner such that if all nonterminal symbols were to be replaced by their definitions, the result is a finite sequence consisting only of terminal symbols.

There are three kinds of combinations of terminal and non-terminal symbols: sequence, repetition, and choice.

Introduction and Notation Conventions

Notation for sequences of symbols

A sequence of symbols is denoted by several boxes connected by lines or tracks, as in the diagram below of a `while` loop.

WhileStatement

The following program fragment is constructed according to this rule.

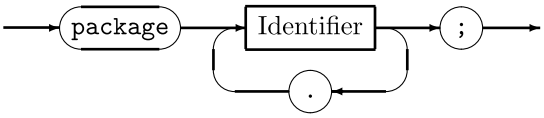
```
while a == b
{
   a:= a-1;
}
```

Introduction and Notation Conventions

Notation for repetition of symbols

Repetition is denoted by a track that loops back to an earlier point in the same diagram, as in the example below, which has multiple names separated by periods, used in the specification of the name of an interface implemented by a class.

PackageSpecification



The following program fragment illustrates how to use this rule.

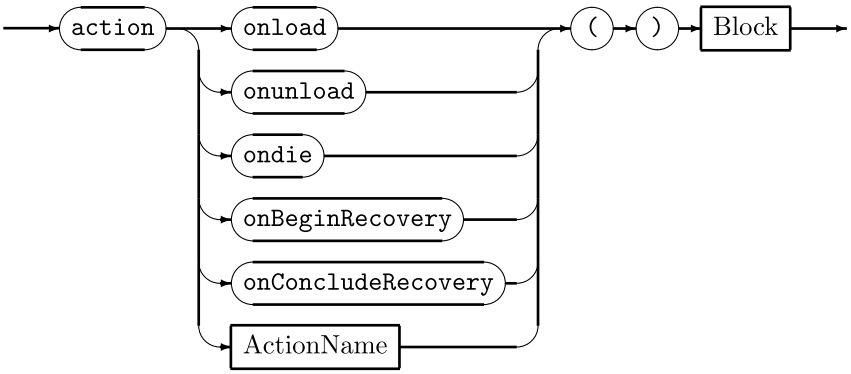
```
package com.apamax.mypackage;
```

Introduction and Notation Conventions

Notation for choices of symbols

A choice among several alternatives is indicated by branches in the tracks, as in the example below where you have several options for specifying a method's characteristics.

SimpleAction

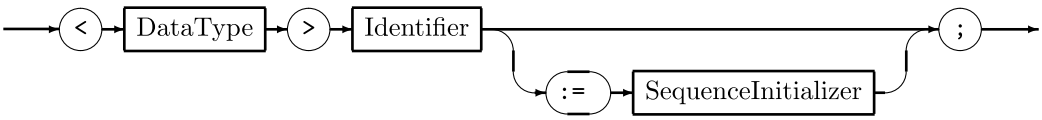


The following program fragments illustrate how to use this rule.

```
action onload()
{
}
action onunload()
{
}
action ondie()
{
}
action myAction()
{
}
```

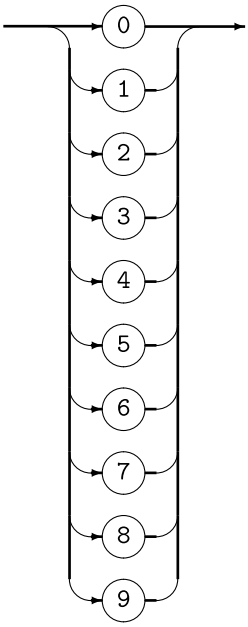
In the preceding example, you had to choose one of the four alternatives. Sometimes the choices are optional and you can choose one of the options or none. In the following diagram, you can optionally include a *SequenceInitializer*.

SequenceDeclaration



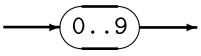
This is indicated by a straight-through track with the optional construct below it. When there are multiple options, they are shown "stacked", as shown in the next diagram.

Digits1



This means you must choose from one of the digits `0` through `9`. For the sake of brevity, the notation `..` is often used to indicate a sequence of consecutive characters from which you can choose one. Thus the following diagram means exactly the same thing as the longer form of *Digits* shown in the previous diagram.

Digits2



Introduction and Notation Conventions

Chapter 2: Types

EPL has primitive types and reference types. Data in the primitive types are simple scalar values. Reference types (also called complex types or object types) have values that are more complicated and some, like the `dictionary` type, have multiple values and have definitions that involve more than one type.

When values are passed as parameters in action and method invocations, primitive types are passed by value, and reference types are passed by reference. When a parameter is passed by value, the called action or method receives a copy of the value and has no direct way to change the variable that the value may have been derived from. When a parameter is passed by reference, the called action or method receives a reference instead of a copy and if the called action changes the value, the caller also sees the change.

Note that there is no type equivalent to a memory address or pointer.

DataTypeName

EPL supports the following types:

- Primitive types:
  - "boolean" on page 18
  - "decimal" on page 20
  - "float" on page 21
  - "integer" on page 27

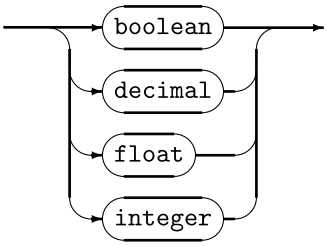
- "string" on page 30 — Technically, a string is a reference type but it behaves like a primitive type.
- Reference types:
  - "action" on page 39
  - "chunk" on page 41
  - "context" on page 43
  - "dictionary" on page 44
  - "event" on page 49
  - "listener" on page 55
  - "location" on page 55
  - "sequence" on page 57
  - "stream" on page 63

The `dictionary` and `sequence` reference types are also container types.

Primitive and string types

Apama supports these primitive types: `boolean`, `decimal`, `float`, and `integer`. Technically, a `string` is a reference type. However, because strings are immutable, it behaves more like a primitive type than a reference type. Consequently, `string` appears in the ReferenceDataTypeName diagram, but it is discussed with the other primitive types.

PrimitiveDataTypeName



- "boolean" on page 18
- "decimal" on page 20
- "float" on page 21
- "integer" on page 27
- "string" on page 30

Types

boolean

The boolean type has two possible values: `true` or `false`.

Operators

The table below lists the EPL operators that you can use with Boolean values.

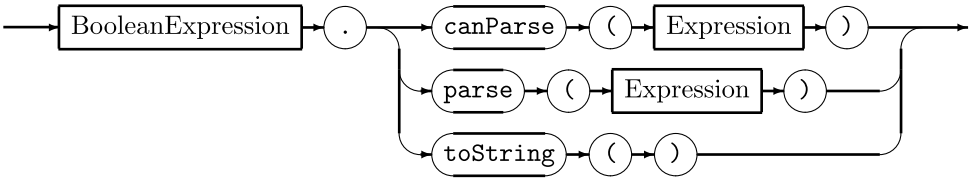
Operator	Description	Result Type
`=`	Equal comparison	`boolean`
`!=`	Not equal comparison	`boolean`
`or`	Boolean (logical) or	`boolean`
`and`	Boolean (logical) and	`boolean`
`xor`	Boolean (logical) exclusive or	`boolean`
`not`	Boolean (logical) inverse	`boolean`

False sorts before true.

Methods

The following methods may be called on variables of `boolean` type:

BooleanMethods



- `canParse()` — returns `true` if the string argument can be successfully parsed.
- `parse()` — method that returns the `boolean` instance represented by the `string` argument. You can call this method on the `boolean` type or on an instance of a `boolean` type. The more typical use is to call `parse()` directly on the `boolean` type.

  The `parse()` method takes a single string as its argument. This string must be the string form of a `boolean` object. The string must adhere to the format described in *Deploying and Managing Apama Applications*, "Event file format". For example:

  ```
  boolean a;
  a := boolean.parse("true");
  ```

  You can specify the `parse()` method after an expression or type name. If the correlator is unable to parse the string, it is a runtime error and the monitor instance that the EPL is running in terminates.
- `toString()` – returns a `string` representation of the `boolean`. The return value is "`true`" if the referenced Boolean's value is `true`. The return value is "`false`" if the referenced Boolean's value is `false`.

Primitive and string types

decimal

A signed decimal floating point number. Either a decimal point (.) or an exponent symbol (e) must be present within the number for it to be a valid `decimal`, plus a decimal suffix (`d`) to distinguish it from a `float`.

When perfect accuracy of base-10 numbers is a requirement, use the `decimal` type in place of the `float` type. When extremely small floating point variations are acceptable, you might choose to use the `float` type to obtain better performance.

For information about Apama client API support for the `decimal` type, see *What's New in Apama*.

Values

Values of the `decimal` type are a finite-precision approximation of the mathematical real numbers, encoded as 64-bit decimal floating-point values consisting of sign, significand, and exponent, as defined by the "IEEE Standard for Floating-Point Arithmetic, ANSI/IEEE Standard 754 -2008 (IEEE, New York)". Values of the `decimal` type have a precision of exactly 16 decimal digits.

The largest positive decimal floating point value that can be stored in a variable of type `decimal` is

$9.999999999999999 * 10^{384}$

The smallest nonzero positive value that can be stored is

10^{-398}

In addition to the usual positive and negative numbers, the IEEE standard also defines positive and negative zeros, positive and negative infinities, and Not-a-Number values.

Because decimal values are of finite precision, they cannot accurately represent all values, for example, recurring decimals or irrational numbers. However, decimals have the advantage over floats in that provided a decimal literal does not exceed the 16-place precision, it will be represented exactly within the correlator. The following program illustrates the difference between `decimal` and `float` types in this regard:

```
monitor foo
{
   action onload()
   {
      float f;
      decimal d;
      f := 0.1;
      d := 0.1d;
      print f.formatFixed(18);
      print d.formatFixed(18);
   }
}
```

This program produces the output below.. Note the small error in the least significant digit in the `float`, versus the `decimal`.

```
0.100000000000000006
0.100000000000000000
```

There are a number of `decimal` constants provided in EPL. See "Support for IEEE 754 special values" on page 75.

Operators

The EPL operators that you can use with `decimal` types are the same operators that you can use with `float` types. See `float` "Operators" in the next section of the documentation.

Methods

The methods that you can call on `decimal` types are the same methods that you can call on `float` types. See `float` "Methods" in the next section of this documentation. There are a few differences according to whether the method is called on a `decimal` or `float` type and these are noted in the descriptions.

Primitive and string types

float

A signed floating point number. Either a decimal point (`.`) or an exponent symbol (`e`) must be present within the number for it to be a valid `float`.

When perfect accuracy is a requirement, use the `decimal` type in place of the `float` type. When extremely small floating point variations are acceptable, you might choose to use the `float` type to obtain better performance.

Values

Values of the `float` type are a finite-precision approximation of the mathematical real numbers, encoded as 64-bit binary floating-point values consisting of sign, significand, and exponent, as defined by the “IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754 -1985 (IEEE, New York)”. Values of the `float` type have a precision of approximately 16 decimal digits. (The binary significand is 52 bits wide.)

The largest positive floating point value that can be stored in a variable of type `float` is $1.7976931348623157 * 10^{308}$ and the smallest nonzero positive value that can be stored is $2.2250738585072014 * 10^{-308}$

In addition to the usual positive and negative numbers, the IEEE standard also defines positive and negative zeros, positive and negative infinities, and Not-a-Number values. For information about how the correlator handles these values, see "Support for IEEE 754 special values" on page 75.

Because float values are of finite precision and binary encoded, they cannot accurately represent all values. In particular, when a floating point literal expressed in decimal notation is converted to its binary floating-point representation, there can be a slight loss of accuracy. This occurs because most decimal fractions cannot be represented precisely in binary. So the fraction `0.1` or `1/10` in base 10 becomes the infinitely repeating fraction `0.0001100110011001100110011...` when it is converted to base 2. Similarly, conversions from floating point values to integral or string types will sometimes be inexact. The following program illustrates the effects of finite precision and conversions between base 10 and base 2:

```
monitor foo
{
   action onload()
   {
      float f;
      f := 0.1;
      print f.formatFixed(18);
   }
```

```
}
```

This program produces the output `0.10000000000000006`. Note the small error in the least significant digit.

There are a number of `float` constants provided in EPL. See "Support for IEEE 754 special values" on page 75.

Operators

The following table lists the EPL operators available for use with floating point values, that is `decimal` or `float` types.

Operator	Description	Result Type
`<`	Less-than comparison	`boolean`
`<=`	Less-than or equal comparison	`boolean`
`=`	Equal comparison	`boolean`
`!=`	Not equal comparison	`boolean`
`>=`	Greater-than or equal comparison	`boolean`
`>`	Greater-than comparison	`boolean`
`+`	Unary floating point identity	`decimal` or `float`
`-`	Unary floating point additive inverse	`decimal` or `float`
`+`	Floating point addition	`decimal` or `float`
`-`	Floating point subtraction	`decimal` or `float`
`*`	Floating point multiplication	`decimal` or `float`
`/`	Floating point division	`decimal` or `float`

Overflows and underflows are ignored by the EPL runtime.

The correlator compares floating point values as follows:

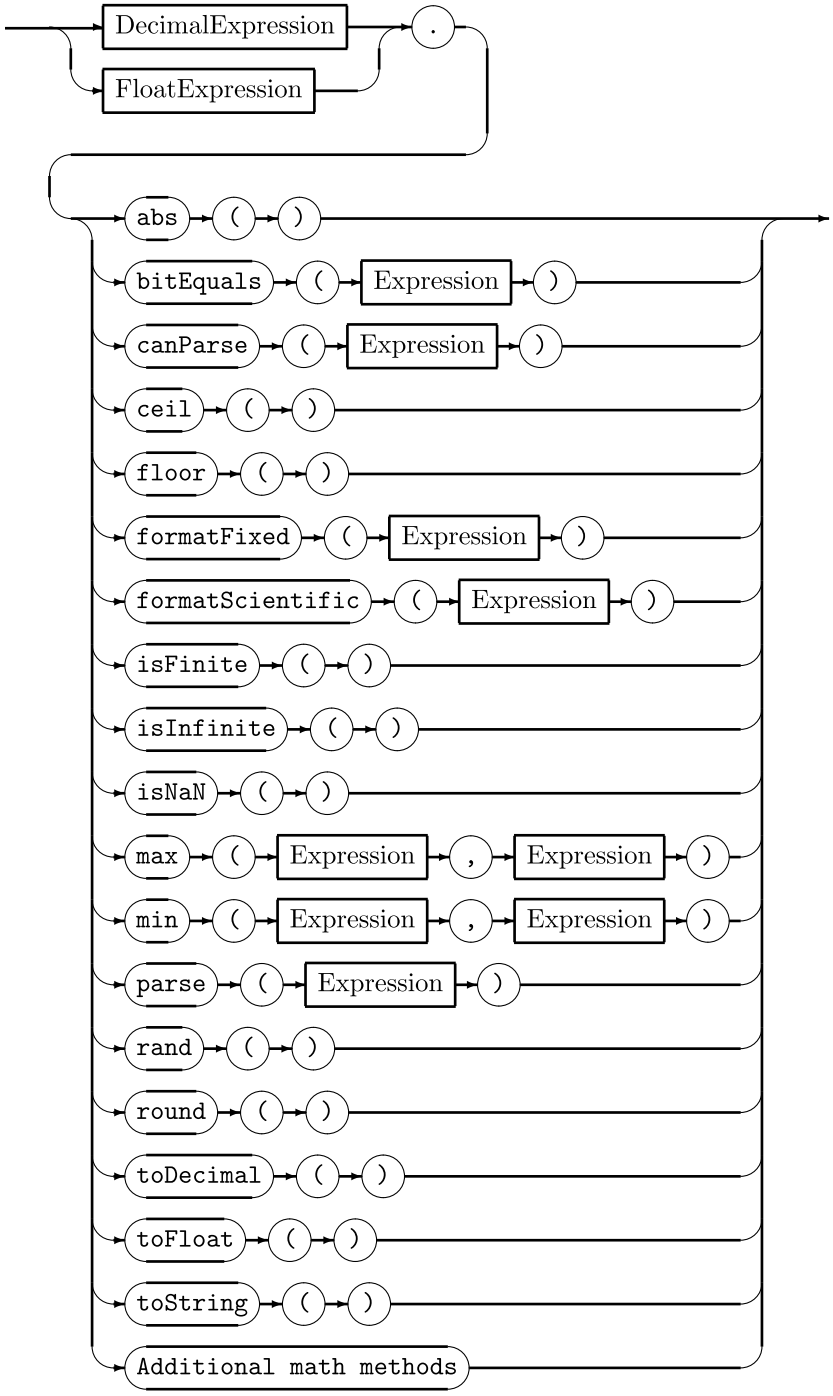
- Finite `float` and `decimal` types compare in the obvious way.
- `-Inf` is equal to `-Inf` and is less than any finite number or `+Inf`.
- `+Inf` is equal to `+Inf` and is greater than any finite number or `-Inf`.
- `NaN` is not equal to anything, including another `NaN`.
- If you try to use `NaN` for keying or sorting the correlator terminates the monitor instance.

Methods

The following inbuilt methods may be called on variables of `decimal` or `float` type. Unless noted otherwise, if you call a method on a `decimal` type the return value is a `decimal`, and if you call the

method on a `float` type, the return value is a `float`. In all method descriptions, *x* represents the value that the method is called on.

FloatMethods and *DecimalMethods*



- `abs()` — returns `|x|`, the absolute value of *x*.
- `acos()` — returns the inverse cosine of *x* in radians. Special case: *x*`.acos()` = `NaN`, if `|x| > 1`.
- `acosh()` — returns the inverse hyperbolic cosine of *x*. Special case: *x*`.acosh()` = `NaN`, if *x* `< 1`.
- `asin()` — returns the inverse sine of *x* in radians. Special cases:

  - `(NaN).asin() = NaN`
  - `x.asin() = NaN`, if `|x| > 1`
- `asinh()` — returns the inverse hyperbolic sine of `x`.
- `atan()` — returns the inverse tangent of `x`.
- `atan2(y)` — returns the two-parameter inverse tangent of `x` and `y`. Special cases:
  - `(anything).atan2(NaN) = NaN`
  - `(NaN).atan2(anything) = NaN`
  - `(±0).atan2(anything except NaN) = ±0`
  - `(±0).atan2(-anything except NaN) = ±pi`
  - `(anything except 0 and NaN).atan2(0) = ±pi/2`
  - `(anything except ±Infinity and NaN).atan2(+Infinity) = ±0`
  - `(anything except ±Infinity and NaN).atan2(-Infinity) = ±pi`
  - `(±Infinity).atan2(+Infinity) = ±pi/4`
  - `(±Infinity).atan2(-Infinity ) = ±3pi/4`
  - `(±Infinity).atan2(anything except 0, NaN and ±Infinity) = ±pi/2`
- `atanh()` — returns the inverse hyperbolic tangent of `x`. Special cases:
  - `x.atanh() = NaN`, if `|x| > 1`
  - `(NaN).atanh() = NaN`
  - `(±1).atanh() = ±Infinity`
- `bitEquals(decimal)` or `bitEquals(float)` — returns true if the value it is called on and the value passed as an argument to the method are the same. The value the method is called on and the argument to the method must both be `decimal` types or must both be `float` types. The method performs a bitwise comparison. This is useful because `bitEquals()` returns true for `NaN.bitEquals(NaN)` for `NaN`s that are bitwise identical whereas `NaN = NaN` is always false even if the `NaN`s have identical representations.
- `canParse(string)` — returns `true` if the string argument can be successfully parsed.
- `cbrt()` — returns the cube root of `x`.
- `ceil()` — returns the smallest possible integer that is greater than or equal to the value the method is called on. Special cases:
  - `(+Infinity).ceil() = integer.MAX`
  - `(-Infinity).ceil() = integer.MIN`
  - `(NaN).ceil()` causes a runtime error; the correlator terminates the monitor
- `cos()` — returns the cosine of `x`. See also the note at the end of this list.
- `cosh()` — returns the hyperbolic cosine of `x`. Special case: `(±Infinity or NaN).cosh() = |x|`
- `erf()` — returns the error function of `x`. The formula is as follows:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

- `exp()` — returns `e` to the power *x* or `e`*x*, where *x* is the value of the `decimal` or `float` and where `e` is approximately 2.71828183. Special cases:
  - `exp(NaN) = NaN`
  - `exp(+Infinity) = +Infinity`
  - `exp(-Infinity) = 0`
- `exponent()` — When called on a `float` value, this method returns the integer that is the exponent where *x* = `mantissa*2`$^{\texttt{exponent}}$ assuming `0.5 <= |mantissa| < 1.0`. When called on a `decimal` value, this method returns the exponent where *x* `= mantissa*10`$^{\texttt{exponent}}$ assuming `0.1 <= |mantissa| < 1.0`. Special cases:
  - `(0.0).exponent() = 0`
  - `(±Infinity or NaN).exponent()` terminates the monitor instance that contains the method call.
- `floor()` — returns the largest possible integer that is less than or equal to the value the method is called on. Special cases:
  - `(+Infinity).floor() = integer.MAX`
  - `(-Infinity).floor() = integer.MIN`
  - `(NaN).floor()` causes a runtime error; the correlator terminates the monitor.
- `fmod(`*y*`)` — returns *x* `mod` *y* in exact arithmetic.
- `formatFixed(`*integer*`)` — returns a string representation of the value the method is called on where the value is rounded to the number of decimal places specified in the argument. This method can operate on the IEEE special values.
- `formatScientific(`*integer*`)` — returns a string representation of the value the method is called on where the value is truncated to the number of significant figures specified in the argument and formatted in Scientific Notation. This method can operate on the IEEE special values.
- `fractionalPart()` — returns the fractional component of *x*.
- `gammal()` — returns the logarithm of the gamma function.
- `ilogb()` — returns an `integer` that is the binary exponent of non-zero *x*. Special case: throws exception for `ilogb(NaN)`.
- `integralPart()` — returns an `integer` that is the integral part of a floating point value. Similar to `floor()`, which rounds down, and `ceil()`, which rounds up. `integralPart()` rounds towards zero. Special case: throws exception for `integralPart(NaN)`.
- `isFinite()` — returns true if and only if the value it is called on is not `±Infinity` or `NaN`.
- `isInfinite()` — returns true if and only if the value it is called on is `±Infinity`.
- `isNaN()` — returns true if and only if the value it is called on is `NaN`.
- `ln()` — returns the natural log of the value the method is called on. Special cases:
  - `(0).ln() = -Infinity`

  - `(-anything).ln() = NaN`

- `log10()` — returns the log to base 10 of the value the method is called on. Special cases:
  - `(0).log10() = -Infinity`
  - `(-anything).log10() = NaN`
- `mantissa()` — When called on a `float` value, this method returns a mantissa where $x = mantissa*2^{exponent}$ assuming that `0.5 <= |mantissa| < 1.0`. When called on a `decimal` value, this method returns a mantissa where $x = mantissa*10^{exponent}$ assuming that `0.1 <= |mantissa| < 1.0`. Special cases:
  - `(0.0).mantissa() = 0.0`
  - `(Infinity or NaN).mantissa()` terminates the monitor instance that contains the method call
- `max(decimal, decimal)` or `max(float, float)` — returns the value of the larger operand. You can call this method on the `decimal` or `float` type or on an instance of a `decimal` or `float` type.
- `min(decimal, decimal)` or `min(float, float)` — returns the value of the smaller operand. You can call this method on the `decimal` or `float` type or on an instance of a `decimal` or `float` type.
- `nextafter(y)` — returns the next distinct floating-point number after x that is representable in the underlying type in the direction toward y.
- `parse(string)` – method that returns the `decimal` or `float` instance represented by the `string` argument. You can call this method on the `decimal` or `float` type or on an instance of a `decimal` or `float` type. The more typical use is to call `parse()` directly on a `decimal` or `float` type.

  The `parse()` method takes a single string as its argument. This string must be the string form of an `event` object. The string must adhere to the format described in *Deploying and Managing Apama Applications*, "Event file format". For example:

  ```
  float a;
  a := float.parse("123.456");
  ```

  You can specify the `parse()` method after an expression or type name. If the correlator is unable to parse the string, it is a runtime error and the monitor instance that the EPL is running in terminates.

  A call to `decimal.parse()` can include or exclude the appended `d`. In other words, `decimal.parse("1.0")` and `decimal.parse("1.0d")` both work.

  The `parse()` method can operate on the string form of the IEEE special values.
- `pow(decimal)` or `pow(float)` — returns x to the power y (where y is the argument) or x^y. See also "Special cases of pow()" on page 77.
- `rand()` — returns a random value from 0.0 up to (but not including) the value the method was invoked on. If the value was negative, then the random value will be from the value (but not including it) up to 0.0. When you are calling the `rand()` method on a variable, the method behaves correctly if the variable value is zero, for example, `(0.0).rand()` returns `0.0`.

  Special case: `(±Infinity or NaN).rand()` causes a runtime error; the correlator terminates the monitor.
- `round()` – rounds to the nearest integer. Uses banker's rounding, which means the round-to-even method, to break ties. For example, it rounds both `3.5` and `4.5` to `4`. Special cases:

  - `(+Infinity).round() = integer.MAX`
  - `(-Infinity).round() = integer.MIN`
  - `(NaN).round()` causes a runtime error; the correlator terminates the monitor.
- `scalbn(n)` — When called on a `float` value, this method returns $x*2^n$, where `n` is of `integer` type. When called on a `decimal` value, this method returns $x*10^n$, where `n` is of `integer` type.
- `sin()` — returns the sine of `x`. See also the note at the end of this list.
- `sinh()` — returns the hyperbolic sine of `x`. Special case: `(±Infinity or NaN).sinh() = |x|`
- `sqrt()` — returns the positive square root of the value it is called on. Special cases:
  - `(-anything).sqrt() = NaN`
  - `(+Infinity).sqrt() = +Infinity`
- `tan()` — returns the tangent of `x`. See also the note at the end of this list.
- `tanh()` — returns the hyperbolic tangent of `x`. Special case: `NaN.tanh() = NaN`
- `toDecimal()` — returns a `decimal` representation of the `float`. This method can operate on the IEEE special values.
- `toFloat()` — returns a `float` representation of the `decimal`. This method can operate on the IEEE special values.
- `toString()` — returns a `string` representation of the `float` or `decimal` it is called on. This method can operate on the IEEE special values. A call to `decimal.toString()` does not include a `d` suffix.

Note: Let `trig` be any of `sin`, `cos`, or `tan`. The argument to these functions is in units of Radian. Also `(±Infinity or NaN).trig() = NaN`.

Primitive and string types

integer

Values of the `integer` type are negative, zero, and positive integers encoded as 64-bit signed two's complement binary integers. The lowest negative value that can be stored in a variable of type integer is -9223372036854775808 or (-2^{63}) and the highest positive value that can be stored is 9223372036854775807 or ($2^{63} - 1$).

There are a few `integer` constants provided in EPL. See "Support for IEEE 754 special values" on page 75.

Operators

The following table describes the EPL operators available for use with `integer` values.

Operator	Description	Result Type
<	Less-than comparison	`boolean`

Operator	Description	Result Type
`<=`	Less-than or equal comparison	`boolean`
`=`	Equal comparison	`boolean`
`!=`	Not equal comparison	`boolean`
`>=`	Greater-than or equal comparison	`boolean`
`>`	Greater-than comparison	`boolean`
`+`	Unary integral identity	`integer`
`-`	Unary integral additive inverse	`integer`
`+`	Integral addition	`integer`
`-`	Integral subtraction	`integer`
`*`	Integral multiplication	`integer`
`/`	Integral division	`integer`
`%`	Integral remainder	`integer`
`or`	Bitwise or	`integer`
`and`	Bitwise and	`integer`
`xor`	Bitwise exclusive or	`integer`
`not`	Unary bitwise inverse	`integer`
`>>`	Bitwise shift right	`integer`
`<<`	Bitwise shift left	`integer`

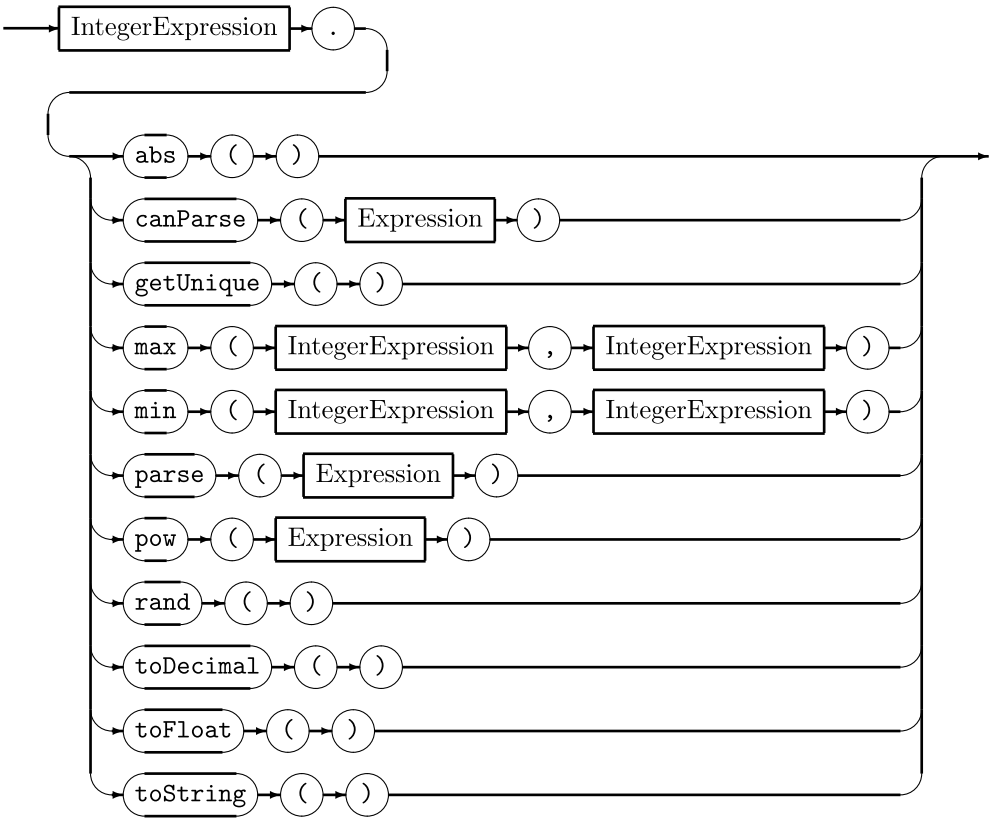
An attempt to divide by zero (0) or to compute a remainder of zero raises an error. Overflows and underflows in arithmetic are ignored by the EPL runtime.

When you use the shift operators, the sign of a result value can differ from that of the operand value being shifted. When you use `not` the sign of the result value will be the opposite of that of its operand.

Methods

The following methods may be called on variables of `integer` type:

IntegerMethods



- `abs()` – returns as an `integer` the absolute value of i or |i|, where i is the value of the integer.
- `canParse()` — returns `true` if the string argument can be successfully parsed.
- `getUnique()` – method that generates a unique integer in the scope of the correlator. This is a type method as well as an instance method. It returns an `integer` that is unique for the correlator session's lifetime. When the correlator is shut down and restarted, then the integers returned might be the same as some or all of the values produced in the earlier session.

  When correlator persistence is enabled the state of this method is preserved across shutdown and recovery. In other words, as long as you use the same recovery datastore, it does not matter how many times you restart the correlator. The result of invoking `getUnique()` will always be a unique number across all restarts.

  This method starts by generating 0, 1, 2, 3, and so on. However, you cannot assume that you will receive the integer you might expect. The returned numbers are 64-bit signed integers.

  For example, the following statement prints a different number every time the correlator executes it:

  ```
  print integer.getUnique().toString();
  ```

  Following are more examples:

  ```
  monitor M {
     action onload() {
        integer i;
        i := integer.getUnique(); // called on type
        i := i.getUnique();       // called on instance
      }
  }
  ```

- `max(integer, integer)` – returns as an `integer` the value of the larger operand. You can call this method on the `integer` type or on an instance of an `integer` type.
- `min(integer, integer)` – returns as an `integer` the value of the smaller operand. You can call this method on the `integer` type or on an instance of an `integer` type.
- `parse()` – method that returns the `integer` instance represented by the `string` argument. You can call this method on the `integer` type or on an instance of an `integer` type. The more typical use is to call `parse()` directly on the `integer` type.

  The `parse()` method takes a single string as its argument. This string must be the string form of an `integer` object. The string must adhere to the format described in *Deploying and Managing Apama Applications*, "Event file format". For example:

  ```
  integer a;
  a := integer.parse("20080116");
  ```

  You can specify the `parse()` method after an expression or type name. If the correlator is unable to parse the string, it is a runtime error and the monitor instance that the EPL is running in terminates.
- `pow(integer)` – returns as an `integer` the value of the operand to the power *x* (where *x* is the argument) or i^x, where *i* is the value of the operand. Note that negative values of *x* are not allowed, as these would generate floating point results.
- `rand()` – returns a random `integer` value from `0` up to (but not including) the value of the variable the method was invoked on. The following snippet of code would set `B` to a random value from `0` to `19`:

  ```
  integer A;
  integer B;
  A := 20;
  B := A.rand();
  ```

  while the next snippet would set `B` to a random value from `-14` and `0`:

  ```
  integer A;
  integer B;
  A := -15;
  B := A.rand();
  ```

  When you are calling the `rand()` method on a variable, the method behaves correctly if the variable value is zero, that is `(0).rand()` returns `0`.
- `toDecimal()` – returns a `decimal` representation of the `integer`.
- `toFloat()` – returns a `float` representation of the `integer`.
- `toString()` – returns a `string` representation of the `integer`.

Primitive and string types

string

A text string.

Usage

Enclose string literals in double quotes. Values of the `string` type are sequences of non-null Unicode characters encoded in UTF-8 format. Note that UTF-8 is a variable-width encoding and a character can occupy from 1 to 4 bytes of storage. The characters in the 7-bit ASCII character set are a subset of UTF-8 and occupy a single byte each.

Although `string` types are discussed as though they are primitive types, they are actually reference types. However, EPL's `string` objects are immutable. For example, a statement such as `s:=s+" suffix";` creates a new string object and changes the variable `s` to refer to that new string object. Any other references to the old value continue to point to the old value.

Operations that can return a different string value, such as concatenation, case folding, or trimming white space, always create new strings rather than modifying the existing value in place. The previous value's storage is recovered later by the EPL runtime garbage collector.

The length of a string is limited by the memory available at runtime, which can be multiple gigabytes. In practice, you are unlikely to exceed the limit in a single string. (The total address space available to the EPL runtime system is limited to roughly four gigabytes when running on a 32-bit system.)

Use the `\` to enter special characters in string literals:

To enter this...	Insert this...
`"` (double quote)	`\"`
`\` (backslash)	`\\`
newline character	`\n`
tab character	`\t`

Operators

The table below lists the EPL operators available for use with string values.

Operator	Description	Result Type
`<`	Less-than string comparison	`boolean`
`<=`	Less-than or equal string comparison	`boolean`
`=`	Equal string comparison	`boolean`
`!=`	Not equal string comparison	`boolean`
`>=`	Greater-than or equal string comparison	`boolean`
`>`	Greater-than string comparison	`boolean`
`+`	String concatenation	`string`

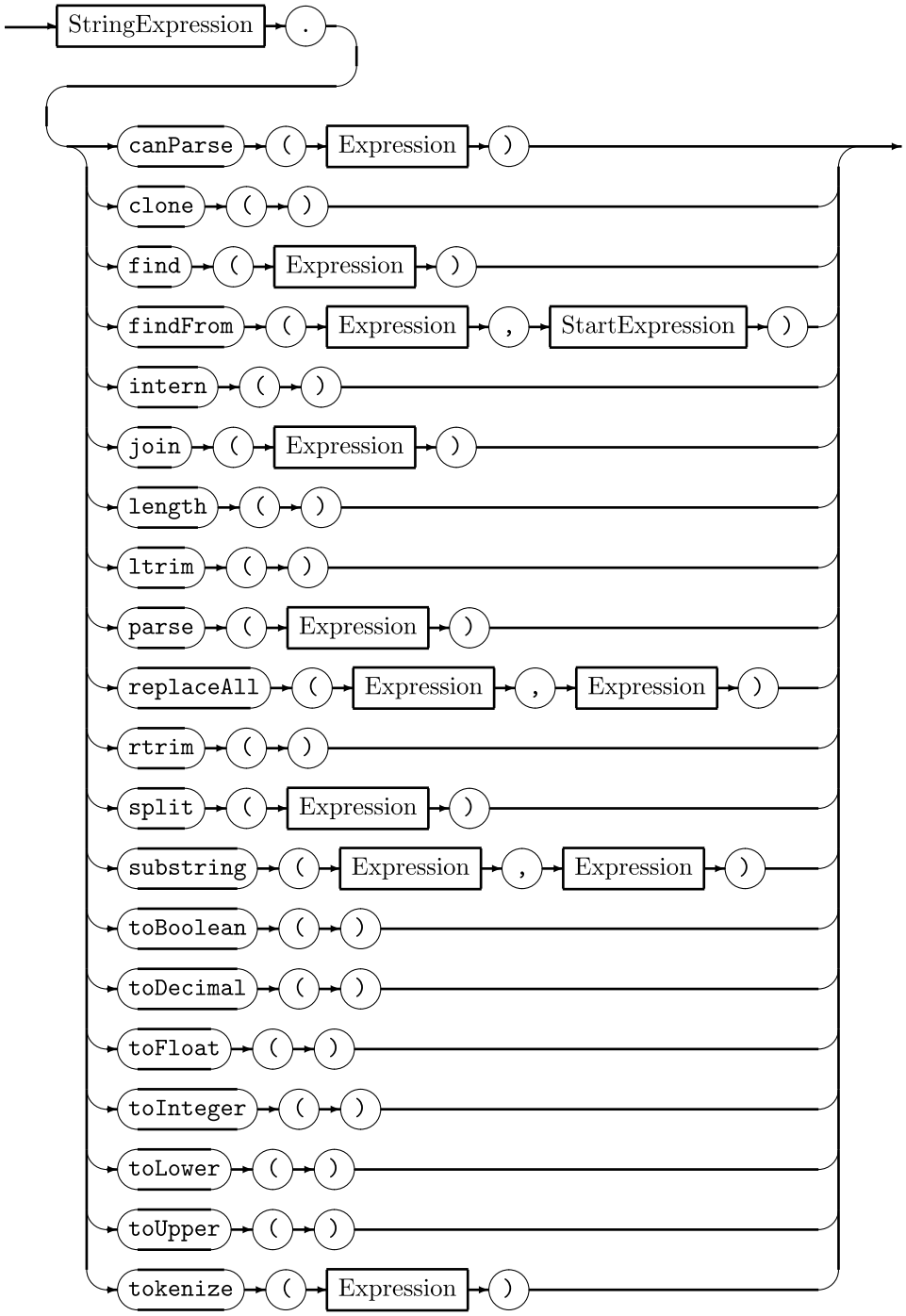
When you compare two strings for equality, the result is `true` if the strings are the same length and each character in one string is identical to the corresponding character at the same position in the other string.

When you compare two strings for less than or greater than, the characters in the strings are compared pairwise according to the numerical values of their Unicode code points. The comparison is case-sensitive so capital letters are not equal to their lower case equivalents. Characters earlier in the character set sort before characters later in the character set. To order two unequal strings, the earliest difference is considered. For example, `"abcXdef"` sorts earlier than `"abcYdef"`, `"abc"` sorts earlier than `"abcXYZ"`; the empty string sorts earliest of all.

Methods

The following methods may be called on values of `string` type:

StringMethods



- `canParse()` — returns `true` if the string argument can be successfully parsed.
- `clone(`*`string`*`)` — returns a reference to the specified string. When called on a `string`, the `clone()` method does not make a copy of the string since strings are immutable.
- `find(`*`substring`*`)` — returns an `integer` indicating the index position of the substring passed as parameter to the method. If the string parameter does not exist as a substring within the string, the method returns `-1`. Note that in EPL string indices (the position of a character within the string) count upwards from `0`.

- `findFrom(substring, fromIndex)` — behaves like the `find()` method, but starts searching for the specified substring with the character indicated by `fromIndex`. For example, if the value of `fromIndex` is `7`, the search begins with the character that has an index of `7`.
- `intern()` — marks the string it is called on as interned. Subsequent incoming events that contain a string that is identical to an interned string use the same string object. The `intern()` method takes no arguments and returns the interned version of the string it is called on. For example:

  ```
  print "hello world";
  print "hello world".intern();
  ```

  Both statements print:

  ```
  hello world
  ```

  The benefit of using the `intern()` method is that it reduces the amount of memory used and the amount of work the garbage collector must do. A disadvantage is that you cannot free memory used for an interned string.

  If there are a limited number of strings that will be used many times then calling `intern()` on these strings speeds the handling of events that use them. You might want to call `intern()` on the names of products or stock symbols, which are all used frequently. For example, invoking `"APMA".intern()` might make sense if you are expecting a large number of incoming events of the form `Tick("APMA", ...)`. You would not want to call `intern()` on order IDs, because there are so many and each one is likely to be unique.

  Calling `intern()` on a string is a global operation. That is, all contexts can then use the same string object. Any strings already in use by the correlator are not affected, even if they match the string `intern()` is called on.

  If you use correlator persistence, details of which strings have been interned are not stored in the recovery datastore. If the correlator shuts down and restarts, you must call `intern()` again on the pertinent strings.

- `join(sequence<string> s )` — concatenates the strings in `s` using the string it is called on as a separator. The single parameter must be a `sequence` type that contains strings. You cannot specify a variable number of `string` parameters. For example:

  ```
  sequence<string> s :=
     ["Something", "Completely", "Different"];
  print ", ".join(s);
  ```

  This prints the following:

  ```
  Something, Completely, Different
  ```

- `length()` — returns an `integer` indicating the length of the string.
- `ltrim()` — returns a string where all white space characters at the beginning have been removed. White space characters are space, new line and tab characters.
- `parse()` — method that returns the `string` value represented by the `string` argument without enclosing that value in quotation marks. You can call this method on the `string` type or on an instance of a `string` type. The more typical use is to call `parse()` directly on the `string` type.

  The `parse()` method takes a single string as its argument. The string must adhere to the format described in *Deploying and Managing Apama Applications*, "Event file format".

  Use the following format to specify the string you want to parse:

  ```
  "your_string_with_escape_characters"
  ```

Use a backslash to escape each quotation mark or backslash in your string, including quotation marks that enclose your string. For example, to parse `"Hello World"`, specify it as "\"Hello World \"". In other words, if you are writing literal strings in EPL, you must precede all backslashes and quotation marks with a backslash. For example:

```
string a := "\".\\\\.\"";
string b := string.parse(a);
print a;
print b;
```

This prints the following:

```
".\\."
.\.
```

The `string.parse()` method is useful when you have a string that contains backslash escape characters and you want to obtain a string without them.

More examples:

```
string a := string.parse("\"Hello World\"");
string b := string.parse("\"\\\"\"");
print a;
print b;
```

This prints the following:

```
Hello World
"
```

You can specify the `parse()` method after an expression or type name. If the correlator is unable to parse the string, it is a runtime error and the monitor instance that the EPL is running in terminates. For example, the following is an error and causes the correlator to terminate:

```
a := string.parse("Hello World");
```

The `parse()` method cannot parse the result of a `toString()` method. This is because the `toString()` method does not enclose its result in quotation marks, nor does it escape any special characters. For example, the following is false:

```
x = string.parse(x.toString())
```

If a string contains no special characters (for example, " or \) then the following equality does hold true:

```
x = string.parse("\""+x.toString()+"\"")
```

- `replaceAll(`*`string`*`, `*`string`*`)` — takes two `string` arguments, *`string1`* and *`string2`*. For the `string` the method is called on, the `replaceAll()` method makes a copy of that string, replaces instances of *`string1`* with instances of *`string2`* and returns the revised string. For example:

  ```
  string x := "XYZ";
  print x.replaceAll("Y","y");
  print x;
  ```

  This prints the following:

  ```
  XyZ
  XYZ
  ```

  Notice that `x` itself is unchanged. If *`string1`* is an empty string then the monitor instance dies. If instances of *`string1`* overlap then the method replaces only the first instance in the overlapping instances.

- `rtrim()` — returns a string where all *whitespace* characters at the end have been removed. Whitespace characters are space, new line and tab characters.

- `split(string)` — returns a sequence of strings that represent the string argument split at occurrences of the string that the method is called on. The returned sequence always contains at least one string. The `split()` method is useful for separating a a string that contains newline characters into individual lines or for dividing comma-separated values in a single string into multiple strings. For example:

Method Call	Returned Sequence
`",".split("x,y,z")`	`["x","y","z"]`
`",".split("")`	`[""]`
`",".split(",x,,y")`	`["","x","","y"]`

- `substring(integer, integer)` — returns the substring indicated by the `integer` parameters. The parameters indicate the position of the first and last characters of the substring, the first being inclusive, while the second is exclusive. If a parameter is a positive value it is taken to be the position of a character going from left to right counting upwards from `0`. If a parameter is a negative value it is taken to be the position of a character going from right to left counting downwards from `-1`. Therefore if

```
string s;
s := "goodbye";
```

then

```
s.substring(0, 0) is ""
s.substring(0, 2) is "go"
s.substring(2, 4) is "od"
s.substring(0, 7) is "goodbye"
s.substring(0, -1) is "goodby"
s.substring(-4, -1) is "dby"
s.substring(-7, -1) is "goodby"
s.substring(-7, 7) is "goodbye"
```

- `toBoolean()` — returns true if the string is `"true"` and false in all other cases. This method is case sensitive.
- `toDecimal()` — returns a `decimal` representation of the string, if the string starts with one or more numeric characters. The numeric characters can optionally have amongst them a decimal point or mantissa symbol. Returns `0.0` if there are no such characters.
- `toFloat()` — returns a `float` representation of the string, if the string starts with one or more numeric characters. The numeric characters can optionally have amongst them a decimal point or mantissa symbol. Returns `0.0` if there are no such characters.
- `toInteger()` — returns an `integer` representation of the string, if the string starts with one or more numeric characters. Returns `0` if there are no such characters.
- `tokenize(string)` — the format for invoking this method is *`delimiters`*`.tokenize(`*`text`*`)`. The `tokenize()` method categorizes each character in the *`text`* argument as either part of a delimiter (the character appears in the *`delimiters`* string) or part of a token (any other character) and then divides the *`text`* argument into tokens separated by delimiters. The method returns the tokens as a sequence of strings. If you try to tokenize an empty string the returned sequence is empty. The `tokenize()` method is useful for extracting words from whitespace. For example:

```
string s := "     This   is\na test!  See? ")
print " ".tokenize(s).toString();
print " .,:;!?\n\t".tokenize(s).toString();
```

This prints the following:

```
["This","is\na","test!","See?"]
["This","is","a","test","See"]
```

- `toLower()` — returns an all-lowercase `string` representation of the string.
- `toUpper()` — returns an all-uppercase `string` representation of the string.

Primitive and string types

Reference types

In addition to the primitive types, EPL provides for a number of object types. These types are manipulated by reference as opposed to by value (in the same way as complex types are handled in Java). They are:

- "action" on page 39
- "Channel" on page 40
- "chunk" on page 41
- "context" on page 43
- "dictionary" on page 44
- "event" on page 49
- "Exception" on page 52
- "listener" on page 55
- "location" on page 55
- "sequence" on page 57
- "StackTraceElement" on page 62
- "stream" on page 63

When a variable of reference type is assigned to another one of the same type, the latter will reference the same object as the former, and should one be changed, the other one would reflect the change as well.

If you require a variable of reference type to contain a copy of another one of the same type, that is a completely distinct but identical copy, then you should use the `clone()` method as described below. This returns a deep copy of the variable, that is, it copies it and all its contents (and their contents in turn) recursively.

The `string` type is technically a reference type, but unlike all other reference types, the `string` type is immutable; its value cannot change. The `clone()` method has no effect on strings, as they cannot be changed.

Note that you cannot use an object type for matching in an event template. For example, suppose you have the following event types:

```
InnerEvent
{
   float f;
}
```

```
WrapperEvent
{
   string s;
   InnerEvent anInnerEvent;
}
```

The following statement is correct:

```
on all WrapperEvent(s = "some_string")
```

However, the following statement is not allowed:

```
on all WrapperEvent(anInnerEvent.f = 5.5)
```

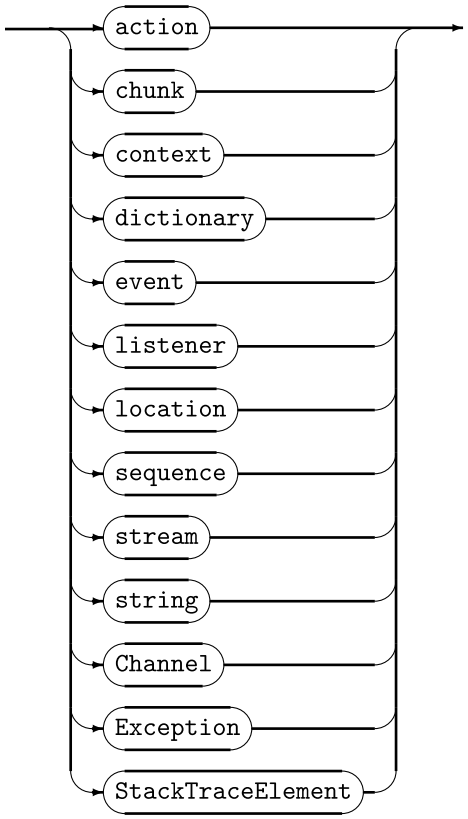
More than one variable can have a reference to the same underlying data value. For example, consider the following code:

```
sequence <integer> s1;
sequence <integer> s2;
s1 := [12, 55, 42];
s2 := s1;
print s1[1].toString; // print second element of s1
s2[1] := 99; // change the second element
print s1[1].toString; // print second element of s1 again
```

Both `s1` and `s2` refer to the same array, so whichever variable you use, there is only one copy of the data values. So the program's output is

```
55
99
```

ReferenceDataTypeName



Types

action

In addition to defining an action, you can define a variable whose type is `action`. This lets you assign an action to an `action` variable of the same `action` type. An action is of the same type as an `action` variable if they have the same argument list (the same types in the same order) and return type (if any).

Usage

Defining `action` type variables is useful for invoking an action and for passing an action to another action.

You can use an `action` variable anywhere that you can use a `sequence` or `dictionary` variable. For example, you can

- Pass an action as a parameter to another action.
- Return an action from execution of an action.
- Store an action in a local variable, global variable, event field, `sequence`, or `dictionary`.

You must initialize an `action` variable before you try to invoke it.

You cannot send, route, emit, or enqueue an event that contains an `action` type member.

When an action variable is a member of an event the behavior of the action depends on the instance of the event that the action is called on. Consequently, it can be handy to bind an action variable member with a particular event instance. This is referred to as creating a closure. For details, see "Declaring action variables in event definitions" in *Developing Apama Applications in EPL*.

An `action` variable is a potentially-cyclic type — a type that directly or indirectly refers to itself. For details about the behavior of such objects, see "Potentially cyclic types" on page 72.

When the correlator clones a value that contains an action variable, or copies a value that contains an action variable into a new monitor because of a spawn operation, the correlator preserves the structure inside the value. This means that if two things are references to the same object in the original value, they will be references to the same object in the copy. This includes objects referred to by closures that have been assigned to action variables.

When you call `toString()` on an object that contains an action variable the result is the name of the method or action in the action variable. If the action variable contains a closure, the `toString()` method outputs the bound value followed by the name of the action or method being called on the value. For example:

```
"E(42).f"
"12.0.rand"
```

See "String form of potentially cyclic types" on page 74.

When the `toString()` method encounters an empty action variable the output is `new` followed by the type. Following are two examples:"

- "`new action<>`"
- "`new action<sequence<string>,float> returns boolean`"

Methods

The only operation that you can peform on an `action` variable is to call it. You do this in the normal way by passing a set of parameters in parentheses after an expression that evaluates to the `action` variable. For an example and additional details, see "Using action type variables" in *Developing Apama Applications in EPL*.

Reference types

Channel

Values of `Channel` type are objects that hold either a string, which is a channel name, or a context object depending on how you construct it.

Usage

The `Channel` type is defined in the `com.apama` namespace. Typically, to easily refer to `Channel` objects, you specify

```
using com.apama.Channel
```

The `Channel` type lets you send an event to a channel or context. If the `Channel` object contains a string then the event is sent to the channel with that name. If the `Channel` object contains a context then the event is sent to that context.

A Channel object has three constructors:

```
Channel(string)
Channel(context)
new Channel
```

The third constructor creates a `Channel` object that contains an empty context object. The contained empty context is the same result you would get from `new context`. It is a runtime error to send an event to an empty context. Likewise, it is a runtime error to send an event to a `Channel` object that contains an empty context.

For example, the following two lines have the same result:

```
send e to "MyChannel";
send e to Channel("MyChannel");
```

Similarly, the following two lines have the same result when `c` is a variable of the `context` type:

```
send e to c;
send e to Channel(c);
```

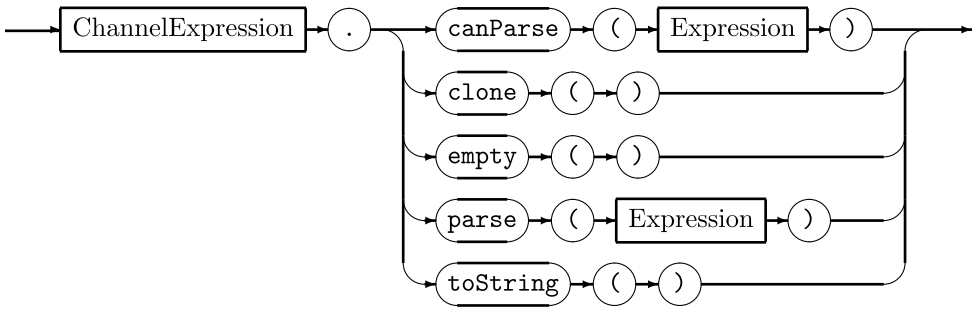
The benefit of using a `Channel` object rather than a `string` or `context` object is that the `Channel` object is polymorphic. For example, by using a `Channel` object to represent the source of a request, you could write a service monitor so that the same code sends a response to a service request. You would not need to have code for sending responses to channels and separate code for sending responses to contexts.

You cannot send an event to a sequence of `Channel` objects. You cannot route a `Channel` object but a routable object can have a `Channel` object as a member.

Methods

The following methods may be called on values of `Channel` type:

ChannelMethods



- `canParse()` — returns `true` if the string argument can be successfully parsed to create a `Channel` object. You cannot parse a string representation of a `Channel` object that contains a context. For more information about the parseable type property, see the table in "Type properties summary" on page 64.
- `clone()` — returns a new `Channel` that is an exact copy of the `Channel` the `clone()` method is called on. The original `Channel`'s content is copied into the new `Channel`.
- `empty()` – returns true if the `Channel` object contains an empty context. This lets you distinguish between an object that contains a default initialization value and an object that has been explicitly populated.
- `parse()` – returns the `Channel` instance represented by the `string` argument. You can call this method on the `Channel` type or on an instance of a `Channel` type. The more typical use is to call `parse()` directly on the `Channel` type.

  The `parse()` method takes a single string as its argument. This string must be the string form of a `Channel` object. The string must adhere to the format described in *Deploying and Managing Apama Applications*, "Event file format". For example:

  ```
  Channel a;
  a := Channel.parse(com.apama.Channel("channelName");
  Channel b;
  b := Channel.parse(com.apama.Channel(context(3, "contextName", true) );
  ```

  You can specify the `parse()` method after an expression or type name. If the correlator is unable to parse the string, it is a runtime error and the monitor instance that the EPL is running in terminates.
- `toString()` — returns a string that contains the channel name or the name of the contained context.

Reference types

chunk

Values of the `chunk` type are references to dynamically allocated opaque objects whose contents cannot be seen or directly manipulated in EPL. They are used by correlator plug-ins to store state information across multiple plug-in method calls.

In EPL, `chunk` reference values can be held in variables of the type `chunk` and passed as parameters to plug-ins when they are called. The `chunk` type lets you reference data that has no equivalent EPL type.

It is not possible to perform operations on data of type `chunk` from EPL directly; the `chunk` type exists purely to allow data output by one external library function to pass through to another function. Apama does not modify the internal structure of `chunk` values in any way. As long as a receiving function expects the same type as that output by the original function, any complex data structure can be passed around using this mechanism.

To use `chunks` with plug-ins, you must first declare a variable of type `chunk`. You can then assign the chunk to the return value of an external function or use the chunk as the value of the *`out`* parameter in the function call.

The following example illustrates this. The `complex.test4()` method prints output to `stdout`. Apama provides the source code for `complex_plugin`. You can find it in the Apama `samples\correlator_plugin\cpp` directory.

```
monitor ComplexPluginTest {
   // Load the plugin
   import "complex_plugin" as complex;
   // Opaque chunk value
   chunk myChunk;
   action onload() {
      // Generate a new chunk
      myChunk := complex.test3(20);
      // Do some computation on the chunk
      complex.test4(myChunk);
   }
}
```

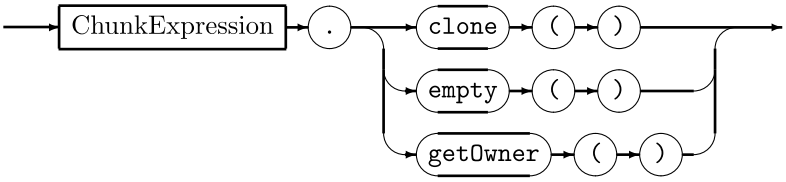
Although the `chunk` type was designed to support unknown data types, it is also a useful mechanism to improve performance. Where data returned by external plug-in functions does not need to be accessed from EPL, using a `chunk` can cut down on unnecessary type conversion. For example, suppose the output of a `localtime()` method is a 9-element array of type `float`. While you could declare this output to be of type `sequence<float>`, there is no need to do so because the EPL never accesses the value. Consequently, you can declare the output to be of type `chunk` and avoid an unnecessary conversion from native array to EPL `sequence` and back again.

An event can contain a field of type `chunk`, however you cannot send, emit, route, or enqueue an event that has a `chunk` type field.

Methods

The following methods may be called on variables of `chunk` type.

ChunkMethod



- `clone()` – requests that the plug-in return a new `chunk` that is an exact copy of the `chunk` that `clone()` was called on. `The clone()` method calls the `copy()` C++ virtual member function on the existing `AP_Chunk` object.

  See "Working with chunk in C++" in *Writing Correlator Plug-ins*.

- `empty()` – returns true if the chunk is empty. This lets you distinguish between a chunk that contains a default initialization value and a chunk that has been explicitly populated by a correlator plug-in. You can also get an empty chunk as a result of a `new chunk` expression.
- `getOwner()` – returns a string that contains the name of the correlator plug-in that the chunk belongs to. The name returned is the name you specify as the first argument in the `import` statement that loads the correlator plug-in. For example:

  ```
  import "TimeFormatPlugin" as tfp;
  ```

  The `getOwner()` method on a chunk from that plug-in returns `"TimeFormatPlugin"` and not `"tfp"`.

  The `getOwner()` method returns an empty string if the chunk is empty.

Reference types

context

Values of the `context` type are references to contexts. A context lets EPL applications organize work into threads that the correlator can concurrently execute.

Usage

Use one of the following constructors to create a `context` reference:

```
context(string name)
context(string name, boolean receivesInput)
```

The optional `receivesInput` Boolean flag controls whether the context is public or private:

- `true` — A public context can receive external events on the default channel, which is the empty string (`""`). There is no requirement for a monitor instance in this context to subscribe to the default channel.
- `false` — A private context does not receive external events on the default channel. This is the default.

When you create a `context` reference, the context might or might not already exist. You use the context reference to spawn to the context or send an event to the context. When you spawn to a context, the correlator creates the context if it does not already exist.

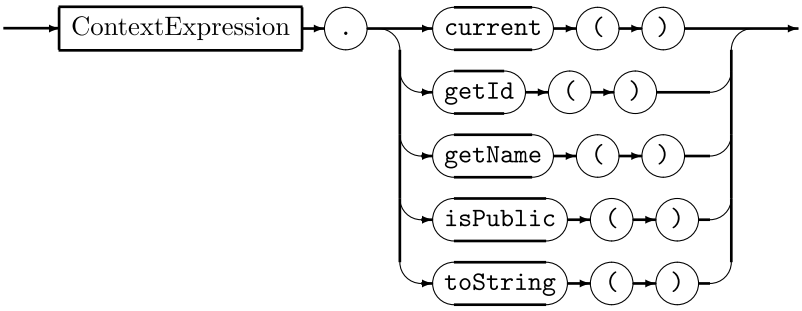
When you start a correlator it has a single main context. You can then create additional contexts. Context objects are lightweight and creating one only creates a stub object and allocates an ID. In other words, when you create a context, you are actually creating a context reference.

A context is subscribed to the union of the channels each of the monitor instances in that context is subscribed to. This is a property of the monitor instances running in a context and is not accessible by means of the context reference object.

Methods

After you create a context, you can call the following instance methods on that context:

ContextMethods



- `current()` — Returns a context object that is a reference to the current context. The current context is the context that contains the monitor instance or event instance that is calling this method.
- `getId()` — Returns an integer that is the ID of the context.
- `getName()` — Returns a string that is the name of the context.
- `isPublic()` — Returns a Boolean that indicates whether the context is public. If the context was created as a public context then the return value is true.
- `toString()` — Returns a string that contains the properties of the context. For example, for a public context whose name is `test`, the content of the returned string would be something like this:

  ```
  context(1, "test", true)
  ```

In addition, the `current()` static method returns a reference to the current context.

See also "Contexts" on page 103.

Reference types

dictionary

A `dictionary` is a means of storing and retrieving data based on an entry key. This enables, for example, a user's name to be retrieved from a unique user ID.

The syntax of a `dictionary` definition is:

```
dictionary < key, item > varname
```

Dictionaries are dynamic and new entries can be added and existing entries deleted as desired.

The dictionary key must be a comparable type. See "Comparable types" on page 71.

The *`item`* can be any Apama type.

Two dictionaries are equal only if they contain the same keys and the same value for each key. When dictionaries are not equal they are ordered as though they were sequences of key-value pairs, sorted in key order.

Example

```
// A simple stock dictionary, each stock's name is gained and
// stored from a numerical key
//
dictionary< integer, string > stockdict;
```

```
// A dictionary that can be used to store the number of times
// that a given event is received
//
dictionary< StockChoice, integer > stockCounterDict;
```

Note that a `dictionary` of `sequence`s or `dictionary`s is supported. Care must be taken in how these are specified by separating trailing `>` characters with whitespace, to distinguish them from the right-shift operator `>>`. For example:

```
// A correctly specified dictionary containing sequence elements
   dictionary< integer, sequence<float> > willWork;

// An incorrectly specified dictionary containing sequence elements
// dictionary< integer, sequence<float>> willNotWork;
```

A global variable of type `dictionary` is initialized by default to an empty instance of the type defined. On the other hand, a local variable must be explicitly initialized using the `new` operator, as follows:

```
dictionary<integer, string> stockdict;
stockdict := new dictionary <integer, string>;
```

It is also possible to both declare and populate a variable of type `dictionary` as a single statement, regardless of the scope in which the variable is declared, as follows:

```
dictionary<integer, string> stockdict := {1:"IBM", 2:"MSFT", 3:"ORCL"};
```

using `{}` to delimit the dictionary, a comma `,` to delimit individual entries, and a colon `:` to separate keys and values.

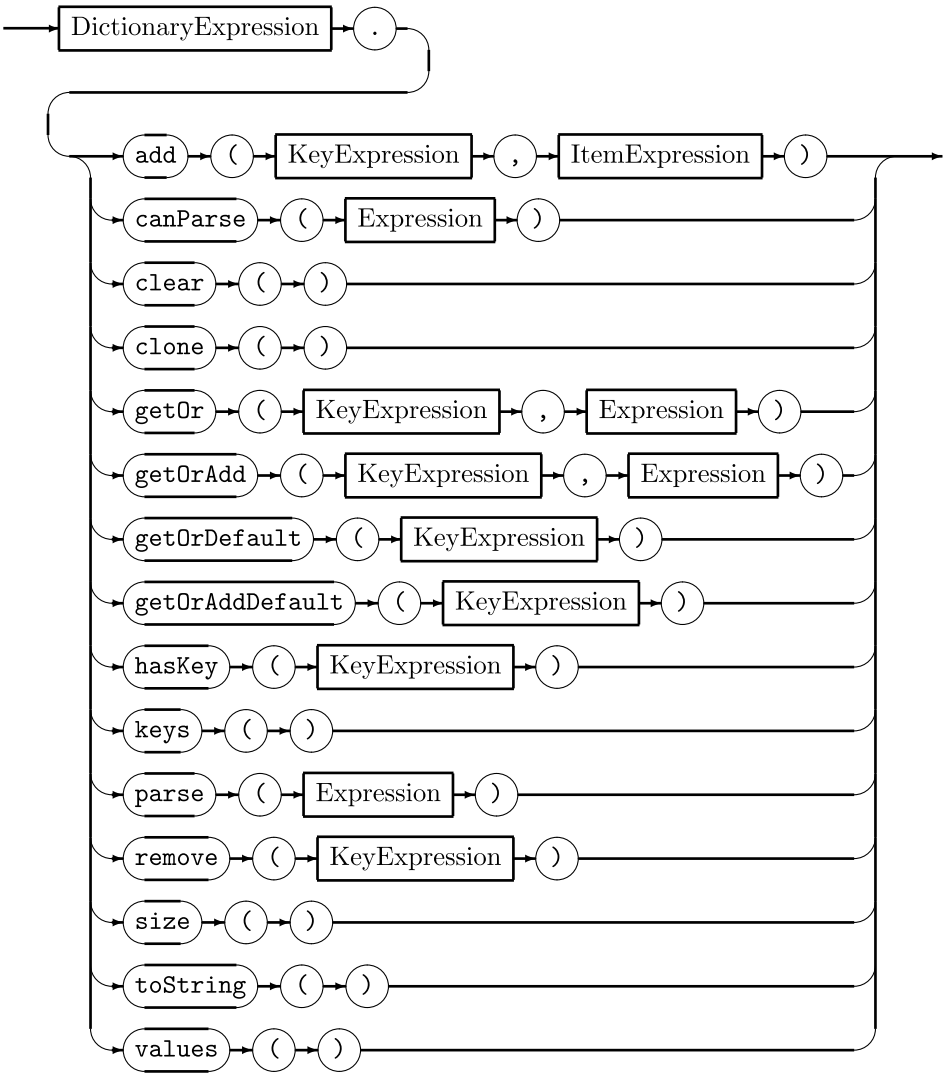
Dictionary types do not allow duplicate keys. Ensure that you do not specify duplicate keys when initializing a dictionary or in a string that will be parsed to produce a dictionary.

A `dictionary` variable can be a potentially cyclic type — a type that directly or indirectly refers to itself. For details about the behavior of such objects, see "Potentially cyclic types" on page 72.

Methods

The methods available on the `dictionary` data structure are:

DictionaryMethods



- `add(key, item)` – add an entry to the dictionary. The first parameter is an expression whose type is the same type as the dictionary's key type and which becomes the entry's key. The second parameter is an expression whose type is the same type as the dictionary's item type and whose value becomes the entry's item value. The key expression is evaluated first, then the item expression. There is no return value. For example:

  ```
  stockdict.add(71, "ACME");
  ```

  When you are adding an entry and the key you specify already exists in the dictionary, the correlator replaces the item already in the dictionary with the new item.

- `canParse()` — this method is available only on dictionaries where the item type is parseable. Returns `true` if the string argument can be successfully parsed to create a dictionary object. For more information about the parseable type property, see the table in "Type properties summary" on page 64.

- `clear()` – sets the size of the dictionary to 0, deleting all entries. Takes no parameters. Returns no value.

- `clone()` – returns a new `dictionary` that is an exact copy of the `dictionary`. All the `dictionary`'s contents (both keys and items) are cloned into the new `dictionary`, and if the items were complex types themselves, their contents are cloned as well.

  When the dictionary you are cloning is a potentially cyclic type, the correlator preserves multiple references, if they exist, to the same object. That is, the correlator does not create a separate copy of the object to correspond to each reference. See also "Potentially cyclic types" on page 72.

- `getDefault(`*`key`*`, `*`item`*`)` – Before Apama 5.0, the `getOr()` method was called `getDefault()`. You should not use the `getDefault()` method. It remains only for backwards compatibility, it is deprecated, and it will be removed in a future release. Use the `getOr()`) method instead.

- `getOr(`*`key`*`, `*`alternative`*`)` – returns the item that corresponds to the specified key. If the specified key is not in the dictionary, the `getOr()` method returns *`alternative`*. The benefit of calling this method is that if you were to call `dictionary[key]` instead of `dictionary.getOr()` and the key you were trying to look up did not exist, the correlator would terminate the monitor instance.

  The `getOr()` method lets you avoid a call to the `hasKey()` method before you look up a key.

  For example, suppose you have the following dictionary:

  ```
  dictionary<integer,string> integerSqrts := {
     1:"one", 4:"two", 9:"three", 16:"four", 25:"five", 36:"six",
     49:"seven", 64:"eight", 81:"nine", 100:"ten" };
  ```

  Now suppose you call the following method:

  ```
  integerSqrts.getOr(key, "irrational")
  ```

  Assume that you specify a key that is in the range of 1 - 100. If the value of the key is a square of an integer, `getOr()` returns the written form of the key's square root. For any other key value, `getOr()` returns `"irrational"`.

- `getOrDefault(`*`key`*`)` – retrieves an existing item by its key or returns a default instance of the dictionary's item type if the dictionary does not contain the specified key.

  The `getOrDefault()` method lets you avoid a call to the `hasKey()` method before you look up a key.

- `getOrAdd(`*`key`*`, `*`alternative`*`)` – retrieves an existing item by its key or adds the specified key to the dictionary with *`alternative`* as its value if it is not already present and also returns the specified alternative.

  The `getOrAdd()` method lets you avoid a call to the `hasKey()` method before you look up a key. If the item type is complex, a call to the `getOrAdd()` method can be more efficient than a call to the `getOr()` method, because it will not construct a default item unless necessary.

- `getOrAddDefault(`*`key`*`)` – retrieves an existing item by its key or, if it is not already present, adds the specified key with a default instance of the dictionary's item type and returns that instance.

  For example, suppose you want to maintain a record of which client companies each sales representative handles. You might write:

  ```
  dictionary<string, sequence<string> > representing := {};
      representing.getOrAddDefault("Sue").append("We-Haul");
      representing.getOrAddDefault("Joe").append("McDonuts");
      representing.getOrAddDefault("Sue").append("ACME");
  ```

  The first time `getOrAddDefault()` is called with key `"Sue"`, that key does not exist yet, so it is added with an empty sequence as the item. That empty sequence is then returned, so `"We-Haul"` can be appended to it. The second time `getOrAddDefault()` is called with key `"Sue"`, the existing sequence (containing `"We-Haul"`) is returned, so `"ACME"` can be appended to it.

This idiom is considerably simpler and more efficient than testing `hasKey()` and then either adding or retrieving.

- `hasKey(key)` – returns `true` if a key exists within the dictionary, `false` otherwise. Takes one parameter, which is an expression whose type is the same as the referenced dictionary's key type and whose value is the key value whose presence in the dictionary is tested.

  For example: `stockdict.hasKey(71)`

- `keys()` – returns a `sequence` of the dictionary's keys sorted in ascending order. This will be a `sequence` of the same type as the key type of the dictionary. The primary purpose of this method is to enable one to iterate over a dictionary's contents by looping through the sequence of its keys, as follows;

  ```
  integer k;
  for k in stockdict.keys() {
    myString := stockdict[k];
  }
  ```

  The `keys()` method performs a deep copy (like the `clone()` method) of the dictionary keys into a `sequence`; that is by value as opposed to by reference. This behavior ensures that the result of `keys()` is a consistent view of the dictionary's keys at the time `keys()` was called, regardless of whether entries were added to or removed from the dictionary while examining the result of `keys()`. This also ensures that the dictionary keys themselves cannot be modified by changing the sequence.

- `parse()` – this method is available only on dictionaries where the item type is parseable. Returns the `dictionary` object represented by the `string` argument. For more information about the parseable type property, see the table in "Type properties summary" on page 64. You can call this method on the `dictionary` type or on an instance of a `dictionary` type. The more typical use is to call `parse()` directly on the `dictionary` type.

  The `parse()` method takes a single string as its argument. This string must be the string form of a `dictionary` object. The string must adhere to the format described in *Deploying and Managing Apama Applications*, "Event file format". For example:

  ```
  dictionary<string, integer> d := {};
  d := dictionary<string, integer>.parse("{\"foo\":1, \"bar\":2}");
  ```

  You can specify the `parse()` method after an expression or type name. If the correlator is unable to parse the string, it is a runtime error and the monitor instance that the EPL is running in terminates.

  When a dictionary is a potentially cyclic type , the behavior of the `parse()` method is more advanced. See "Potentially cyclic types" on page 72.

- `remove(key)` – remove an entry by key. Takes one parameter, which is an expression whose type is the same as the referenced dictionary's `key` type and whose value is the value of the key of the entry to be removed. The `remove()` method does not return a value. If the `key` value is not present in the referenced dictionary, a runtime error is raised.

  For example: `stockdict.remove(71);`

- `size()` – returns as an `integer` the number of elements in the dictionary. Takes no parameters.

- `toString()` – converts the entire dictionary in ascending order of key values to a `string`. This will create a string that contains all the elements enclosed within curly braces, `{ }`, separated by commas, `,`, with each element consisting of the key followed by an item, the two being separated by a colon, `:`.

That is,

```
{key1:item1, ... ,keyn:<itemn>}
```

The string is constructed by invoking the `toString()` method on each of the referenced dictionary's key/value pairs and catenating them into the result.

When a dictionary is a potentially cyclic type, the behavior of the `toString()` method is different. See "Potentially cyclic types" on page 72.

- `values()` – returns a `sequence` of the dictionary's items sorted in ascending order of keys. The order of the items in the returned sequence is the order returned by the dictionary's `keys()` method. The `sequence` contains items that are the same type as the item type in the dictionary. The primary purpose of this method is to let you iterate over a dictionary's contents by looping through the sequence of its item values, as follows;

  ```
  string v;
  for v in stockdict.values() {
     myString := v;
  }
  ```

  The `values()` method performs a shallow copy of the dictionary items, that is, if the items are of a reference type the returned `sequence` contains references to the dictionary's items rather than clones of them. This behavior ensures that a change to an object in the dictionary is reflected in the returned `sequence` and a change to an object in the sequence is reflected in the dictionary.

- `[key]`– retrieve or overwrite an existing item by its key, or create a new item.

  For example, `stockdict[71] := "XRX";`

  If you are using `[key]` to write and if an item with the key `key` does not exist, the correlator creates it. If you are using `[key]` to retrieve and if an item with the key `key` does not exist, it is a runtime error.

Reference types

event

Values of the `event` type are data objects that can represent notifications of something happening, such as a customer order, shipment delivery, sensor state change, stock trade, or myriad other things. Event objects can also be used as a container or structure for holding several related data values.

Usage

Each kind of event has a type name and one or more data elements, called event fields, associated with it. An event can also have blocks of executable code, called actions, associated with it.

A field in an event can be any Apama type. If an event contains a field of type `action`, `chunk`, `listener`, or `stream`, you cannot specify that event in an event template, and you cannot send, emit, route or enqueue that event.

Two events are equal if corresponding members are equal. If corresponding members are not equal then the events are ordered according to the first member that differs.

The correlator orders events by considering the event's fields in order.

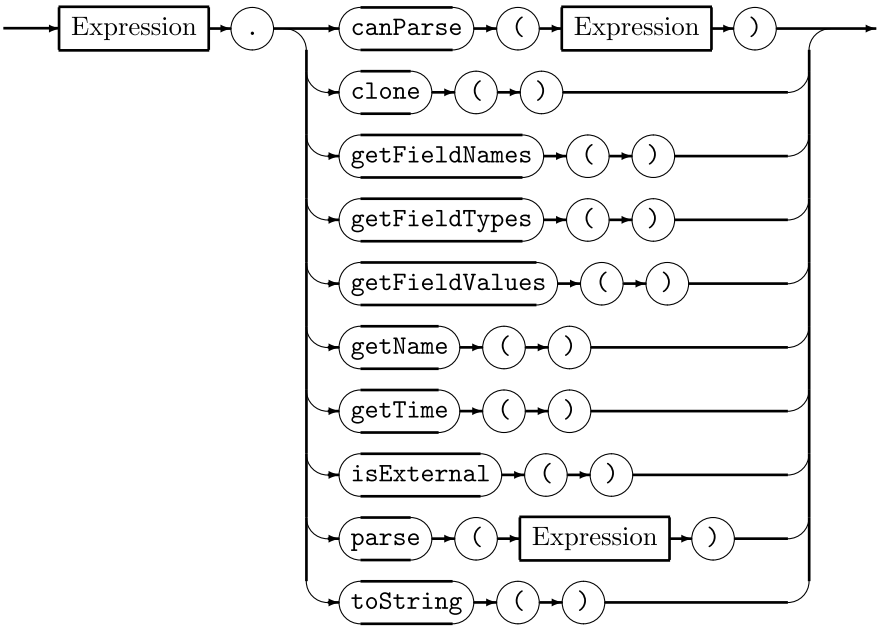
An `event` variable can be a potentially cyclic type — a type that directly or indirectly refers to itself. For details about the behavior of such objects, see "Potentially cyclic types" on page 72.

See *Developing Apama Applications in EPL*, "Defining event types".

Methods

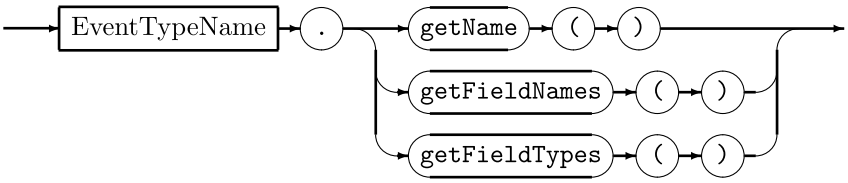
The following methods may be called on variables of `event` type:

EventMethods



The following methods may be called on `event` types:

EventTypeMethodCall



- `canParse()` — this method is available only on events that are parseable. Returns `true` if the string argument can be successfully parsed to create an event object. For more information about the parseable type property, see the table in "Type properties summary" on page 64.
- `clone()` – returns a new `event` that is an exact copy of the `event`. All the `event`'s contents are cloned into the new `event`, and if they were complex types themselves, their contents are cloned as well. Takes no parameters.

  When the event you are cloning is a potentially cyclic type, the correlator preserves multiple references, if they exist, to the same object. That is, the correlator does not create a copy of the object to correspond to each reference. See also "Potentially cyclic types" on page 72.

- `getFieldNames()` – returns a `sequence` of strings that contain the field names of an event type. This method takes no parameters. The return value is of type `sequence <string>`. You can call this method on an event type or on an instance of an `event` type.
- `getFieldTypes()` – returns a sequence of strings that contain the type names of an event type's fields. This method takes no parameters. The return value is of type `sequence <string>`. You can call this method on an event type or on an instance of an `event` type.
- `getFieldValues()` – returns a sequence of strings that contain the field values of an event. This method takes no parameters. The return value is of type `sequence <string>`.
- `getName()` – returns a `string` whose value is an event's type name. This method takes no parameters. You can call this method on an event type or on an instance of an `event` type.
- `getTime()` – returns a `float` that indicates a time expressed in seconds since the epoch, January 1st, 1970. The particular time returned is as follows:
  - If the correlator created this event, the `getTime()` method returns the time that the correlator created the event. This is the creation time in the context in which the correlator created the event.
  - Coassignment sets the timestamp of an event. A call to `getTime()` on a coassigned event returns the time that the correlator performed the coassignment. This is the time in the context in which the correlator performed the coassignment and it can be the time the event was received or routed. For an enqueued event, a call to `getTime()` returns the receiving context's current time when the enqueued event arrived in the context.

    An event's timestamp might not match the time when an event listener for that event fires. For example, consider the following:

    ```
    on A():a and B():b {
    ...
    }
    ```

    Suppose that `currentTime` is `1` when the correlator processes `A` and `currentTime` is `2` when the correlator processes `B`. A call to `a.getTime()` returns `1`, while a call to `b.getTime()` returns `2`. Of course, the event listener fires only after processing `B`.
- `isExternal()` – returns a `boolean` that indicates whether the event was generated by an external source. Such an event came from an external event sender, triggered an event listener, and was coassigned to a monitor instance variable. When a monitor instance enqueues an event, this event is considered to be generated by an external source.

  A return value of `true` indicates an event that was generated by an external source.

  When the correlator spawns a monitor instance, it preserves the value that the `isExternal()` method returns. In other words, if you coassign an external event in a monitor instance, and then spawn that monitor instance, the `isExternal()` method returns true in the spawned monitor instance.

  This method takes no parameters.

  The `isExternal()` method returns `false` when a monitor instance
  - Routes an event that was external
  - Creates an event inside the correlator
  - Clones an event

This method is useful when you need to determine whether an event came from outside or was in some way derived inside the correlator. Although this distinction is often clear from the event type, that is not always the case.

- `parse()` – this method is available only on events that are parseable. Returns the `event` object represented by the `string` argument. For more information about the parseable type property, see the table in "Type properties summary" on page 64. You can call this method on an event type or on an instance of an `event` type. The more typical use is to call `parse()` directly on the event type.

  The `parse()` method takes a single string as its argument. This string must be the string form of an `event` object. The string must adhere to the format described in *Deploying and Managing Apama Applications*, "Event file format". For example:

  ```
  A a := new A;
  a := A.parse("A(10, \"foo\")");
  ```

  You can specify the `parse()` method after an expression or type name. If the correlator is unable to parse the string, it is a runtime error and the monitor instance that the EPL is running in terminates.

  When an event is a potentially cyclic type, the behavior of the `parse()` method is different. See "Potentially cyclic types" on page 72.

- `toString()` – returns a `string` representation of the event. Takes no parameters. The return value is constructed by calling the `toString()` method on each of the referenced event's fields and catenating the individual return values into the result.

  When you define an event type inside a monitor it has a fully qualified name. For example:

  ```
  monitor Test
  {
     event Example{}
  }
  ```

  The fully qualified name for the `Example` event type is `Test.Example` and the `toString()` output for the event name is `"Test.Example()"`.

  When an event is a potentially cyclic type, the behavior of the `toString()` method is different. See "Potentially cyclic types" on page 72.

Also, you can define your own actions on events.

Reference types

Exception

Values of `Exception` type are objects that contain information about runtime errors.

Usage

The `Exception` type is defined in the `com.apama.exceptions` namespace. Typically, you specify `using com.apama.exceptions.Exception` so you can easily refer to `Exception` objects.

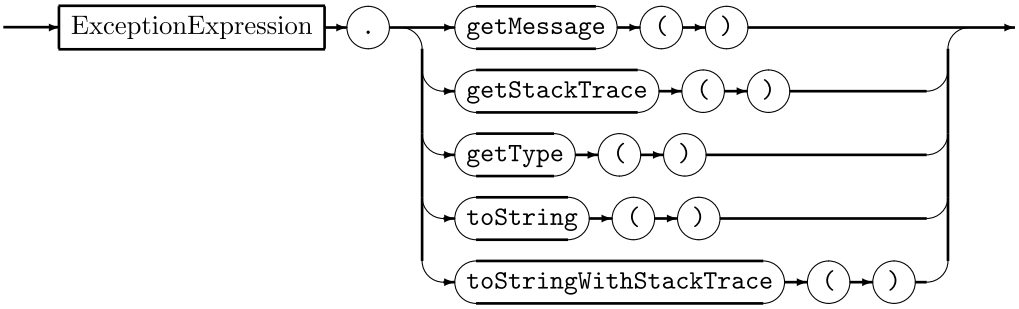
An `Exception` object has methods for accessing an error message, an error type, and a `sequence` of `com.apama.exceptions.StackTraceElement` objects that show where the exception occurred and what the calls were that led to the exception.

You cannot route an `Exception` object but a routable object can have an `Exception` object as a member.

Methods

The following methods may be called on values of `Exception` type:

ExceptionMethods



- `getMessage()` — returns a string that contains the exception message.
- `getStackTrace()` — returns a `sequence` of `StackTraceElement` objects that represent the stack trace for when the exception was first thrown. The sequence is empty if the exception has not been thrown.
- `getType()` — returns a string that contains the exception type, which is one of the following:

Exception Type	Description	Example
`ArithmeticException`	Illegal arithmetic operations	Attempt to divide by 0, call to the `ceil()` method on `NaN`, call to the `exponent()` method on infinity, specifying `NaN` as all or part of an ordered key, call to the `rand()` method on an illegal `float` value such as `Infinity`
`DefaultContextException`	Spawning, sending or enqueuing to a context and specifying a `context` variable that has been declared but the context has not yet been created	`monitor m { context c; action onload() { send A() to c; } }`
`IndexOutOfBoundsException`	Invalid index in a sequence or string operation	`sequence.insert(-1, x)`
`IllegalArgumentException`	Illegal argument value in an expression	`"".split()`
`IllegalStateException`	Calling an action when it is illegal to do so	`spawn` statement in `ondie()` or `onunload()`

Exception Type	Description	Example
`MemoryAllocationException`	Unable to fulfill memory allocation request	An invalid size is passed to the `sequencesetCapacity()` method
`NullPointerException`	Attempt to call an `action` variable when that variable has not been initialized	`event E { action<string> x; } monitor m { E e; action onload() { e.x("This will fail!"); } }`
`OtherInternalException`	An internal error occurred	
`ParseException`	Error that occurs while parsing	`parse("two")` on an integer
`PluginException`	An exception thrown by a correlator plug-in. See the note that follows this table.	
`StackOverflowException`	Attempt to use more space than is available on the stack	

In C++ correlator plug-ins, you can customize exception types so that the type returned has this format:

`PluginException:`*`user_defined_type`*

See `AP_UserPluginException` in the `correlator_plugin.hpp` file in the `include` folder of your Apama installation.

In Java plug-ins, the exception type returned has this format:

`PluginException:`*`class_name`*

For example:

```
import "MyJavaPlugin" as myjavaplugin;
...
  action myAction() {
    try {
      myjavaplugin.processfile("config.txt");
    } catch (Exception e) {
      log "Exception of type "
        + e.getType() at WARN;
    }
  }
...
```

Returns something like:

```
Exception of type
PluginException:java.io.FileNotFoundException
```

- `toString()` — returns a string that contains the exception message and the exception type.
- `toStringWithStackTrace()` — returns a string that contains the exception message, the exception type, and the stack trace elements.

Reference types

listener

A handle to a listener.

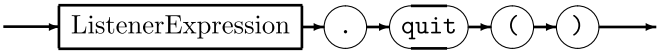
Usage

A `listener` variable can be instantiated only by assigning to it the outcome of an `on` statement, a `from` statement, or by assigning to it the value of another `listener` variable. Values of `listener` type are references to `listener` objects created with the `on` statement or `from` statement. The main use of `listener` variables is, in combination with the `listener` type's inbuilt `quit` method, to terminate an active listener when it is no longer needed.

An event can contain a field of type `listener`, however you cannot send, emit, route, or enqueue an event that has a `listener` type field. Also, you cannot specify an event with a `listener` field in an event template.

The following method may be called on variables of `listener` type.

ListenerMethods



- `quit()` – causes the listener to terminate immediately.

  If the listener is invalid or has already been quit, then the `quit()` method does nothing and does not raise an error.

  The `quit()` method takes no parameters and does not return a result.

Reference types

location

Values of the `location` type describe rectangular areas in a two-dimensional unitless Cartesian coordinate plane. Locations are defined by the `float` coordinates of two points `x1`, `y1` and `x2`, `y2` at diagonally opposite corners of an enclosing boundary rectangle.

The format of a `location` type is as follows:

```
location(x1, y1, x2, y2)
```

An example of a valid location therefore looks as follows:

```
location(15.23, 24.234, 19.1232, 28.873)
```

A point can be represented simply as a rectangle with both corners being the same point. You can access the data members of a `location` type in the same way that you access the fields of an event. For example:

```
location l := location(1.0, 2.0, 3.0, 4.0);
print l.x1.toString();
```

This prints `1.0`. You can use a `location` type to describe a rectangular area but you can also use it to describe various other quantities, such as line segments connecting two endpoints, circles, vectors, or points in a four-dimensional space. However, certain inbuilt methods, such as the `inside()` method, give correct results only for boundary rectangles.

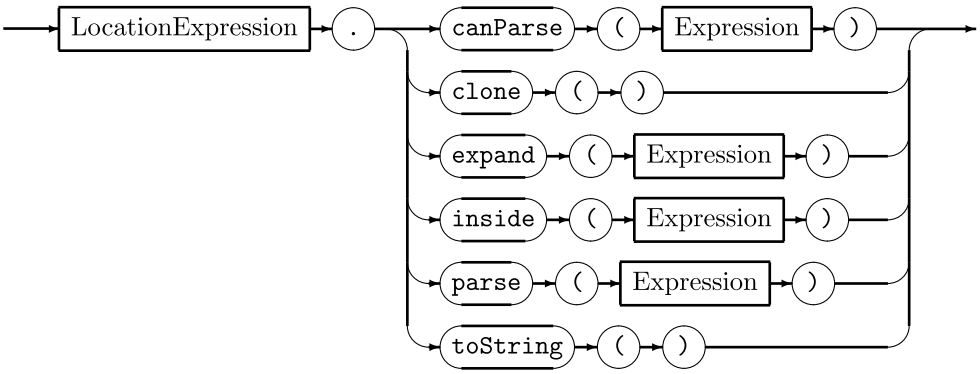
A listener that is watching for a particular value for a `location` field matches when it finds a `location` field that intersects with the `location` value specified by the listener. In the following example, the listener matches each `A` event whose `loc` field specifies a location that intersects with the square defined by `(0.0, 0.0, 1.0, 1.0)`. When the limits specified for a `location` type are out of order, the correlator correctly orders them before performing a comparison. When locations touch it is considered to be an intersection.

```
location l := location(0.0, 0.0, 1.0, 1.0);
on all A(loc = l) ...
```

Methods

The following methods may be called on variables of `location` type:

LocationMethods



- `canParse()` — returns `true` if the string argument can be successfully parsed.
- `clone()` – returns a new `location` that is an exact copy of the `location`.
- `expand(`*`float`*`)` – returns a new `location` expanded by the value of the `float` parameter in each direction. For example:

  ```
  location l := location(0.0, 0.0, 0.0, 0.0);
  on all A(loc = l.expand(0.5)) ...
  ```

  This event listener watches for `A` events whose `loc` field specifies a location that intersects with `(-0.5, -0.5, 0.5, 0.5)`.

- `inside(`*`location`*`)` – returns `true` if the location is entirely enclosed by the space defined by the `location` parameter, `false` otherwise. Note that if the two locations are exactly equal, the result of calling the `inside()` method is `false`.

- `parse()` – method that returns the `location` instance represented by the `string` argument. You can call this method on the `location` type or on an instance of a `location` type. The more typical use is to call `parse()` directly on the `location` type.

  The `parse()` method takes a single string as its argument. This string must be the string form of a `location` object. The string must adhere to the format described in *Deploying and Managing Apama Applications*, "Event file format". For example:

  ```
  location a;
  a := location.parse("(15.23, 24.234, 19.1232, 28.873)");
  ```

  You can specify the `parse()` method after an expression or type name. If the correlator is unable to parse the string, it is a runtime error and the monitor instance that the EPL is running in terminates.

- `toString()` – returns a `string` representation of the `location`.

Reference types

sequence

Values of the `sequence` type are finite ordered sets or arrays of entries whose values are all of the same primitive or reference type. The type can be any Apama type.

Usage

Sequences are indexed by nonnegative integers from 0 to one less than the number of entries given by their `size` inbuilt method. Sequences are dynamic and new entries can be added and existing entries deleted as needed.

The individual elements of a sequence can be referenced in several ways.

- With subscripts — use the `[ ]` operators in combination with an integral expression, to reference sequence elements as an array. For example, `aSequence[3]` refers to the fourth element of a sequence. The first element is `aSequence[0]`. The last, for a sequence with `n` elements is `aSequence[n-1]`
- With the `for` loop — use the `for` loop to iterate over the individual elements of the sequence from first to last. See "The for statement" on page 123.
- With instance methods — You can use the `indexOf`, `insert`, `delete` (and others) methods to operate on individual elements.

Two sequences are equal if they are the same length and corresponding elements are equal. Otherwise, they sort according to the earliest difference. For example:

- `"abc"` sorts earlier than `"abcXYZ"`
- `[1,2,3]` sorts earlier than `[1,3,0]`
- `[1,2,3]` sorts earlier than `[1,2,3,77,88,99]`

The empty sequence sorts earliest of all.

Syntax

The syntax for `sequence`s is:

```
sequence< type > varname
```

For example:

```
// A sequence to hold the names and volume of all my stocks
// (assuming the StockNameAndPrice event type includes a string
// for stock name and float for the volume)
sequence<StockNameAndPrice> MyPortfolio;

// A sequence to hold a list of prices
sequence<float> myPrices;
```

Note that `sequence`s of `sequence`s (and so on) are also supported. Care must be taken in how these are specified by separating trailing `>` characters with white space, to distinguish them from the right-shift operator `>>`. For example:

```
// A correctly specified sequence containing sequence elements
sequence< sequence<float> > willWork;

// An incorrectly specified sequence containing sequence elements
sequence<sequence<float>> willnotWork;
```

A global variable of type `sequence` is initialized by default to an empty instance of the type defined. On the other hand, you must explicitly initialize a local variable using the `new` operator, as follows

```
sequence<integer> someNumbers;
someNumbers := new sequence<integer>;
```

It is also possible to both declare and populate a variable of type `sequence` as a single statement, regardless of the scope in which the variable is declared, as follows:

```
sequence<integer> someNumbers := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
```

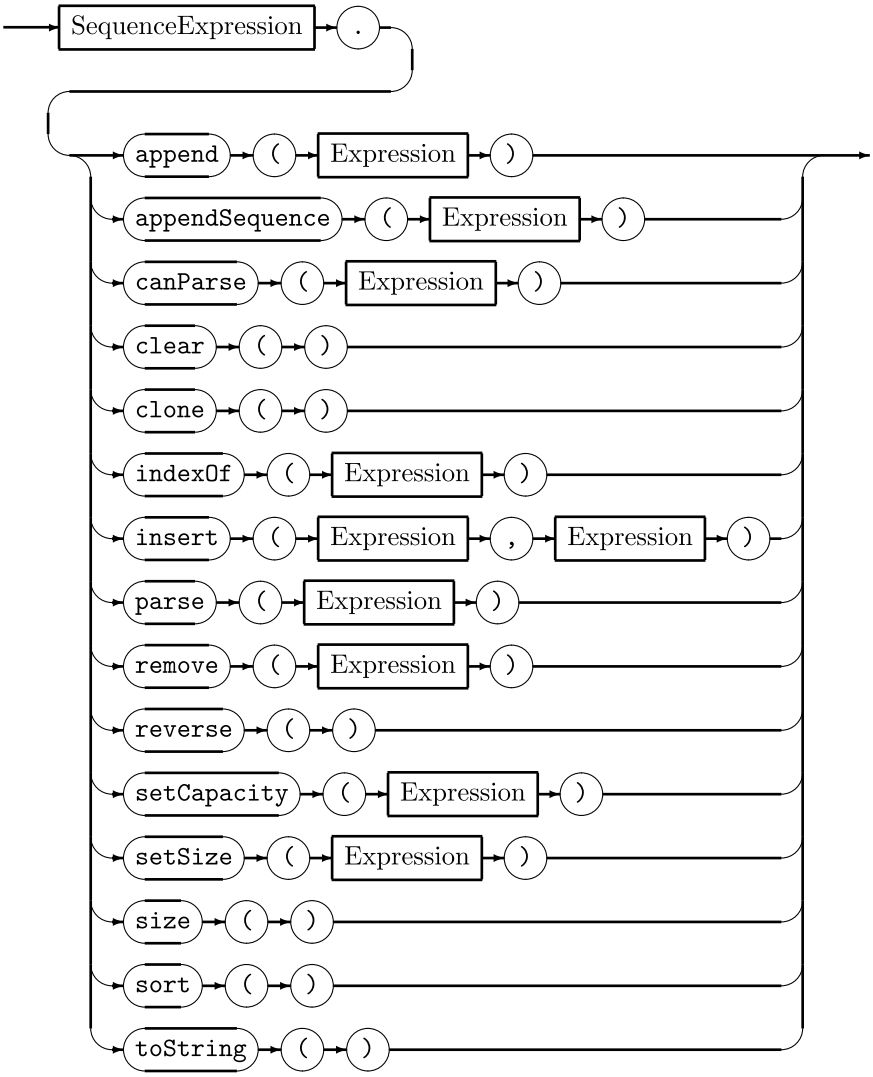
Use `[]` to delimit the sequence and a comma `,` to delimit individual elements.

A `sequence` variable can be a potentially cyclic type — a type that directly or indirectly refers to itself. For details about the behavior of such objects, see "Potentially cyclic types" on page 72.

Methods

The methods available to the `sequence` data structure are:

SequenceMethods



- `append(item)` – appends the item to the end of the sequence.

  For example: `myPrices.append(55.20);`

- `appendSequence(sequence)` – appends the *`sequence`* to the end of the sequence that this method is called on. The appended sequence must be the same type as the sequence this method is called on.

- `canParse()` — this method is available only on sequences where the item type is parseable. Returns `true` if the string argument can be successfully parsed to create a sequence object. For more information about the parseable type property, see the table in "Type properties summary" on page 64.

- `clear()` – sets the size of the sequence to 0, deleting all entries.

- `clone()` – returns a new `sequence` that is an exact copy of the `sequence`. All the `sequence`'s contents are cloned into the new `sequence`, and if they were complex types themselves, their contents are cloned as well.

When the sequence you are cloning is a potentially cyclic type, the correlator preserves multiple references, if they exist, to the same object. That is, the correlator does not create a copy of the object to correspond to each reference. See also "Potentially cyclic types" on page 72.

- `indexOf(item)` – return as an `integer` the location of the first matching item within the sequence. This method is available only if the item type is a comparable type. For details about whether a type is comparable and, if so, how the comparison is done, see "Comparable types" on page 71. The value returned by `indexOf()` will be from `0` to `size() - 1` if the item is found, or `-1` if the item is not a member of the sequence. A call to `indexOf()` to find the index of a `NaN` value in a sequence of `decimal` or `float` values returns `-1` because `NaN` values cannot be compared for equality by using the standard operator.

- `insert(item, integer)` – insert the item specified into the location indicated by the second parameter. The location must be a valid index within the sequence, or the next index due to be filled. That means that the only valid values are from `0` to `size()`, inclusive. An invalid value will cause a runtime error, which will terminate the enclosing monitor instance.

- `parse()` – this method is available only on sequences where the item type is parseable. Returns the `sequence` object represented by the `string` argument. For more information about the parseable type property, see the table in "Type properties summary" on page 64. You can call this method on the `sequence` type or on an instance of a `sequence` type. The more typical use is to call `parse()` directly on the `sequence` type.

  The `parse()` method takes a single string as its argument. This string must be the string form of a `sequence` object. The string must adhere to the format described in *Deploying and Managing Apama Applications*, "Event file format". For example:

  ```
  sequence<float> s := [];
  s := sequence<float>.parse("[1.0, 4.0, 9.0, 16.0, 25.0]");
  ```

  You can specify the `parse()` method after an expression or type name. If the correlator is unable to parse the string, it is a runtime error and the monitor instance that the EPL is running in terminates.

  When a sequence is a potentially cyclic type , the behavior of the `parse()` method is different. See "Potentially cyclic types" on page 72.

- `remove(integer)` – remove the n^{th} element in the sequence, moving all the elements above it down and reducing the size by 1. Note that in EPL sequence elements are indexed from `0`, i.e. the first element is at location `0`.

  For example: `myPrices.remove(1);`

- `reverse()` – modifies the sequence by reversing the order of the items in the sequence. For example, if the sequence contains `1`, `2`, `3`, `4`, then after execution of `reverse()` the updated sequence contains `4`, `3`, `2`, `1`. There is no return value; the method modifies the sequence in place and does not create a new sequence nor does it create any new items.

- `setCapacity(integer)` – sets the amount of memory initially allocated for the sequence. Note that this does not limit the amount of memory the sequence can use. By default, as you add more elements to a sequence, the correlator allocates more memory. Calling `sequence.setCapacity()` can improve performance because it removes the need to add more memory repeatedly as you add elements to the sequence. For example, consider a sequence that you intend to populate with 1000 elements. A call to `setCapacity(1000)` removes the need to allocate additional memory unless more than 1000 elements are added. A call to this method does not change the behavior of your code.

- `setSize(integer)` – sets the number of elements in the sequence to the specified integer, either deleting entries from the end or adding initialized (using default values of variables) entries to the end.

  For example: `myPrices.setSize(10);`

- `size()` – returns as an `integer` the number of elements in the sequence.

- `sort()` – Sorts the sequence it is called on in ascending order. The type of the sequence items must be comparable. See "Comparable types" on page 71. There is no return value; the method modifies the sequence in place and does not create a new sequence nor does it create any new items. A sequence of `decimal` or `float` values that contains `NaN` values cannot be sorted and will result in termination of the monitor instance that contains the method call.

  For example:

  ```
  sequence<integer> s := [4,2,3,1];
  s.sort();
  ```

  After that, `s` is `[1,2,3,4]`.

- `toString()` – convert the entire sequence to a `string`. This will create a string containing all the elements enclosed within square brackets `[ ]`, separated by commas, `,` . That is, `[`<item$_1$>` , ... , `<item$_n$>` ]`

  When a sequence is a potentically cyclic type, the behavior of the `toString()` method is different. See "Potentially cyclic types" on page 72.

- `[integer]` – retrieve or overwrite an existing entry from the sequence, specifically the one located at the index specified. Note that in EPL sequence elements are indexed from `0`, that is, the first element is number `0`. The index specified must be valid, that is it must be between `0` and `size() - 1`, inclusive, as otherwise a runtime error will occur and the monitor instance will terminate.

  For example: `totalCost := myPrices[1] + myPrices[2];`

Iterating over sequence elements

You can iterate over a sequence both on the elements and on the indices. The indices are numbered from `0` to `size() - 1`, inclusive. For example:

```
sequence<string> seq := ["zero", "one", "two"];

// sequence elements
string s;
for s in seq {
   print s;
}

// sequence indices
integer i := 0;
while i < seq.size() {
   print seq[i];
   i := i + 1;
}
```

Loops are discussed in "Compound statements" on page 123.

Reference types

StackTraceElement

A `StackTraceElement` type value is an object that contains information about one entry in the stack trace.

Usage

A `com.apama.exceptions.Exception` object contains a `sequence` of `StackTrackElement` objects, which indicate where an exception occurred. The correlator generates this sequence. You should not need to create `StackTraceElement` objects yourself. The first object in the sequence points to the line of code that caused the exception. The next object points to the action that contains the code that caused the exception. The next object points to the action that called that action, and so on.

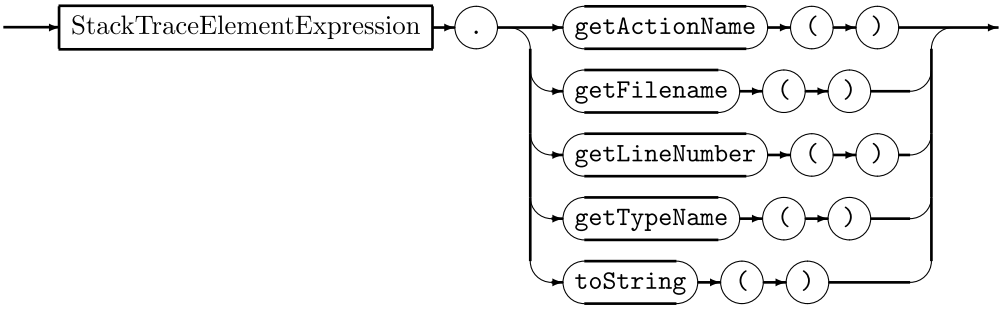
The `StackTraceElement` type is defined in the `com.apama.exceptions` namespace. Typically, you specify `using com.apama.exceptions.StackTraceElement` so you can easily refer to `StackTraceElement` objects.

It is permissible to parse an event that contains a `StackTraceElement` object or a sequence of `StackTraceElement` objects.

Methods

The following methods may be called on values of `StackTraceElement` type:

StackTraceElementMethods



- `getActionName()` — returns a string that contains the name of the action in which the exception occurred.
- `getFilename()` — returns a string that contains the name of the file that contains the code in which the exception occurred.
- `getLineNumber()` — returns an integer that indicates the line number of the code in which the exception occurred.
- `getTypeName()` — returns a string that indicates the type (event, aggregate, monitor) that contains the action in which the exception occurred..
- `toString()` — returns a string whose format is "*`typeName.actionName() filename:linenumber`*".

Reference types

stream

A value of `stream` type refers to a stream. Each stream is a conduit or channel through which items flow. The item types that can flow through streams are `event`, `location`, `boolean`, `decimal`, `float`, `integer`, or `string`. A stream transports items of only one type. Streams are internal to a monitor.

Usage

An event can contain a field of type `stream`, however you cannot send, emit, route, or enqueue an event that has a `stream` type field. Also, you cannot specify an event that has a `stream` field in an event template.

Syntax

The syntax for declaring a `stream` variable is:

```
stream< type > varname
```

Replace *`type`* with the type of the items in the stream. This can be an event type, or `location`, `boolean`, `decimal`, `float`, `integer`, or `string`.

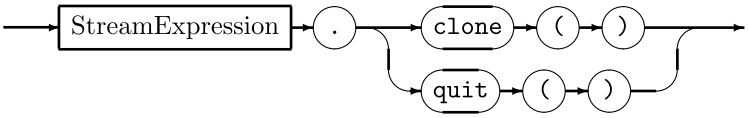
Replace *`varname`* with an identifier for the stream. For example:

```
stream<Tick> ticks;
```

Methods

The methods available to the `stream` type are:

StreamMethods



- `clone()` – returns the original stream. It does not clone it.
- `quit()` – causes a stream listener to terminate.

  If the referenced listener's value is an inert stream, then the `quit()` method does nothing and does not raise an error.

  The `quit()` method takes no parameters and does not return a result.

Reference types

monitor pseudo-type

The use of the `monitor` keyword as a pseudo-type is limited to invocation of the `subscribe()` and `unsubscribe()` methods.

Usage

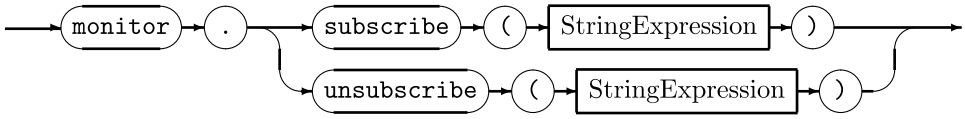
Use the following formats:

```
monitor.subscribe(channel_name);
monitor.unsubscribe(channel_name);
```

Replace *channel_name* with a string expression that resolves to the name of the channel you want to subscribe to or unsubscribe from. In a monitor instance, call these methods from inside an action.

It is not possible to use instances of the `monitor` type. For example, there cannot be variables or event members of type `monitor`. You cannot specify a `com.apama.Channel` object as the argument to `subscribe()` or `unsubscribe()` method.

MonitorMethods



- `subscribe()` — Subscribes the calling context to the specified channel. All listeners in the same context as the calling monitor instance can process events sent to the specified channel. The calling monitor instance owns the subscription. If the calling monitor instance terminates the subscription ends.

  Multiple monitor instances in the same context can subscribe to the same channel. Each event is delivered once as long as any of the subscriptions are active. An event is not delivered once for each subscription.

- `unsubscribe()` — Unsubscribes the calling context from the specified channel. If this was the only subscription in the context to the specified channel then the context no longer processes events sent to the unsubscribed channel.

Types

Type properties summary

Apama type properties include the following:

- Indexable — An indexable type can be referred to by a qualifier in an event template.
- Parseable — A parseable type can be parsed and has `canParse()` and `parse()` methods. The type can be received by the correlator.
- Routable — A routable type can be a field in an event that is
  - Sent by the `route` statement
  - Sent by the `send...to` or `enqueue...to` statement
  - Sent by the `enqueue` statement
  - Sent outside the correlator with the `emit` statement
- Comparable — A comparable type can be used as follows:
  - Dictionary key
  - Item in a sequence on which you can call `sort()` or `indexOf()`
  - Stream query partition key
  - Stream query group key

  - Stream query window `with-unique` key
  - Stream query equijoin key
- Potentially cyclic — A potentially cyclic type uses the `@n` notation when it is parsed or converted to a string. When a potentially cyclic type is cloned, the correlator uses an algorithm that preserves aliases. See "Potentially cyclic types" on page 72
- Acyclic — An acyclic type is a type that is not potentially cyclic.
- `E`-free — `E`-free types cannot contain references to instances of a particular event type `E`. This property is used only to determine whether `E` is acyclic.

The following table shows the properties of each Apama type.

Type	Indexable	Parseable	Routable	Comparable	Acyclic	`E`-free
`boolean`	✓	✓	✓	✓	✓	✓
`decimal`	✓	✓	✓	✓[1]	✓	✓
`float`	✓	✓	✓	✓[1]	✓	✓
`integer`	✓	✓	✓	✓	✓	✓
`string`	✓	✓	✓	✓	✓	✓
`location`	✓	✓	✓	✓	✓	✓
`Channel`	✗	✓[2]	✓	✓	✓	✓
`Exception`	✗	✓	✓	✓	✓	✓
`context`	✗	✗	✓[3]	✓	✓	✓
`listener`	✗	✗	✗	✗	✓	✓
`chunk`	✗	✗	✗	✗	✓	✓
`stream`	✗	✗	✗	✗	✓	✓
`action`	✗	✗	✗	✗	✗	✗
`sequence`	✗	⚠	⚠	★	⚠	⚠

Type	Indexable	Parseable	Routable	Comparable	Acyclic	E-free
dictionary	✗	⚠	⚠	★	⚠	⚠
event E	✗	⚠ 4	⚠ 4	★	★	✗

✓	Yes. This type has the corresponding property. [1] Attempts to use a NaN in a key terminates the monitor instance. [2] A Channel object is parseable only when it contains a string. [3] Although a context can be enqueued, it is not parseable, so the correlator will reject it from the input queue with a warning.
✗	No. This type does not have the corresponding property.
⚠	This type inherits the corresponding property from its constituent types, that is, the item type in a sequence, the key and item types in a dictionary, the types of fields in an event. The type has the corresponding property only when all its constituent types have that property. [4] An event defined inside a monitor cannot be received from an external source nor emitted from that correlator. An event defined inside a monitor can be sent or enqueued only within the same correlator.
★	The type is comparable only when all its constituent types are both comparable and acyclic.
★	An event E is acyclic only when all its constituent types are both acyclic and E-free.

Examples

The following code provides examples of event type definitions and their properties.

```
// You can do everything with "Tick", including index both its fields.
   event Tick {
       string symbol;
       float price;
   }
// You can do everything with "Order", except refer to its target or
// properties fields in an event template.
   event Order {
       string customer;
       Tick target;
       string symbol;
       float quantity;
       dictionary<string,string> properties;
   }
// The correlator cannot receive the next event as an external event and
// you cannot usefully enqueue it, but you can send it, route it, or
// enqueue it to a context.
    event SubscriptionRequest {
        string channel;
        context recipient;
    }
```

```
// You can do very little with this event except access its members and
// methods. It cannot be routed, you cannot sort sequence<TimeParse>,
// trying to group a stream query by TimeParse is illegal, and so on.
     event TimeParse {
         import "TimeFormatPlugin" as TF;
         string pattern;
         chunk compiledPattern;
     }
// This has all the same restrictions as TimeParse, but is also
// potentially cyclic, so will use the @n format when parsed or
// converted to a string.
     event Room {
         string roomName;
         float squareFeet;
         sequence<Room> adjacentRooms;
         sequence<Employee> occupants;
     }
```

Types

Timestamps, dates, and times

Although EPL does not have time, date, or datetime types, timestamp (a date and time) values can still be represented and manipulated because EPL uses the `float` type for storing timestamps. See "currentTime" on page 160.

Timestamp values are encoded as the number of seconds and fractional seconds (to a resolution of milliseconds) elapsed since midnight, January 1, 1970 UTC and do not have a time zone associated with them. Although the resolution is to milliseconds, the accuracy can be plus or minus 10 milliseconds, or some other value depending on the operating system.

If you have two float variables that both contain timestamp values, subtracting one from the other gives you the difference in seconds.

You can add or subtract a time interval from a timestamp by adding or subtracting the appropriate number of seconds (60.0 for 1 minute, 3600.0 for 1 hour, 86,400.0 for 1 day, and so forth).

See also:

- `event.getTime()` for information about when the correlator assigns timestamps to events.
- "Using the Time Format plug-in" in *Developing Apama Applications in EPL* for information about formatting timestamps.

Types

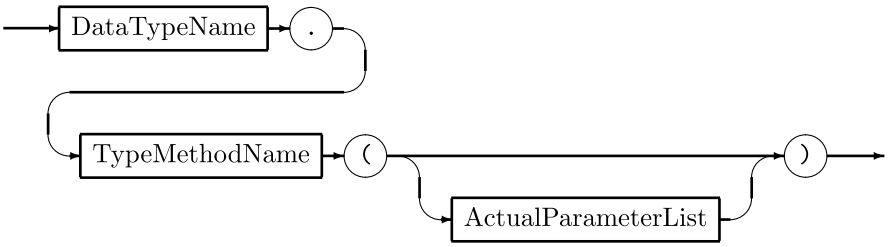
Type methods and instance methods

There are two kinds of inbuilt methods — type methods and instance methods. Type methods are associated with types. Instance methods are associated with values.

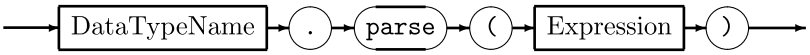
Type methods

To call a type method, you specify the name of the type followed by a period, followed by the method name with its parameters enclosed in parentheses. Some methods do not have parameters and for them you must supply an empty parameter list.

TypeMethodCall



CommonTypeMethodCall



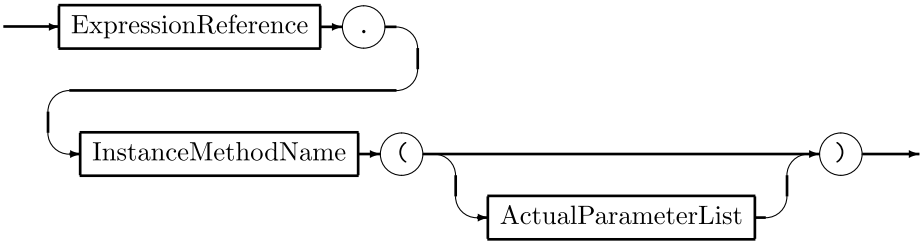
Examples

```
event someEvent;
{
   integer n;
}
integer i;
i:=integer.getUnique();
print someEvent.getName();
```

Instance methods

Each type (except `action`), whether primitive or reference, has a number of instance methods that provide a number of useful functions and operations on instance variables of that type. These methods are quite similar to actions except that they are predefined and associated with variables, not monitors or events.

InstanceMethodCall



To call an instance method, you specify an expression followed by a period and the name of the method, followed by a parenthesized list of actual parameters or arguments to be passed to the method when it is called.

Example

```
integer i := 642;
float f;
f := i.toFloat ();
print f.formatFixed (5);
```

The ExpressionReference can be a simple variable name or a more complicated expression that includes a namespace, action or method calls, literals, subscript operators and so on.

The InstanceMethodName is the name of a method that is specific to the type of the variable. The ActualParameterList syntax is described in "Action and method calls" on page 133.

Some methods do not have parameters and for those, you must supply an empty argument list as follows: `methodname ()`.

See Also

See the following sections for the methods you can call on types and instances:

- "boolean" on page 18
- "float" on page 21
- "integer" on page 27
- "string" on page 30
- "action" on page 39
- "context" on page 43
- "dictionary" on page 44
- "event" on page 49
- "listener" on page 55
- "location" on page 55
- "sequence" on page 57
- "stream" on page 63

Types

Type conversion

EPL requires strict type conformance in expressions, assignment and other statements, parameters in action and method calls, and most other constructs. This means that

- The left and right operands of most binary operators must be of the same type.
- An actual parameter passed in a method or action invocation must be of the same type as the type of the corresponding formal parameter in the action or method definition.
- The expression result type on the right side of an assignment statement must be the same type as that of the target variable.
- The expression result type in a variable initializer must be the same type as that of the variable.
- The expression result type in a subscript expression must be integer.
- The expression result type in a `return` statement must be the same type as that specified in the action's `returns` clause.

EPL does not allow implicit or explicit casting to perform type conversions. Instead, the inbuilt methods associated with each type include a set of methods which perform type conversion. For example:

```
string number;
integer value;
number := "10";
value := number.toInteger();
```

This illustrates how to map a `string` to an `integer`. The string must start with some numeric characters, and only these are considered. So if the string's value was "`10h`", the integer value obtained from it would have been `10`. Had the conversion not been possible because the string did not start with a valid numeric value, then `value` would have been set to `0`.

These method calls can also be made inside event expressions as long as the type of the value returned is of the same type as the parameter where it is used. Therefore one can write:

```
on all StockTick("ACME", number.toFloat());
```

Method calls can be chained. For example one can write:

```
print ((2 + 3).toString().toFloat() + 4.0).toString();
```

Note that as shown in this example, method calls can also be made on literals.

The following table indicates the source and target type-pairs for which type conversion methods are provided.

Source Type	Target Type							
	boolean	decimal	dictionary	event	integer	float	sequence	string
boolean	assign							toString()
decimal		assign			round() ceil() floor()	toFloat()		toString()
dictionary			assign and clone					toString()
event				assign and clone				toString()
integer		toDecimal()			assign and toFloat()			toString()
float		toDecimal()			round() ceil() floor()	assign		toString()
sequence							assign and clone	toString()
string	toBoolean() parse()	toDecimal() parse()	parse()	parse()	toInteger() parse()	toFloat() parse()	parse()	assign and parse()

In the table above, "assign" means values of the type can be directly assigned to another variable of the same type, without calling a type conversion method and "clone" means a value of the type can be copied by calling the `clone()` method.

Types

Comparable types

The operators <, >, <=, >=, =, or != can be used to compare two values that are both the same type and that type can only be one of the following types:

- `decimal`
- `float`
- `integer`
- `string`

The following types are considered to be comparable types because you can use each one in a sequence that you plan to sort:

- `boolean`
- `decimal`
- `float`
- `integer`
- `string`
- `context`
- `dictionary` if it contains items that are a comparable type
- `event` if it contains only comparable types
- `location`
- `sequence` if it contains items that are a comparable type

The correlator cannot compare the following types of items:

- `action`
- `chunk`
- `dictionary` if it contains items that are an incomparable type
- `event` if it contains at least one incomparable type
- `listener`
- `sequence` if it contains items that are an incomparable type
- `stream`
- Potentially cyclic types

For details about how the correlator compares items of a particular type, see the topic about that type.

In EPL code, you must use a comparable type in the following places:

- As the key for a `dictionary`. The type of the items in the `dictionary` does not need to be comparable.
- In a `sequence` if you want to call the `indexOf()` or `sort()` method on that `sequence`.
- As a key in the following stream query clauses:
  - Equi-join
  - `group by`
  - `partition by`
  - `with unique`

Types

Cloneable types

Since variables of reference types are bound to the runtime location of the value rather than the value itself, direct assignment of a variable of reference type copies the reference (that is, the value's location) and not the value. To make a copy of the value, you must use the `clone` instance method instead of assignment. The types that have this property are called cloneable types.

The cloneable types are `string`, `dictionary`, `event`, `location`, and `sequence`.

For `dictionary`, `event`, and `sequence` types, the behavior of the `clone()` method varies according to whether or not the instance is potentially cyclic.

- When the instance is potentially cyclic, the correlator preserves multiple references, if they exist, to the same object. That is, the correlator does not create a copy of the object to correspond to each reference. See also "Potentially cyclic types" on page 72.
- When the instance is not potentially cyclic, and there are multiple references to the same object, the correlator makes a copy of that object to correspond to each reference.

While you can call the `clone()` method on a `stream` value, or a value that indirectly contains a `stream` or `listener` value, cloning returns another reference to the original stream or listener and does not clone it.

Types

Potentially cyclic types

A cyclic object is an object that refers directly or indirectly to itself. For example:

```
event E {
   sequence<E> seq;
}
E e := new E;
e.seq.append(e);
```

When an object is cyclic or contains a reference to a cyclic object, it can be referred to as containing cycles. If it is possible to create an object that contains cycles, the type of that object is referred to as potentially cyclic.

When a type has the potential to contain cycles, and you call `parse()` on that type, or `toString()` or `clone()` on an object of that type, the result is different from when those methods are called on a type, or object of a type that is not potentially cyclic. Consequently, it is sometimes important to understand which types are potentially cyclic and what the string form of these objects looks like.

This is described in the following topics:

- "Which types are potentially cyclic?" on page 73
- "String form of potentially cyclic types" on page 74

Types

Which types are potentially cyclic?

A type is potentially cyclic if it contains one or more of the following:

- A `dictionary` or `sequence` type that has a parameter that is of the enclosing type. For example:

  ```
  event E {
     dictionary<integer,E> dict;
  }
  event E {
     sequence<E> seq;
  }
  ```

- An `action` variable member. For example:

  ```
  event E {
     action<E> a;
  }
  ```

- A potentially cyclic type. For example:

  ```
  event E {
     sequence <E> seq;
  }

  event F {
     E e;
  }
  ```

  `F` does not have any members that refer back to `F`, nor does it contain any `action` variables. However, it does contain `E`, which is a potentially-cyclic type. Therefore, an instance of `F` might contain cycles.

  Likewise, a `dictionary` or `sequence` is potentially cyclic if it has a parameter that is a potentially cyclic type. Consider the following event type:

  ```
  event E {
     sequence <E> seq;
  }
  ```

  Given this event type, `dictionary<string, E>` is potentially cyclic because its parameter is potentially-cyclic. Similarly, `sequence<E>` is potentially cyclic.

A cyclic object can indirectly contain itself. Consider the following, using the same definition of `E` as above.

```
E e1 := new E;
E e2 := new E;
e1.seq.append(e2);
e2.seq.append(e1);
```

In this example, both `e1` and `e2` are cyclic:

- `e1` is `e1.seq[0].seq[0]`
- `e2` is `e2.seq[0].seq[0]`

Following is another example of an object that indirectly contains a cycle:

```
E e3 := new E;
E e4 := new E;
e3.seq.append(e4);
e4.seq.append(e4);
```

In this example, `e3` is cyclic, even though it does not refer back to itself. Instead, `e3` refers to `e4` and `e4` refers back to itself.

You can pass objects that contain cycles between EPL and Java. Remember that JMon programs do not support `action` type variables, and so any cyclic types you pass cannot contain them.

Potentially cyclic types

String form of potentially cyclic types

A potentially cyclic object might have more that one reference to the same object. When you need the string form of a potentially cyclic object, the correlator uses a special syntax to ensure that you can distinguish multiple references to the same object from references to separate objects that merely have the same content.

When the correlator converts a potentially cyclic object to a string, the correlator labels that object `@0`. If the correlator encounters a second object during execution of the same method, it labels that object as `@1`, and so on. Whenever the correlator encounters an object that it has already converted, it outputs that object's `@index` label rather than converting it again. For example:

```
event E { sequence<E> seq; }
E e := new E;
e.seq.append(e);
print e.toString(); // "E([@0])"
```

Following is a more complicated example:

```
event Test {
   string str;
   sequence<Test> seq;
   string str2;
}

monitor m {
   action onload() {
      Test t:=new Test;
      t.str:="hello";
      t.str2:=t.str;
      t.seq.append(t);
      Test t2:=new Test;
      t.seq.append(t2);
      t.seq.append(t2);
      t2.seq.append(t);
      print t.toString();
   }
}
```

This prints the following:

```
Test("hello",[@0,Test("",[@0],""),@2],"hello")
```

The objects `@0`, `@1`, `@2`, and `@3` correspond to the following:

`@0`	`Test("hello",[@0,Test("",[@0],""),@2],"hello")`	`t` in the above example
`@1`	`[@0,Test("",[@0],""),@2]`	`t.seq` in the above example
`@2`	`Test("",[@0],"")`	`t2` in the above example
`@3`	`[@0]`	`t2.seq` in the above example

The following example uses the `clone()` method and contains `action` references. The result uses the new string syntax for aliases to the same object.

```
event E {
   action<> act;
   sequence<string> x;
   sequence<string> y;
}

monitor m {
   action onload() {
      E a:=new E;
      a.x.append("alpha");
      a.y:=a.x;
      E b:=a.clone();
      b.x[0]:="beta";
      print b.y.toString();
      print a.toString();
   }
}
```

The output is as follows:

```
["beta"]
E(new action<>,["alpha"],@1)
```

Note that dictionary keys can never contain aliases so they do not receive `@n` labels for referenced objects in `toString()` and `parse()` methods.

Whether you need to do anything to handle this string syntax depends on why you want a string representation of your object:

- If you are using the string for diagnostic messages, you just need to understand the syntax.
- If you plan to feed the string into the `parse()` method, the `parse()` method will handle it correctly.
- If you plan to feed the string into some other program, you should either avoid repeated references in an object or make sure the other program can handle the `@index` syntax.

Potentially cyclic types

Support for IEEE 754 special values

EPL supports the following IEEE 754 special `float` and `decimal` values:

- `NaN` — in EPL, these are quiet `NaNs`. The string representation is "`NaN`".
- `+Infinity` — The string representation is "`Infinity`".
- `-Infinity` — The string representation is "`-Infinity`".

The correlator returns one of these values as the result of an invalid computation. For example, dividing zero by zero or calculating the square root of a negative number. The correlator returns infinities as the result of computations that overflow, for example taking a very large number and dividing it by a very small number.

The correlator can receive external events that contain these special values. You can send, route, emit, and enqueue events that contain these values. If the correlator receives an event that contains a floating point value that is too large to be represented as a 64-bit floating point number the behavior is as if the value had overflowed and the correlator represents the value as infinity.

The following operations return `NaN`:

- `0.0/0.0`
- `x.sqrt()` (where `x` < `0`)
- `x.ln()` (where `x` < `0`)
- `x.log10()` (where `x` < `0`)
- `Infinity - Infinity`
- `0.0 * Infinity`

In addition, most operations that accept `NaN` as a parameter return `NaN`. For example:

- `NaN.exp() = NaN`
- `NaN + 3.0 = NaN`

The `NaN` value behaves differently when compared to other floating point numbers. `NaN` does not compare equal to any other number, including itself. It is unordered with respect to all other floating point numbers, so `NaN` < `x` and `NaN` > `x` are both false.

The following operations return positive infinity (note that IEEE 754 has signed zeroes):

- `x/0.0` (where `x` > `0`)
- `x/-0.0` (where `x` < `0`)
- `Infinity.sqrt()`

The following operations return negative infinity:

- `x/0.0` (where `x` < `0`)
- `x/-0.0` (where `x` > `0`)
- `(0.0).ln()`

The following table lists the available constants. These are provided to ensure consistent values, and a few have been provided for convenience.

Constant	Value
`decimal.E` `float.E`	Euler's number, e
`decimal.PI` `float.PI`	The ratio of a circle's circumference to its diameter — `3.14159265`

Constant	Value
`decimal.MIN` `float.MIN`	The smallest, positive, normalized floating point number. (~2e-308)
`decimal.MAX` `float.MAX`	The largest, finite, positive floating point number. (~2e+308)
`decimal.EPSILON` `float.EPSILON`	The smallest *x* where `(1+x) > 1`. Note that `decimal.EPSILON` and `float.EPSILON` are not the same value. The value is dependent on whether the type is `decimal` or `float`.
`decimal.NAN` `float.NAN`	IEEE 754 Not-a-Number.
`decimal.INFINITY` `float.INFINITY`	IEEE 754 positive infinity.
`integer.MAX`	Largest positive value an integer can take (2^{63} - 1).
`integer.MIN`	Largest negative value an integer can take (-2^{63}).

Special cases of pow()

In the normal case, `x.pow(y)` yields exactly what you might expect, so `3.0.pow(3.0) = 27.0` and `2.0.pow(0.5) = 1.41421`. But there are a very large number of special cases. The documentation for `fdlibm`, which is the mathematics library used by the EPL interpreter for `float` types lists the special cases shown below. Although EPL uses a different math library for `decimal` types, the behavior is the same for `float` and `decimal` types.

- $(anything)^{0} = 1$
- $(x)^{1} = x$, for any `x`
- $(anything)^{NaN} = NaN$
- $NaN^{(\text{anything except } 0)} = NaN$
- $x^{+\infty} = +\infty$, if $|x| > 1$
- $x^{-\infty} = +0$, if $|x| > 1$
- $x^{+\infty} = +0$, if $|x| < 1$
- $x^{-\infty} = +\infty$, if $|x| < 1$
- $\pm 1^{\pm\infty} = NaN$
- $+0^{(+\text{anything except } 0 \text{ and } NaN)} = +0$
- $-0^{(+\text{anything except } 0, NaN \text{ and odd integer})} = +0$
- $+0^{(-\text{anything except } 0 \text{ and } NaN)} = +\infty$
- $-0^{(-\text{anything except } 0, NaN \text{ and odd integer})} = +\infty$
- $-0^{(\text{odd integer})} = -(+0^{(\text{odd integer})})$

- $+\infty^{(+\text{anything except 0 and NaN})} = +\infty$
- $+\infty^{(-\text{anything except 0 and NaN})} = +0$
- $-\infty^{(\text{anything})} = -0^{(-\text{anything})}$
- $(-\textit{anything})^{(\text{integer})} = (-1)^{(\text{integer})} * (+\textit{anything}^{(\text{integer})})$
- $(-\textit{anything}\ \text{except 0 and}\ \infty)^{(\text{non-integer})} = \text{NaN}$

Types

Chapter 3: Events and Event Listeners

In EPL, an event is a data object that is a notification of something happening, such as arrival of a customer order, shipment delivery, sensor state change, stock trade, or myriad other things. Each kind of event has an event type name, zero or more data elements or fields, and zero or more event actions associated with it.

Event objects can also be used simply as complex data structures to hold multiple related data values. They can also be used as a container for actions that can be shared by multiple monitors.

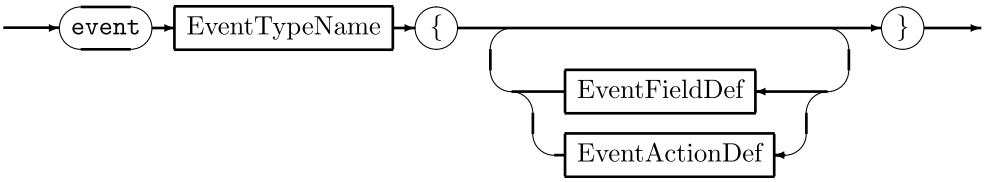
Event objects are hierarchical structures that can contain simple values, other events, and arrays.

When the correlator executes an `on` statement, it creates an event listener. An event listener watches for an event, or a sequence of events, that matches the event or event sequence specified in the `on` statement. Conceptually, event listeners sift the events that come in to the correlator and watch for matching events.

Event definitions

An event definition specifies the event type, and any event fields and/or event action fields.

EventDefinition



Events and Event Listeners

Event fields

An event field definition specifies the type and name of the field.

EventFieldDef

Rule components

DataTypeDef is as described in "Variable declarations" on page 150. Event fields that do not have the wildcard attribute are indexed by the correlator when you listen for them. There can be at most 32 indexes on an event type. Event fields of the type `location` use two indexes for each field.

An event that contains an `action`, `chunk`, `listener`, and/or `stream` field is valid only within the monitor that creates it. You cannot send, enqueue or route an event that contains, directly or indirectly, a field of such types.

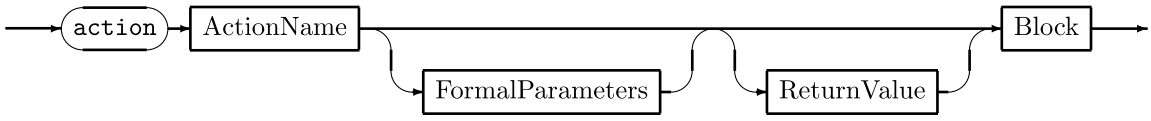
Event definitions

Event actions

An event action is a subprogram or function that is associated with the event definition. It can be invoked or called from any monitor or from another action in the same event. Like monitor actions, the caller must supply actual parameters of the same type and number as the event action's formal parameters and if the action returns a value, then the return value must be consumed by the caller.

Unlike monitor actions (see "Monitor actions" on page 101), events do not have the special actions `onload()`, `onunload()`, and `ondie()`.

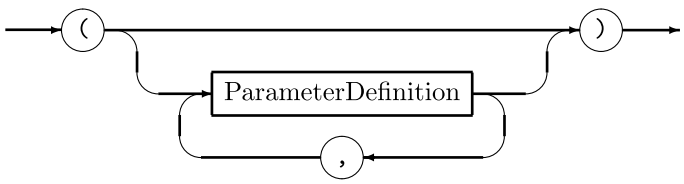
EventActionDef



Event action formal parameters

Event action FormalParameters is a comma-separated list of ParameterDefinitions, enclosed in parentheses.

FormalParameters



Rule components

The FormalParameters is a comma-separated list of ParameterDefinitions, enclosed in parentheses. A parameter definition consists of a type name and an identifier. The identifier is the name of a parameter variable which will be bound to a copy of the value of an expression specified by the caller (that is, the value passed by the caller) when the action is invoked. The number and type of actual parameter values passed by a caller must match those listed in the action's formal parameters.

The scope of a parameter variable is the statement or block that forms the action body. Parameter variables are very similar to an action's local variables.

Event action return value

An event action return value specifies the return value type.

ReturnValue

Rule components

If the event action definition includes a `returns` clause, then the action returns a value of the specified type. All control paths within the action body must lead to a `return` statement before the end of the action body.

Event action body

The block construct forms the event action body. All variable references within an event action body must be one of the following:

- A field of the event
- A formal parameter of the action
- A local variable defined in the action body

Event definitions

Event field and action scope

The scope of an event's fields and actions is the same as the scope of the event itself except that the event fields are always referenceable within the event's actions.

Event definitions

Event templates

An event template is a construct that allows you to specify qualifying or matching criteria based on values of one or more of an event's fields. In event templates, you can qualify only on those event fields whose type is a primitive type. Event templates are used with `on` statements. See "The on statement" on page 125.

EventTemplate

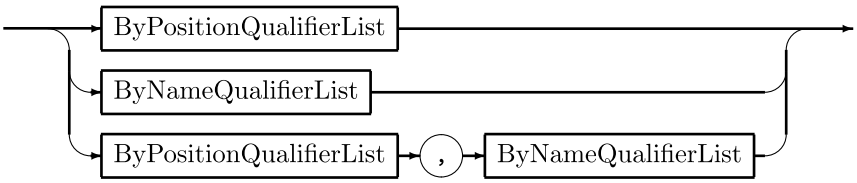
An event template begins with the name of an event type that is to be matched.

Event template s can be either positional or named or a combination of both. Further, the criteria can be omitted entirely, in which case any event of the same event type will match.

Optionally, a colon and an identifier can follow the event expressions. This is called an event coassignment and specifies a variable whose value will become (that is, will be assigned) a reference to the matched event structure when the correlator detects a matching event and listener, and invokes the actions defined in the listener.

See also "Stream source templates" on page 148.

EventQualifierExprList



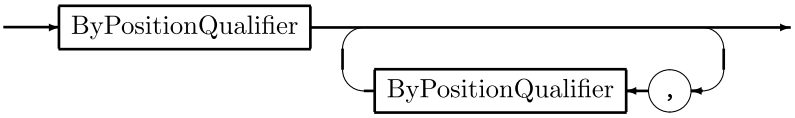
When both positional and named qualifiers are present in an event template qualifier expression list, the positional matches must come first.

Events and Event Listeners

By-position qualifiers

The correlator evaluates a positional event template against the event field that is at the same position in the event definition as the qualifier's position in the qualifier list.

ByPositionQualifierList



Example

For example, suppose an event has the fields shown below:

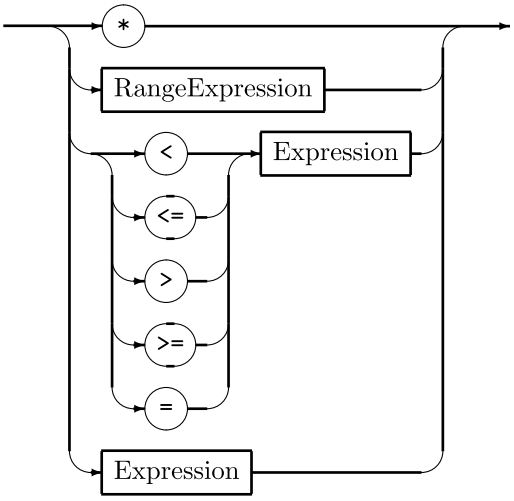
```
event sample1
{
   string  itemName;
   float   price;
   integer quantity;
}
```

An example of a by-position qualifier list for this event is as follows:

```
sample1 ("eggs", 0.50, 3)
```

This template matches `sample1` events that have an `itemName` value of `"eggs"`, a `price` value of `0.50`, and a `quantity` value of `3`.

ByPositionQualifier



Rule components

In ByPositionQualifier constructs, * matches any value of an event field in the corresponding position.

A RangeExpression(see "Range expressions" on page 84) matches the event field values in the corresponding position to a low and high boundary value of the range. A match occurs when the field value is within the range.

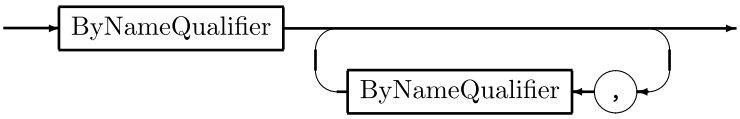
The relational operators < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), and = (equal to) specify a comparison of the event field value with the expression value that follows. A match occurs when the relation result is true. The expression to the right of the relational operator cannot contain any references to the event's fields and must have a result type that is the same as the event field's type and must be one of `decimal`, `float`, `integer` or `string`.

Event templates

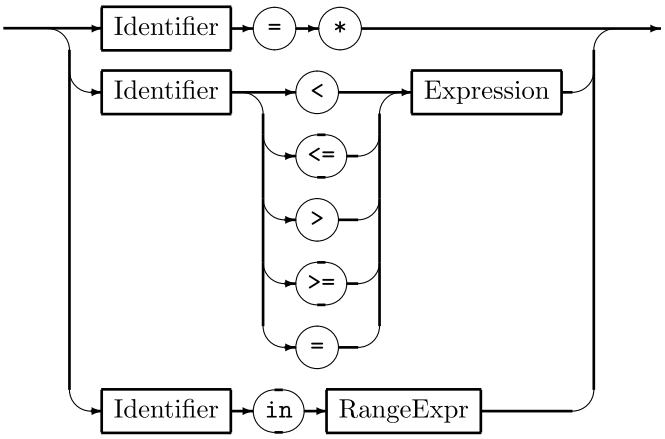
By-name qualifiers

In ByNameQualifiers, the qualifier names an event field whose value is to be matched, instead of matching by position.

ByNameQualifierList



ByNameQualifier



Rule components

The identifier must be the name of one of the event's fields. The field's type must be `integer`, `decimal`, `float`, or `string`. Each event field is allowed to appear only once on the left side of a by-name qualifier and the same field is not allowed in both a by-position qualifier and a by-name qualifier in the same event template.

If the qualifier is of the form Identifier `= *`, this means the qualifier matches all possible values of the specified event field.

If the qualifier is of the second form, using one of the relational operators < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), and = (equal to), then the event field value is compared with the event template's value and a match occurs when the result of the comparison is true.

If the qualifier is of the third form, using `in` followed by a range expression, then the field is compared against the boundary values of the range.

The expression or range expression on the right side is not allowed to refer to any of the event's fields.

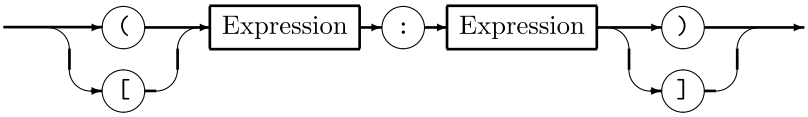
The expression or range expression is evaluated once, when the `on` statement containing the template is executed and its event expressions evaluated, not each time an event of the same type is processed by the correlator.

Event templates

Range expressions

A RangeExpression is a part of a qualifier expression that describes a range of consecutive `decimal`, `float`, `integer`, or `string` values between a low boundary and a high boundary. The correlator tests an event's field value against this range to determine whether or not it falls within the specified range.

RangeExpr



Rule components

The first expression's value forms the low boundary value and the second expression's value forms the high boundary value.

Both expression values must be of the same type and one of `decimal`, `float`, `integer`, or `string`. Both expression types must be of the same type as the event field being tested. Neither expression can contain any references to the event's fields.

If the low boundary value is greater than the high boundary value the EPL runtime automatically reverses them.

Example

In the following EPL, the three `on` statements specify event listeners that are all listening for the same range of events:

```
event test
{
   string s;
   float f;
}
monitor RangeExample
{
   test t;
   action onload()
   {
      on test (f > 9.0 ) and test (f <= 10.0)
      {
      }
      on test ("", (9.0 : 10.0])
      {
      }
      on test (f in (9.0 : 10.0])
      {
      }
   }
}
```

Depending on which of the starting operators, [or `(`, and ending operators, `]` or `)`, you use, the boundary values will either be included in the range or excluded from it. If the starting operator is [, then the low boundary value is included and candidate values greater than or equal to the low boundary value are in the range. If the starting operator is `(`, then the low boundary value is excluded and candidate values larger than the low boundary value are in the range. If the ending operator is `]`, then the high boundary value is included and candidate values less than or equal to the high boundary value are in the range. If the ending operator is `)`, then the high boundary value is excluded and candidate values lower than the high boundary value are in the range. Note that you can have one kind of starting operator at the beginning and the other kind at the end; they do not need to match.

Event templates

Event listener definitions

You define an event listener in an `on` statement. See "The on statement" on page 125.

Events and Event Listeners

Event lifecycle

An event enters the correlator in one of the following ways:

- An event is received from another component, such as the `engine_send` utility, an adapter, another correlator, or a process that is using the Apama client API. The correlator places the event on the input queue of each context that is subscribed to the channel on which the event is sent. If an event is not sent on a named channel then the correlator places the event on the input queue of each public context.
- An EPL program creates an event instance and executes a `send..to` statement. If the target is a channel then the correlator places the event on the input queue of each context that is subscribed to that channel. If the target is a context (or a sequence of contexts) then the correlator places the event on the input queue of that context (or on the input queue of each context in the sequence).
- An EPL program creates an event instance and executes an `enqueue...to` statement. The correlator places the event on the input queue of the specified context or on the input queue of each context in the specified sequence of contexts.
- An EPL program creates an event instance and executes an `enqueue` statement. The correlator places the event on the input queue of each public context. If the input queue for a public context is full then the correlator keeps the event on a special queue for enqueued events until there is room on the input queue that was full.
- An EPL program creates an event instance and executes a `route` statement. The correlator places the event on the input queue of only the context that contains the monitor instance that routed the event.

When the event gets to the front of the context's input queue, the correlator evaluates the event to determine if it is a match for any active event listeners in that context. That is, the correlator checks whether there are any event listeners in that context that are watching for that particular event. If there is a match, the match triggers the event listener. This means that the correlator executes the actions defined in the matching event listener.

It is possible for the actions defined in the event listener to route one or more events back to the context's input queue. A routed event goes right to the front of the context input queue. When the correlator is finished processing the event that triggered the event listener action, the correlator evaluates any routed events before it moves on to the event that was on the input queue after the matching event.

Events and Event Listeners

Event listener lifecycle

When you inject a monitor into the correlator, the correlator instantiates the monitor in the main context and executes the monitor's `onload()` action. The `onload()` action typically specifies at least one `on` statement. An `on` statement includes an event expression that identifies the event or sequence of events that you are interested in. This is what you want to listen for. An `onload()` statement is not required to specify an `on` statement. If there is no `on` statement, the correlator immediately unloads the monitor.

When the correlator executes an `on` statement, it sets up an event listener for the specified event or sequence of events. After the correlator sets up the event listener, the event listener watches for an event that matches its event expression. When the event listener detects a matching event, the event listener triggers and the correlator executes the action specified in the `on` statement.

For an event listener that is looking for a single instance of an event, this is straightforward. However, the event expression that defines what you are looking for can specify all instances of an event, all instances of a sequence of events, and it can have temporal and logical constraints. This makes the lifecycle of an event listener less straightforward.

For example, consider the following event listener:

```
on all A() success;
```

When the correlator sets up this event listener, it sets up an event template to look for an `A` event. When an `A` event arrives, the correlator does the following:

- Executes the `success()` event listener action.
- Sets up a new event template to look for the next `A` event.

Now consider this event listener:

```
on all A() -> all B() success;
```

Again, suppose that the correlator sets up this event listener and an `A` event arrives. This time the correlator does the following:

1. Sets up an event template to listen for the next `B` event.
2. Sets up an event template to listen for the next `A` event.

This event listener will be active until it is explicitly killed because there will always be an event listener that is looking for the next `A` event.

Additional information about event listener lifecycles is in *Developing Apama Applications in EPL*, "How the correlator executes event listeners".

Events and Event Listeners

Event processing order

As mentioned earlier, contexts allow EPL applications to organize work into threads that the correlator can execute concurrently. When you start a correlator it has a main context. You can create additional contexts to enable the correlator to concurrently process events. Each context, including the main context, has its own input queue. The correlator can process, concurrently, events in each context.

Concurrently, in each context, the correlator

- Processes events in the order in which they arrive on the context's input queue

- Completely processes one event before it moves on to process the next event

When the correlator processes an event within a given context, it is possible for that processing to:

- Send or enqueue an event to a particular channel

  The correlator places the event on the input queue of each context that is subscribed to the specified channel.

- Send or enqueue an event to a particular context or to a sequence of contexts

  The correlator places the event on the input queue of the specified context or on the input queue of each context in the specified sequence.

- Enqueue an event

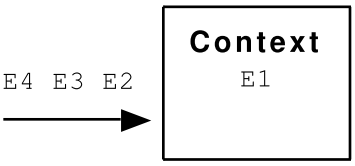
  The correlator places the enqueued event on the special queue just for events generated by the `enqueue` keyword. A separate thread moves these events to the input queue of each public context. This arrangement ensures that if the input queue of a public context is full, the event generated by `enqueue` still arrives on its special queue, and is moved to each appropriate input queue as soon as that queue has room. Active event listeners will eventually receive events that are `enqueue`'d, once those events make their way to the head of the input queue alongside normal events.

- Route an event

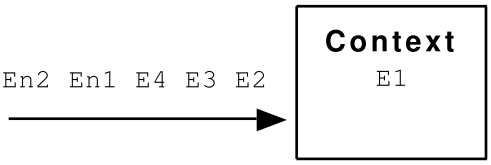
  The correlator places the routed event at the front of that context's input queue. The correlator processes the routed event before it processes the other events in that input queue.

  If the processing of a routed event routes one or more additional events, those additional routed events go to the front of that context's input queue. The correlator processes them before it processes any events that are already on that context's input queue.

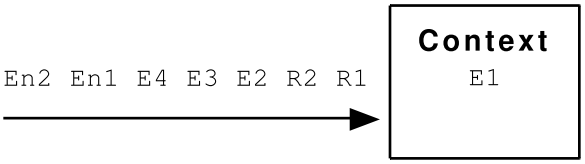
For example, suppose the correlator is processing the `E1` event and events `E2`, `E3`, and `E4` are on the input queue in that order.



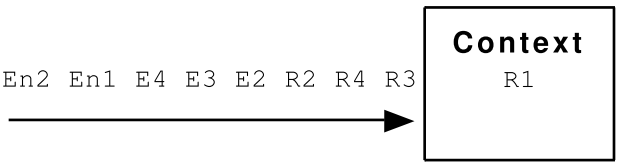
While processing `E1`, suppose that events `En1` and `En2` are created in that order and enqueued. These events go to the special queue for enqueued events. Assuming that there is room on the input queue of each public context, the enqueued events go to to the end of the input queue of each public context:



While still processing `E1`, suppose that events `R1` and `R2` are created in that order and routed. These events go to the front of the queue:

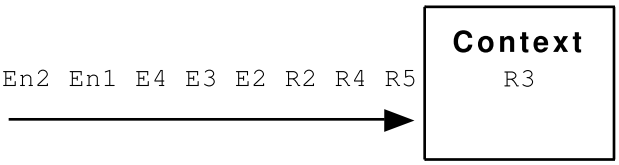


When the correlator finishes processing `E1`, it processes `R1`. While processing `R1`, suppose that two event listeners trigger and each event listener action routes an event. This puts event `R3` and event `R4` at the front of that context's input queue. The input queue now looks like this:



It is important to note that `R3` and `R4` are on the input queue in front of `R2`. The correlator processes all routed events, and any events routed from those events, and so on, before it processes the next routed or non-routed event already on the queue.

Now suppose that the correlator is done processing `R1` and it begins processing `R3`. This processing causes `R5` to be routed to the front of that context's input queue. The context's queue now looks like the following:



See also *Developing Apama Applications in EPL*, "Understanding time in the correlator".

Events and Event Listeners

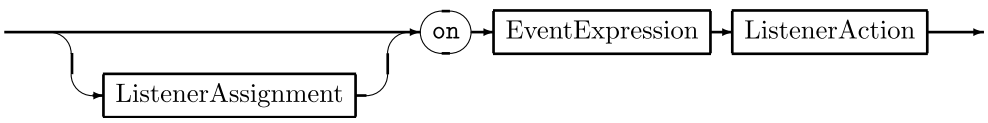
Event expressions

An event expression is a special type of expression that is used with the `on` statement (see "The on statement" on page 125) to define the rules for detecting events of interest and invoking an action when a matching event is detected. In an event expression, you can specify filtering rules based on an event's field values, sequencing rules for events followed by other events, times and time ranges during which an event is of interest, and other rules.

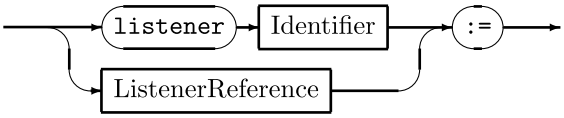
Event expressions should not be confused with ordinary EPL expressions of type `event`. Ordinary EPL expressions of all types are described in "Expressions" on page 130.

The `on` statement is discussed in "The on statement" on page 125, but because the `on` statement and event expressions are so closely related, here are the `on` statement diagrams:

OnStatement



ListenerAssignment

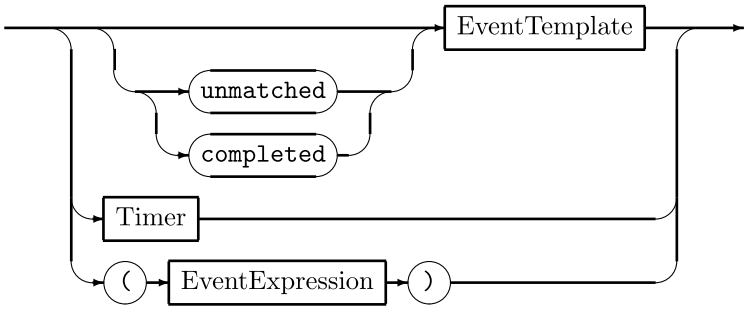


Events and Event Listeners

Event primaries

The event primary is the simplest form of an event expression clause and can be combined with other event primaries and event operators to form more complex event expressions. An event primary can take the following forms:

EventPrimary



Event templates are constructs that allows you to specify filtering or matching criteria based on values of one or more of an event's fields. See "Event templates" on page 81. For convenience, the syntax diagram is repeated here.

EventTemplate

The completed operator

A `completed` event template matches only after all other work is completed. When an event that matches a `completed` template comes into the correlator, the correlator

1. Runs all of the event's normal or `unmatched` event listeners. Normal event templates do not specify the `completed` or `unmatched` keyword.
2. Processes all routed events that result from those event listeners.
3. Triggers the `completed` event listeners.

For example:

```
on all completed A(f < 10.0) {}
```

The unmatched operator

An `unmatched` event template matches against events for which both of the following are true:

- Except for `completed` and `unmatched` event templates, the event is not a match with any other event template currently loaded in the context.
- The event matches the `unmatched` event template.

The correlator processes events as follows:

1. The correlator tests the event against all normal event templates in the context. Normal event templates do not specify the `completed` or `unmatched` keyword. If there are any matches, those event listeners trigger and the correlator executes those event listener actions. If execution of the event listener actions routes any events, the correlator then processes those events.
2. If the correlator does not find a match, the correlator tests the event against all event templates in the context that specify the `unmatched` keyword. If the correlator finds one or more matches, it triggers an event listener for each match found. In other words, if multiple `unmatched` event templates match a given event, they all trigger. The correlator executes the event listener actions defined by the event listeners that trigger. If any events are routed during execution of those actions, the correlator processes the routed events.
3. The correlators tests the event against all event templates in the context that specify the `completed` keyword. If the correlator finds one or more matches, it triggers an event listener for each match found.

Example

For example, suppose you have the following code:

```
on all A("foo",  < 10) : a {
   print "Match: " + a.toString();
   a.count := a.count+1; // count is second field of A
   route a;
}
on all unmatched A(*,*): a {
   print "Unmatched: " + a.toString();
}
on all completed A("foo", *) : a {
   print "Completed: " + a.toString();
}
```

The incoming events are as follows:

```
A("foo", 8);
A("bar", 7);
```

The output is as follows.

```
Match: A("foo", 8)
Match: A("foo", 9)
Unmatched: A("foo", 10)
Completed: A("foo", 10)
Completed: A("foo", 9)
Completed: A("foo", 8)
Unmatched: A("bar", 7)
```

Specify the `unmatched` keyword with care. Be sure to communicate with any others who write event templates. If you are relying on an `unmatched` event template, and someone else injects a monitor that happens to match some events that you expected to match your `unmatched` event template, you will not get the results you expect.

Parenthesized event expressions

Just as with primary and bitwise expressions, EventExpressions can be enclosed in parentheses to control expression evaluation order or to improve readability.

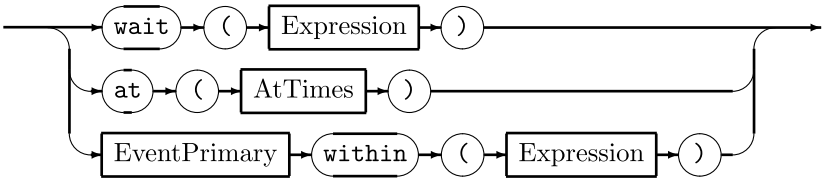
ParenthesizedEventExpression

Event expressions

Timers

Specify a timer with the `wait`, `at`, or `within` keyword.

Timer



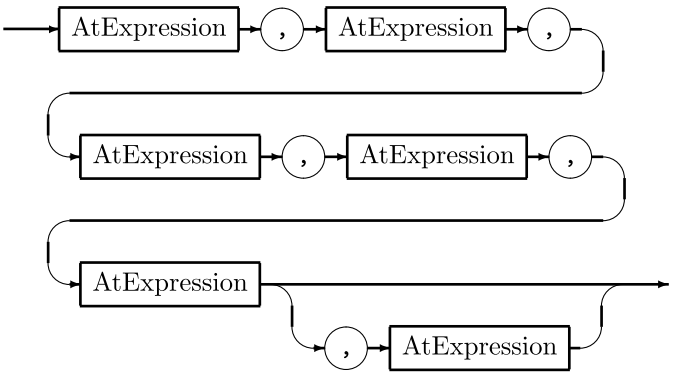
The wait event operator

The `wait` operator can be used to limit the amount of time that an event listener can match an event. The `wait` operator's expression specifies the time in seconds. The result of evaluating the `wait` expression must be of type `float`.

The at event operator

The `at` operator allows triggering of an event listener at a specific time or repeatedly at multiple times, depending on how the series of expressions that follow the `at` operator are constructed.

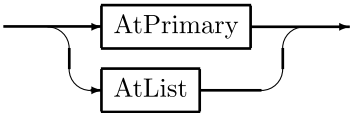
AtTimes



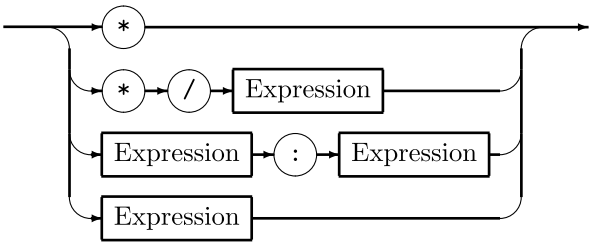
The time specification of the `at` operator consists of either five or six AtExpressions, corresponding to the number of minutes of the hour (0 to 59), hour of the day (0 to 23), day of the month (1 to 31), month of the year (1 to 12), day of the week (0 to 6, 0=Sunday), and seconds respectively.

If the optional number of seconds is omitted, 0 is used.

AtExpression

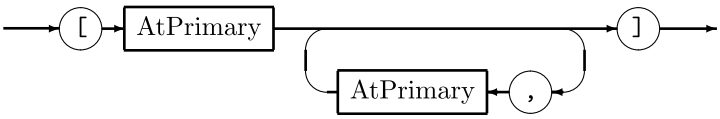


AtPrimary



In the AtPrimary, the `*` operator means that all times (minute, hour, etc.) for the corresponding part of the time specification will match.

AtList

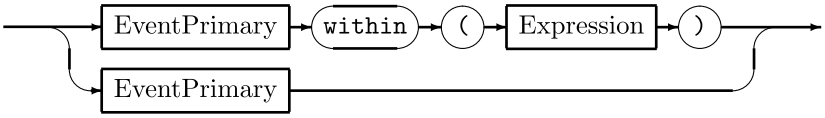


The AtList is a sequence initializer that contains one or more AtPrimary time values separated by commas. See "Sequence variable declarations" on page 157.

The within operator

The `within` operator takes one operand, which is an expression of type `float`, whose value is the number of elapsed seconds from an event primary's activation time that the event primary can be matched. The `within` operator's result type is `boolean`. If the event is matched before the specified time has elapsed, the `within` operator's result is `true`. When the time has elapsed and the event has not been matched, the `within` operator's result is `false`.

WithinExpr

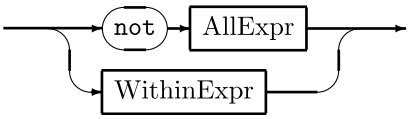


Event expressions

The not Operator

The `not` operator specifies logical negation.

NotExpr

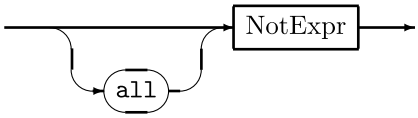


Event expressions

The all Operator

When the `all` operator appears before an event template, when that event template finds a match, it continues to watch for subsequent events that also match the template.

AllExpr



Consider the following event expression:

```
all A -> B
```

This event listener would match on every `A` and the first `B` that follows it. The way this works is that upon encountering an `A`, the correlator creates a second event listener to seek the next `A`. Both event listeners would be active concurrently; one looking for a `B` to successfully match the sequence specified, the other initially looking for an `A`. If more `A`s are encountered the procedure is repeated; this behavior continues until either the monitor or the event listener are explicitly killed.

Consider the following sequence of incoming events:

```
C1 A1 F1 A2 C2 B1 D1 E1 B2 A3 G1 B3
```

With these input events, `on all A() -> B()` would return the following:

{`A1`, `B1`}, {`A2`, `B1`} and {`A3`, `B3`}.

Note that `all` is a unary operator and has higher precedence than `->`, `or` and `and`.

Event expressions

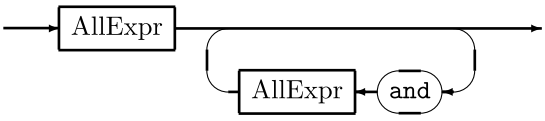
The and, xor, and or logical event operators

The logical operators `and`, `or`, and `xor` are similar to the corresponding operators in primary and bitwise expressions, but do not have quite the same precedence.

The and event operator

The `and` operator specifies logical intersection.

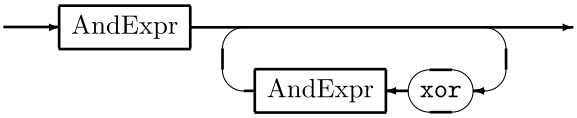
AndExpr



The xor event operator

The `xor` operator specifies logical exclusive or.

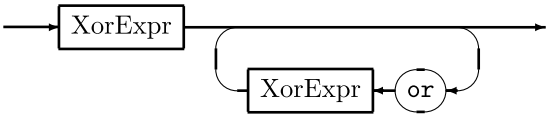
XorExpr



The or event operator

The `or` operator specifies logical or.

OrExpr

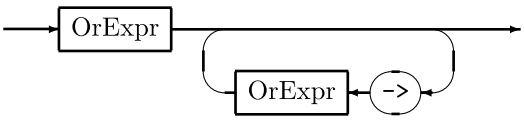


Event expressions

The followed-by event operator

The followed-by operator `->` is a Boolean operator that takes left and right operands, both event expressions. The followed-by operator waits for the left operand to become true and then waits for the right operand to become true. When both are true then the result value is true. If either becomes false, then the result value is false.

FollowedByExpr



Event expressions

Event expression diagram

EventExpression

Event expressions

Event expression operator precedence

The following table lists the event expression operators in order by their precedence, from lowest to highest. See "Expression operator precedence" on page 143 for a corresponding table of primary and bitwise expression operator precedence.

Operation	Operator
Logical negation	`not`
All	`all`
Logical intersection	`and`
Logical exclusive or	`xor`
Logical union	`or`
Followed-by	`->`

For example, the following expression:

```
on all A()or B() and not C() -> D()
```

is equivalent to this expression:

```
on (
   (all A() )
   or
   (B() and (not C() ))
   ) -> D()
```

Event expressions

Event channels

Adapter and client configurations can specify the channel to deliver events to. A channel is a string name that contexts and receivers can subscribe to in order to receive particular events. In EPL, you can send an event to a specified channel. Sending an event to a channel delivers it to any contexts that are subscribed to that channel, and to any clients or adapters that are listening on that channel.

You can use the `com.apama.Channel` type to send an event to a channel or context. The `Channel` type holds a string or a context. When it holds a string an event is sent to the channel that has that name. When it holds a context an event is sent to that context.

Events and Event Listeners

Chapter 4: Monitors

An EPL file is a file that contains the source text for an optional package specification and one or more event declarations or monitor definitions. A file can consist entirely of event declarations without any monitors.

A monitor is a group of related variable declarations and actions. The monitor is the primary unit of execution in EPL. An action is a group of related variable declarations and statements. An action can either be part of a monitor or part of an event declaration.

The executable statements (except for global variable intializers) are always inside an action. An action can be either a subprogram or a function. The difference is that a function has a return value and a subprogram does not.

Each file is injected in either whole or not at all; if some parts compile validly but others do not, nothing is injected and an error is returned. Injecting can also return warnings about the code injected. For example, use of keywords that may be reserved in the future.

Monitor lifecycle

EPL programs are compiled and run (executed) by the Apama correlator. The unit of program execution is a monitor, which starts executing in the monitor's `onload()` action. To execute a monitor, you load (inject) it into the correlator. The correlator then does the following:

1. Compiles the monitor's source text
2. If no errors are detected, creates the main monitor instance along with its global variables
3. Invokes the monitor instance's `onload()` action

When the `onload()` action has executed to completion (that is, the control path reaches the closing curly brace of the `onload()` action), if the monitor instance has event listeners or streaming networks, then it remains active but in a suspended state.

The correlator calls the monitor instance's event listeners whenever it detects events that match the event listeners' event expressions.

A monitor instance terminates when one of the following events occurs:

- The monitor instance executes a `die` statement in one of its actions.
- A runtime error condition is raised.
- The monitor is terminated externally (for example, with the `engine_delete` utility.
- The monitor instance has executed all its code and there are no remaining listeners or streaming networks. This will occur rapidly if the `onload()` action does not create any.

When a monitor instance terminates, the correlator does the following:

1. Invokes the monitor instance's `ondie()` action, if it is defined.
2. If the monitor instance that is terminating is the last active instance of that monitor, the correlator also does the following:
   - Invokes the monitor's `onunload()` action if it is defined.
   - Removes the monitor's code from the correlator.
   - Frees all the monitor's resources.

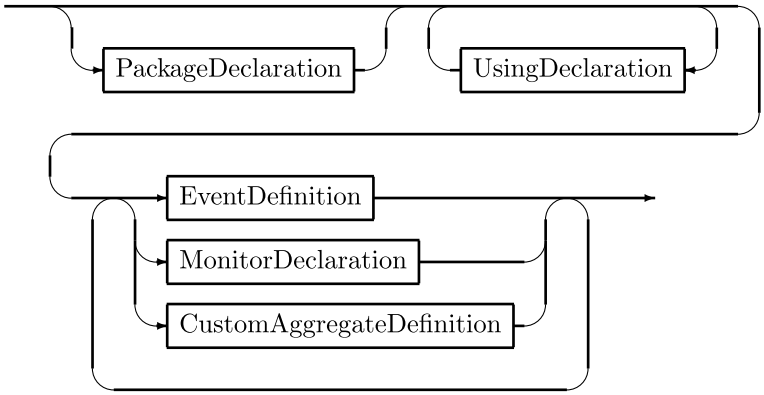
To summarize, consider that when a monitor spawns monitor instances, there is a set of monitors that includes the original monitor instance and any spawned monitor instances. As the monitor instances in this set terminate, the correlator calls the `ondie()` action, if it is defined, for each monitor instance that terminates. When the last monitor instance in the set terminates, the correlator also calls the `onunload()` action. Thus, the correlator calls `ondie()` once for each monitor instance in the set, and calls `onunload()` only once for the entire set.

Monitors

Programs

An EPL program contains an optional package declaration, event declarations and/or monitor declarations.

Program

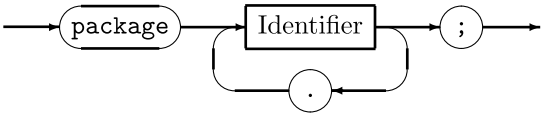


Monitors

Packages

A package declaration provides a scope for events and/or monitors.

PackageDeclaration

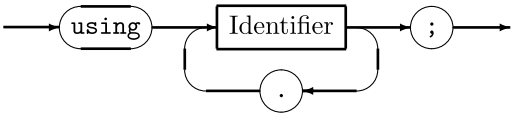


Monitors

The using declaration

The `using` declaration lets you use a type in a package other than the package the type was defined in without having to specify the fully qualified name of the type.

UsingDeclaration



Insert a `using` declaration, after the optional `package` declaration and before any other declarations, that specifies the fully qualified name of the type. For example:

```
using com.myCorporation.custom.myCustomAggregate;
```

You can specify multiple `using` declarations in a file.

In a file, you cannot specify two `using` declarations that bring in types that have the same base name. See also "Name Precedence" on page 185.

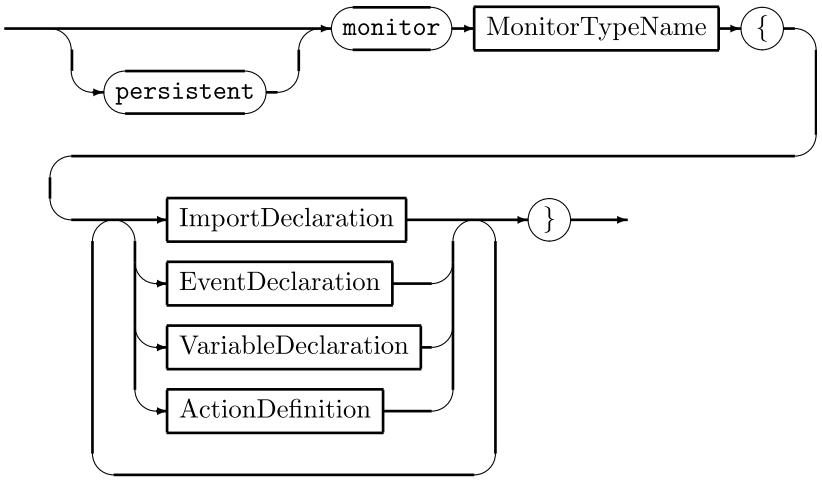
You cannot specify a `using` declaration for named objects such as monitors, JMon monitors, and namespaces.

Monitors

Monitor declarations

Specify `persistent` when you want a persistence-enabled correlator to save the state of the monitor in a recovery datastore on disk. In a monitor, import declarations, event declarations, variable declarations, and action definitions can be freely mixed in any order.

MonitorDeclaration

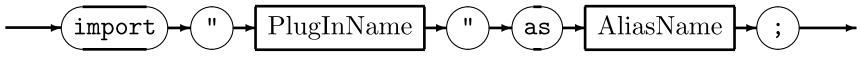


Monitors

The import declaration

The import declaration loads a plug-in library and makes it available to an EPL program. Plug-in libraries are shared libraries on Linux and UNIX systems and Dynamic Link Libraries on Windows systems.

ImportDeclaration



Rule components

The PlugInName is the name of the plug-in's library. On Linux and UNIX systems, the library is loaded from a `libPlugInName.so` file located in one of the directories listed in the environment variable `LD_LIBRARY_PATH`. On Windows, the library is loaded from a `PlugInName.dll` file located in the `bin` folder

The PlugInName is a library filename, not a full file-path, and is not allowed to contain any of the characters used as directory or device separators (forward slash, colon, or backslash). The AliasName is an identifier for use in the EPL program when you call the library's actions.

Example

For example, to call a plug-in action `foo()` in the plug-in library wfffftl, you would write the following:

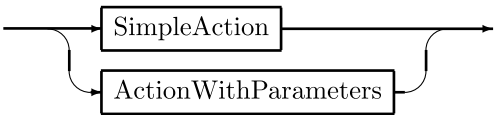
```
monitor m {
   import "wfffftl" as fft;
   action onload()
   {
      sequence <float> data := [];
      fft.foo (data);
   }
}
```

Monitors

Monitor actions

Monitors can have two forms of actions: simple actions and actions with parameters and/or return values.

ActionDefinition



See also

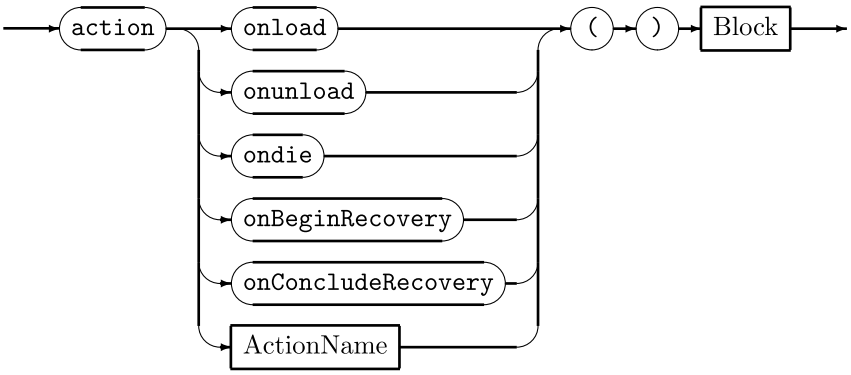
- "SimpleActions" on page 101
- "About onload()" on page 102
- "About ondie()" on page 102
- "About onunload()" on page 102
- "Actions with parameters" on page 102

Monitors

SimpleActions

A simple action has a name and a body consisting of a block. The body contains the executable code of the action. There are no parameters.

SimpleAction



Rule components

The action names `onload()`, `onunload()`, `ondie()`, `onBeginRecovery()`, and `onConcludeRecovery()` are special. These actions are invoked automatically when certain events in a monitor's life cycle occur.

A block must follow the action name. Note that there are no formal parameters in this form of action definition and the action cannot return a value.

About onload()

The `onload()` action is invoked immediately after a monitor has been loaded. This action must be present in every monitor.

About ondie()

The `ondie()` action, if present, is invoked by the correlator when a monitor instance terminates.

About onunload()

The `onunload()` action, if present, is invoked by the correlator after all instances of a monitor have terminated, just before the last monitor instance is unloaded.

About onBeginRecovery()

The `onBeginRecovery()` action, if present, is invoked by the correlator during recovery of a persistence-enabled correlator. The correlator executes `onBeginRecovery()` on monitors and any live events after it reinjects source code and restores state in persistent monitors.

About onConcludeRecovery()

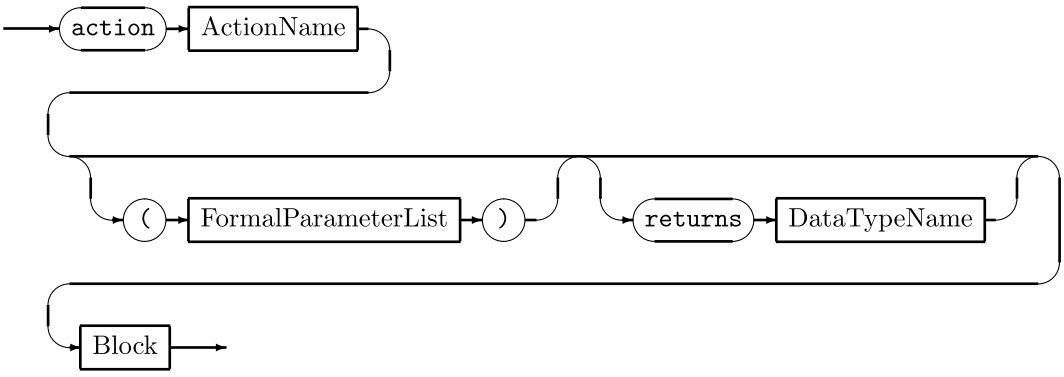
The `onConcludeRecovery()` action, if present, is invoked by the correlator during recovery of a persistence-enabled correlator. The correlator executes `onConcludeRecovery()` on monitors and any live events before it begins to send clock ticks.

Monitor actions

Actions with parameters

An action can take an optional list of parameters.

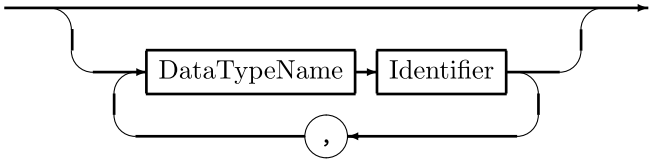
ActionWithParameters



Formal parameters

The FormalParameterList is a comma-separated list of type name and identifier pairs.

FormalParameterList



The identifier is the name of a parameter variable that will be bound to a copy of the value of an expression specified by the caller (that is, the value passed by the caller) when the action is invoked. The number and type of actual parameters passed by a caller must match those listed in the action's formal parameters.

The scope of a parameter variable is the statement or block that forms the action body. Parameter variables are very similar to an action's local variables.

Action return value

If you specify a `returns` clause, then the action must return a value whose type matches that specified in the `returns` clause. You specify the return value by using a `return` statement and result expression within the action. Every control path (see "Transfer of control statements" on page 127) within the action body must lead to a `return` statement with a result expression of the correct type.

Action body

After the `returns` clause (or after the formal parameters if there is no `returns` clause), a statement forms the action body. The action body can be a single statement or a block.

Within the action body, you use the parameter variable names to obtain the values that are passed to the action by its caller.

Monitor actions

Contexts

Contexts allow EPL applications to organize work into threads that the correlator can concurrently execute. In EPL, `context` is a reference type. When you create a variable of type `context`, or an event field of type `context`, you are actually creating an object that refers to a context. The context might or might not already exist. You can then use the context reference to spawn to the context or send an event to the context. When you spawn to a context, the correlator creates the context if it does not already exist.

When you start a correlator it has a single main context. You can then create additional contexts. A context consists of the following:

- One or more monitor instances. Except, the main context exists even if it does not contain any monitor instances.
- An event input queue.
- Listeners that belong to the contained monitor instances.

The correlator maintains event definitions and monitor definitions outside contexts. This lets all contexts share the same event and monitor definitions.

Instances of the same monitor can exist in multiple contexts. Each monitor instance can belong to only one context.

A context has the following properties:

- Name — A string that you specify when you create the context. This name does not need to be unique.
- ID — The correlator assigns a unique integer.
- `receiveInput` flag — A Boolean value that indicates whether the context can receive external input events on the default channel, which is the empty string (`""`).

  A value of `true` lets the context receive external events on the default channel; this is a public context. A value of `true` is equivalent to a subscription to the default channel; there is no requirement for a monitor instance in this context to subscribe to the default channel.

  A value of `false` indicates a private context that does not receive external events on the default channel. This is the default.

  Note that the main context is public.
- Channel subscriptions — A context is subscribed to the union of the channels each of the monitor instances in that context is subscribed to. This is a property of the monitor instances running in a context and is not accessible by means of the context reference object.

You can spawn to other contexts. When the last monitor instance in a context terminates, that context stops doing work and stops consuming resources until you spawn another monitor instance to it.

In a context, when you route an event the event goes to the front of that context's input queue. You can route events only within a context.

You can send an event to a particular context. When you do this, the event goes to the end of the specified context's input queue. The correlator processes it after it processes any other events that are already on the context's input queue. See "The send . . . to statement" on page 119.

You can use a context as part of the key for a dictionary. You can route an event that contains a `context` field. You cannot parse a context. Context objects are immutable reference objects.

Upon injection, each monitor's initial instance runs in the main context. You must explicitly create additional contexts. Conceptually, a context is like a correlator but with the following differences:

- All contexts share the same namespace, and thus share all monitor and event definitions that have been injected.
- A monitor instance must have a context reference to pass an event to that context.
- There is one enqueued events queue for all contexts. When you specify the `enqueue` command (not the `enqueue event to context` command), the enqueued event goes to the special queue for enqueued events. The correlator then places the event on the input queue of each public context. The correlator ensures that an enqueued event always arrives on the appropriate input queue(s). An `enqueue` operation never blocks. However, if the input queue of a context is full and the enqueued events queue gets very large, the result can be an `unbounded memory usage` error.
- Execution of Java is allowed in only the main context.
- The `engine_receive` utility receives events from all contexts or it can be configured to receive events from only specified channels.

- The `engine_send` utility sends events to all public contexts or to the contexts that are subscribed to the channels it is configured to send events on.

For information about creating a context, see "context" on page 43 in "Types" on page 17.

For a monitor instance to interact with the EPL in another context, the monitor instance must have a reference to that context. A monitor instance can obtain a reference to another context in only the following ways:

- By creating the context.
- By receiving a context reference, which must be of type `context`. A monitor instance can receive this reference by means of a sent, routed or enqueued event, or a spawn operation.

If a monitor instance that creates a context does not send a context reference outside itself, no other context can send events to that context, except by means of correlator plug-ins. This affords some degree of privacy for the context.

A `context` object (a context reference) does not do anything. It is simply the target of the following:

- `spawn action_identifier() [( argument_list)] to context_expression;`

  See "The spawn action to context statement" on page 121.

- `send event_expression to context_expression;`

  See The send . . . to statement.

You can create any number of contexts. Creating a context just allocates an ID and creates a small object.

Monitors

Plug-ins

EPL can be extended through the use of plug-ins, which are modules written either in C or C++ and loaded dynamically into the EPL runtime with the `import` statement. Plug-in modules are invoked in exactly the same way as actions in an EPL event.

See *Developing Apama Applications in EPL*, Using correlator plug-ins in EPL".

Monitors

Garbage collection

EPL, like languages such as Java or C#, relies on garbage collection. Intermittently, the correlator analyses the events that have been allocated, including dictionaries, sequences, closures and streaming networks, and allows memory used by events that are no longer referencable to be re-used. Thus, the actual memory usage of the correlator might be temporarily above the size of all live objects. While running EPL, the correlator might wait until a listener or `onload()` action completes before performing garbage collection. Therefore, any garbage generated within a single action or listener invocation might not be disposed of before the action/ listener has completed. It is thus advisable to limit individual actions/listeners to performing small pieces of work. This also aids in reducing system latency.

The cost of garbage collection increases as the number of events a monitor instance creates and references increases. If latency is a concern, it is recommended to keep this number low, dividing the working set by spawning new monitor instances if possible and appropriate. Reducing the number of event creations, including string operations that result in a new string being created, also helps to reduce the cost of garbage collection. The exact cost of garbage collection could change in future releases as product improvements are made.

Monitors

Chapter 5: Aggregate Functions

In stream queries, you can specify aggregate functions in the select clause.An aggregate function calculates a single value across all items currently in the window. EPL provides a number of commonly used aggregate functions. If a supplied aggregate function does not meet your needs, you can define a custom aggregate function.

Built-in aggregate functions

EPL provides the following built-in aggregate functions. In the table, the argument names, for example, *value* and *weight*, are placeholders for expressions. Additional information about some of these functions follows the table.

Aggregate Function	Argument(s)	Returns	Result Description
`avg(value)`	`decimal` or `float`	`decimal` or `float`	The arithmetic mean of the values in the window. The `avg()` and `mean()` functions do exactly the same thing. They are aliases for each other.
`count()`	-	`integer`	The number of items in the window, including any NaN items
`count (predicate)`	`boolean`	`integer`	The number of items for which the argument is true
`count(value)`	`decimal` or `float`	`integer`	The number of items where the `decimal` or `float` value is not NaN
`first(value)`	`decimal` or `float`	`decimal` or `float`	The earliest value in the window being aggregated over
`last(value)`	`decimal` or `float`	`decimal` or `float`	The latest value in the window being aggregated over
`max(value)`	`decimal` or `float`	`decimal` or `float`	The maximum value

Aggregate Function	Argument(s)	Returns	Result Description
`mean(value)`	`decimal` or `float`	`decimal` or `float`	The arithmetic mean of the values in the window. The `mean()` and `avg()` functions do exactly the same thing. They are aliases for each other
`min(value)`	`decimal` or `float`	`decimal` or `float`	The minimum value
`nth(value,index)` `nth(value,0)` returns the same item as `first(value)`.	`decimal`, `integer` or `float, integer`	`decimal` or `float`	The value of the specified `decimal` or `float` item in the *`index`* position, starting with the earliest item in the window (item 0) and moving toward the latest item.
`prior(value,index)` `prior(value,0)` returns the same item as `last(value)`.	`decimal`, `integer` or `float, integer`	`decimal` or `float`	The value of the specified `decimal` or `float` item in the *`index`* position, starting with the most recent item in the window (item 0) and moving toward the earliest item.
`stddev(value)`	`decimal` or `float`	`decimal` or `float`	The standard deviation of the values
`stddev2(value)`	`decimal` or `float`	`decimal` or `float`	The sample standard deviation of the values
`sum(value)`	`decimal`, `float` or `integer`	`decimal`, `float` or `integer`	The sum of the values
`wavg(value,weight)`	`decimal`,`decimal` or `float`,`float`	`decimal` or `float`	The weighted average of the values where each value is weighted by the corresponding weight

Calculations by the built-in aggregate functions might be affected by underflow and overflow. For example, adding a very large number to the collection that the `sum()` function operates on, then adding a very small number, and then removing the very large number will probably result in `0`, and not the very small number. Just adding the very small number would result in behavior that you would expect. The overflow and underflow characteristics are as defined for IEEE 64-bit floating point numbers.

Positional functions

For the `first()`, `last()`, `nth()`, and `prior()` functions, all values (`NaN`, `+-∞` and so on) are treated the same, and position in the window is the only thing that matters.

Overloaded functions

The `sum()` function is overloaded. You can specify a `decimal`, `float` or `integer`. The return type matches the argument type.

The `avg()`, `first()`, `last()`, `max()`, `mean()`, `min()`, `nth()`, `prior()`, `stddev()`, and `stddev2()` functions are overloaded. You can specify a `decimal` or a `float`. The return type matches the argument type.

The `count()` function is overloaded. You can specify a `boolean` , `decimal`, `float` or no argument. The return type is an `integer`.

The `wavg()`function is overloaded. You can specify a `decimal, decimal` or a `float, float` combination. The return type will be a `decimal` or a `float`, respectively.

See also

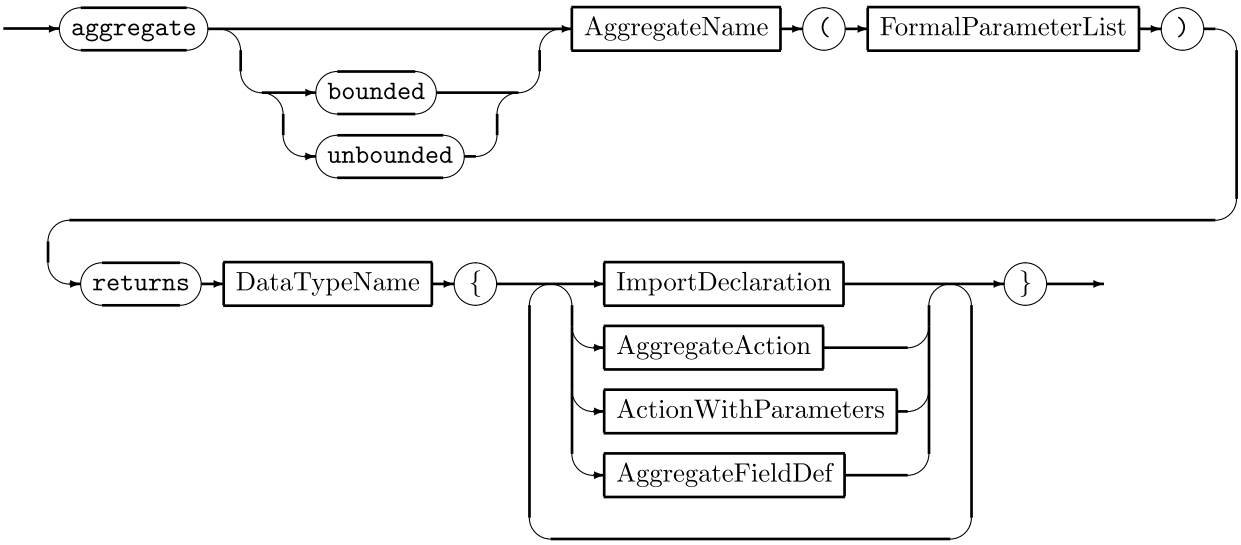
Developing Apama Applications in EPL, "Working with streams and stream queries", "Aggregating items in projections".

Aggregate Functions

Custom aggregates

In a stream query, you can specify an aggregate function in the select clause. If one of the supplied aggregate functions does not meet your needs, you can define a custom aggregate function for use in a select clause.

CustomAggregateDefinition



Rule components

You define custom aggregate functions outside of an event or a monitor and the aggregate function's scope is the package in which you declare it. To use custom aggregate functions in monitors in other packages, specify the aggregate function's fully-qualified name, for example:

```
from a in all A() select com.myCorporation.custom.myCustomAggregate(a)
```

Alternatively, you can specify a `using` statement. See "The using declaration" on page 99.

Specify `bounded` when you are defining a custom aggregate function that will work with only a bounded window. That is, the query cannot specify `retain all`. Specify `unbounded` when you are defining a custom aggregate function that will work with only an unbounded window. That is, the query must specify `retain all`. Do not specify either `bounded` or `unbounded` when you are defining a custom aggregate function that will work with either a bounded or an unbouded window.

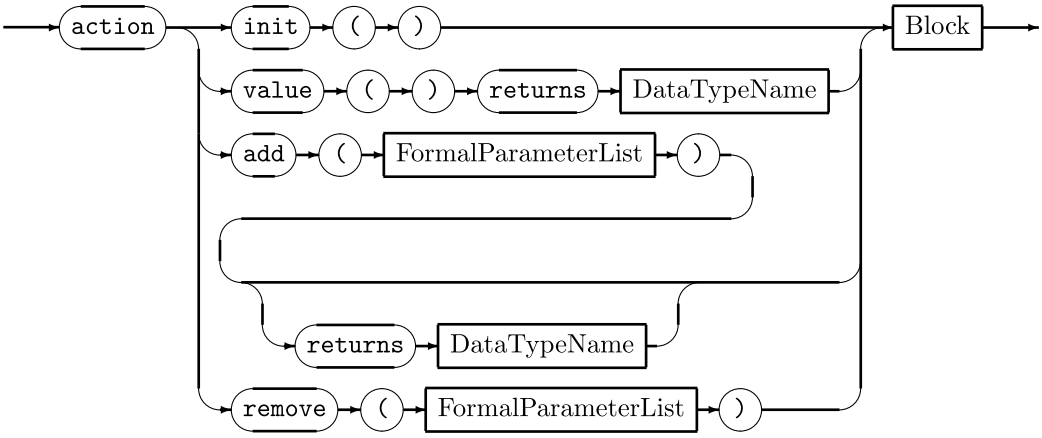
The name of a custom aggregate function must be unique within a package; you cannot overload it or define an event or monitor with the same name as an aggregate function.

The FormalParameterList is zero or more comma-separated type/name pairs. Each pair indicates the type and the name of an argument that you are passing to the aggregate function. For example, `(float price, integer quantity)`.

The DataTypeName must be an EPL type. This is the type of the value that your aggregate function returns.

The body of a custom aggregate function can contain fields that are specific to one instance of the custom aggregate function and actions to operate on the state.

AggregateAction



Rule components

In a custom aggregate function, the `init()`, `add()`, `remove()` and `value()` actions are special. They define how stream queries interact with custom aggregate functions.

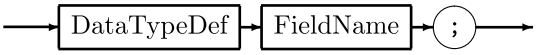
- `init()` — If a custom aggregate function defines an `init()` action it must take no arguments and must not return a value. The correlator executes the `init()` action once for each new aggregate function instance it creates in a stream query.
- `add()` — A custom aggregate function must define an `add()` action. The `add()` action must take the same ordered set of arguments that are specified in the custom aggregate function signature. That is, the names, types, and order of the arguments must all be the same. The correlator executes the `add()` action once for each item added to the set of items that the aggregate function is operating on.
- `remove()` — A bounded aggregate function must define a `remove()` action. An unbounded aggregate function must not define a `remove()` action. If you do not specify either `bounded` or `unbounded`, the `remove()` action is optional. The `remove()` action must take the same ordered set of arguments as the `add()` action, followed by an argument of the type returned by add(), if any, and

must not return a value. The correlator executes the `remove()` action once for each item that leaves the set of items that the aggregate function is operating on.The value that `remove()` is called with is the same value that `add()` was called with.

- `value()` — All custom aggregate functions must define a `value()` action. The `value()` action must take no arguments and its return type must match the return type in the aggregate function signature. The correlator executes the `value()` action once per batch per group and returns the current aggregate value to the query.

Custom aggregate functions can declare other actions, including actions that are executed by the above named actions. A custom aggregate function cannot contain a field whose name is `onBeginRecovery`, `onConcludeRecovery`, `init`, `add`, `value`, or `remove`, even if, for example, the custom aggregate function does not define a `remove()` action.

AggregateFieldDef



Rule components

In the body of a custom aggregate function, you can define fields that are specific to the custom aggregate instance they are in.

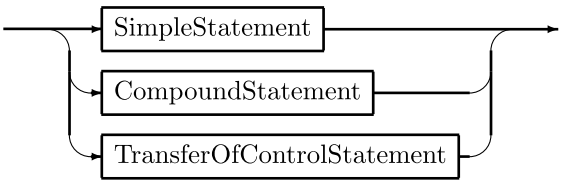
Aggregate Functions

Chapter 6: Statements

Sequences of EPL statements define the steps that are performed by a program. They are executed in the order they are written — sequentially from top to bottom and left to right within a statement block. (For expressions, the evaluation order is affected by parentheses, associativity, and operator precedence.)

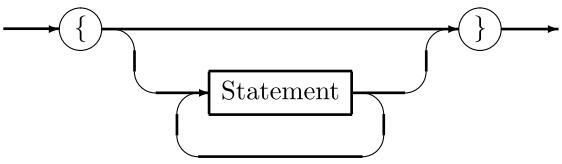
The order in which statements are executed is called the flow of control or the control path. Some statements can contain other statements enclosed within their structure and can be used to execute statements conditionally, thus altering the normal control path. You can use the `break`, `continue`, and `return` statements to change the normal control path.

Statement



A block is zero or more statements enclosed in curly braces.

Block

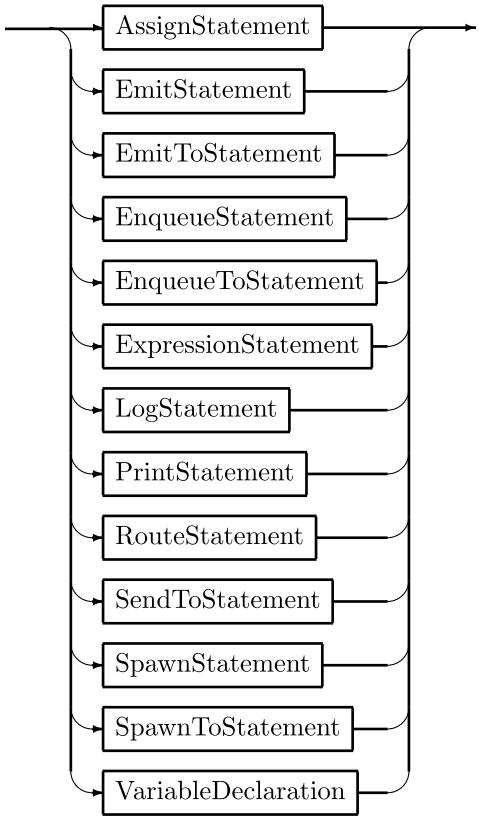


A block can be used wherever a single statement can be used. Variables declared in a block are referenceable only in the block in which they are declared, and only in statements that come after the variable's declaration.

Simple statements

Simple statements are statements that do not enclose other statements or statement blocks and that do not cause a transfer of control. They are executed in the order they are written.

SimpleStatement

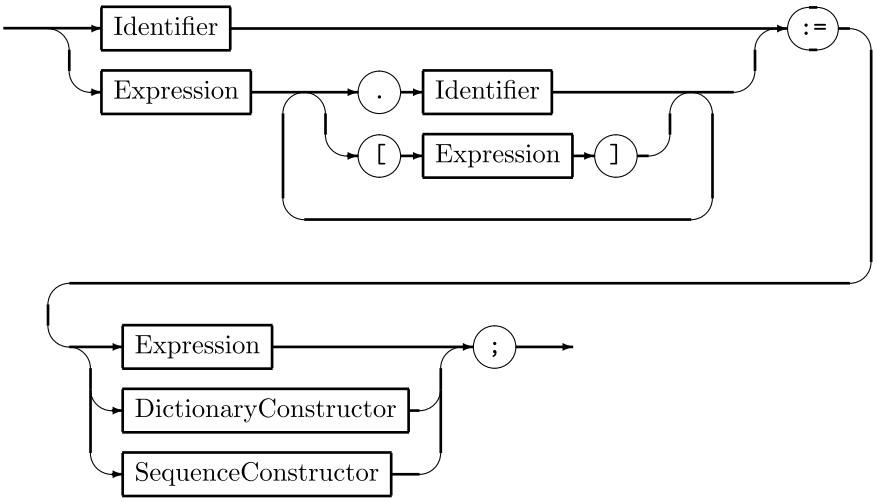


Statements

The assignment statement

The assignment statement binds a value to a variable. The value is determined by evaluating the expression on the right side of the assignment operator :=. The result type of the expression must match the type of the variable. For variables of the reference types, the same value can be bound to more than one variable. See "Reference types" on page 37.

AssignStatement



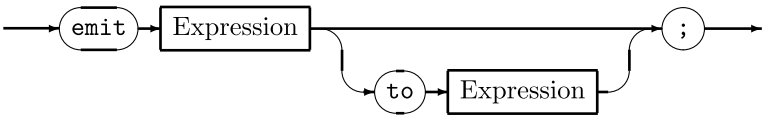
Simple statements

The emit statement

The `emit` statement publishes an event to a named channel of the correlator's output queue. If a channel name is not specified, then the event goes to the default channel whose name is the empty string (`""`). External receivers get events on the default channel only if they are subscribed to all channels.

Note: The `emit` statement will be deprecated in a future release. Use the `send` statement instead. See "The send . . . to statement" on page 119.

EmitStatement



Rule components

The first expression is an expression whose result type is either an event type or string. If the type is string, then the value of the string is assumed to be in the same format as that produced by the event's `toString()` method.

The expression following the keyword `to` must be of type `string` and is the name of the channel to which the event will be sent.

The `emit` method dispatches events to external registered event receivers. That is, the emit statement causes events to go out of the correlator. Active event listeners will not receive events that are emitted.

Events are emitted onto named channels. For an application to receive events from the correlator it must register itself as an event receiver and subscribe to one or more channels. Then if events are emitted to those channels they will be forwarded to it.

Channels effectively allow both *point-to-point* message delivery as well as through *publish-subscribe*. Channels can be set up to represent topics. External applications can then subscribe to event messages of the relevant topics. Otherwise a channel can be set up purely to indicate a destination and have only one application connected to it.

You cannot emit an event whose type is defined inside a monitor.

You cannot emit an event that has a field of type `action`, `chunk`, `listener`, or `stream`.

When you emit an event type that has a dictionary field, the items in the dictionary are sorted in ascending order of their key values.

Simple statements

The enqueue statement

The `enqueue` statement sends an event to the back of the input queue of each public context. The expression is evaluated and the resulting event is sent to all input queues of public contexts. If an input queue is full, then the enqueued event is saved on a temporary holding queue until the input queue has room for it. There is one temporary holding queue for all contexts. When an input queue is full, processing in the context that enqueued the event blocks until the enqueued event arrives on all public input queues.

EnqueueStatement

Rule components

Note that enqueued events are processed in the order they are enqueued.

The expression's result type must be an `event` type or `string`. When it is a string, the correlator parses it as an event.

Enqueued events are put on the back of the input queue, behind any externally sourced events already queued.

You cannot enqueue an event whose type is defined inside a monitor.

You cannot enqueue an event that has a field of type `action`, `chunk`, `listener`, or `stream`.

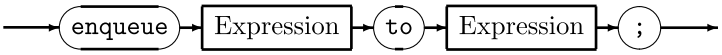
Simple statements

The enqueue . . . to statement

The `enqueue...to` statement sends an event to a context you identify.

Note: The `enqueue...to` statement is superseded by the `send...to` statement. The `enqueue...to` statement will be deprecated in a future release. Use the `send...to` statement instead. See .

Enqueue . . ToStatement



Rule components

The result type of the first Expression must be `event`. It cannot be a string representation of an event.

The result type of the second Expression can be one of the following:

- `context` — The `enqueue...to` statement sends an event to the back of the input queue of the specified context. The expression is evaluated and the resulting event is sent to the input queue of only the specified context.
- `sequence<context>` — The `enqueue...to` statement sends a copy of the event to the back of the input queue of each context in the specified sequence. The expression is evaluated and the resulting event is sent to the input queue of all the contexts in the sequence.

  For example:

  ```
  sequence <context> ctxs := [ c1, c2, c3 ];
  Ping ping = Ping();
  enqueue ping to ctxs;
  ```

You cannot enqueue an event to a `com.apama.Channel` object that contains a context. You cannot enqueue an event to a dictionary of contexts. However, it is a common pattern to enqueue to a sequence generated by `dictionary.values()`. For example:

```
enqueue x to d.values;
```

If the target context's input queue is full the sending context blocks and waits for space on the queue unless doing so would cause a deadlock. See "Deadlock avoidance when parallel processing" in *Developing Apama Applications in EPL*.

Note that enqueued events are processed in the order they are enqueued. Enqueued events are put on the back of the input queue, behind any externally sourced events already queued.

You must create the context before you enqueue an event to the context. You cannot enqueue an event to a context that you have declared but not created. For example, the following code causes the correlator to terminate the monitor instance:

```
monitor m {
   context c;
   action onload()
   {
      enqueue A() to c;
   }
}
```

If you enqueue an event to a sequence of contexts and one of the contexts has not been created first then the correlator terminates the monitor instance. For details, see "Enqueuing to contexts" in *Developing Apama Applications in EPL*.

Enqueueing an event to a sequence of contexts is non-deterministic. For details, see "Enqueuing an event to a sequence of contexts" in *Developing Apama Applications in EPL*.

In an `enqueue...to` statement, you cannot enqueue an event that has a field of type `action`, `chunk`, `listener`, or `stream`.

Simple statements

The expression statement

An expression that does not return a value can be used as a statement.

ExpressionStatement

One would use an expression statement if the expression has desired side effects. For example, an action or method call can be used in this way.

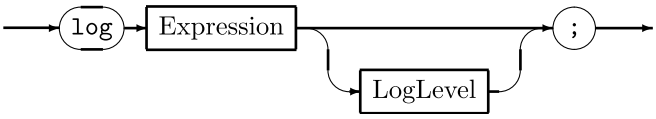
To be used as a statement, an expression must return nothing.

Simple statements

The log statement

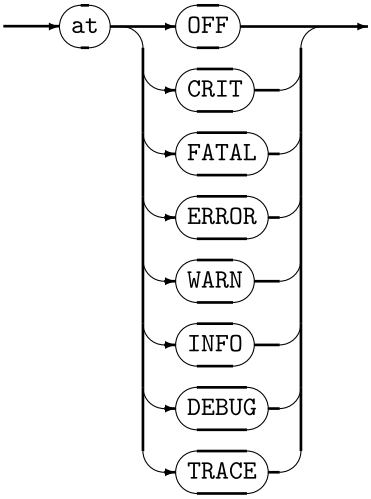
The `log` statement writes messages and accompanying date and time information to the correlator's log file, if one was specified when the correlator was started.

LogStatement



If there is no log file, then the message is written to the correlator's standard output stream `stdout`.

LogLevel



The Expression result must be of type `string`. The value is written only if the current logging level in effect is a priority equal to or higher than the `logLevel` specified in the `log` statement, with the exception of `OFF`. If you do not specify a level, `CRIT`, the highest priority level, is used. At a `logLevel` equal to `OFF`, only logs explicitly set to this level will be written. For details, see "Logging and printing" in *Developing Apama Applications in EPL*.

Example

For example:

```
log "Your message here" at INFO;
```

This EPL statement produces a log message that looks like this:

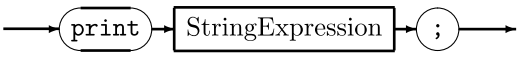
```
2010-07-11 09:08:49.200 INFO [3716] - MyMonitor[1] Your message here
```

Simple statements

The print statement

The `print` statement writes textual messages followed by a newline to the correlator's standard output stream — `stdout`. The Expression result must be of type `string`.

PrintStatement



Example

For example:

```
print "Your message here.";
```

This EPL statement produces output that looks like this:

```
Your message here.
```

The `print` statement is less useful for reporting diagnostic information than the `log` statement, as it does not contain any information about the time or origin of the message, and cannot be turned off by changing the log level.

Simple statements

The route statement

The `route` statement evaluates the expression and then sends the resulting event to the front of the current context's input queue.

RouteStatement

The expression's result type must be an event. The event is processed only within the same context that executes the `route` statement.

Routed events are put on the input queue, ahead of any externally sourced events, and ahead of any previously routed events that have not yet been processed. For more details, see "Event processing order" on page 87.

The `isExternal()` method returns false for routed events.

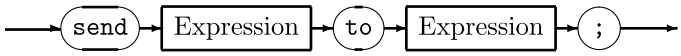
You cannot route an event whose type is defined in a monitor. You cannot route an event that has a field of type `action`, `chunk`, `listener`, or `stream`.

Simple statements

The send . . . to statement

The `send...to` statement sends an event to the channel, context, sequence of contexts, or `com.apama.Channel` object that you specify.

Send...ToStatement



Rule components

The result type of the first Expression must be an event. It cannot be a string representation of an event.

The result type of the second Expression can be one of the following:

- `string` — The `send...to` statement sends the event to the specified channel. All contexts and external receivers subscribed to that channel receive the event. If there are no subscribers to the specified channel or if no receivers are listening on the specified channel then the event is discarded.

- `context` — The `send...to` statement sends the event to the back of the input queue of the specified context. The event expression is evaluated and the resulting event is sent to the input queue of only the specified context.
- `sequence<context>` — The `send...to` statement sends a copy of the event to the back of the input queue of each context in the specified sequence. The event expression is evaluated and the resulting event is sent to the input queue of each context in the sequence.

  For example:

  ```
  sequence <context> ctxs := [ c1, c2, c3 ];
  Ping ping = Ping();
  send ping to ctxs;
  ```

- `com.apama.Channel` — The `send...to` statement sends the event to the specified `Channel` object. If the `Channel` object contains a string, the event is sent to the channel with that name. If the `Channel` object contains a context, the event is sent to that context. You cannot send an event to an empty context object.

You cannot send an event to a dictionary of contexts. However, it is a common pattern to send to a sequence generated by `dictionary.values()`. For example:

```
send x to d.values;
```

If the target context's input queue is full the sending context blocks and waits for space on the queue unless doing so would cause a deadlock. See "Deadlock avoidance when parallel processing" in *Developing Apama Applications in EPL*.

Sent events are processed in the order they are sent. Sent events are put on the back of the input queue, behind any events already queued.

You must create the context before you send an event to the context. You cannot send an event to a context that you have declared but not created. For example, the following code causes the correlator to terminate the monitor instance:

```
monitor m {
   context c;
   action onload()
   {
      send A() to c;
   }
}
```

If you send an event to a sequence of contexts and one of the contexts has not been created first then the correlator terminates the monitor instance. For details, see "Enqueuing to contexts" in *Developing Apama Applications in EPL*.

Sending an event to a sequence of contexts is non-deterministic. For details, see "Sending an event to a sequence of contexts" in *Developing Apama Applications in EPL*.

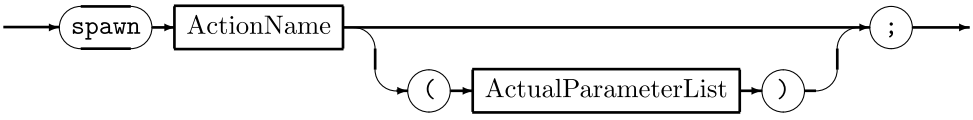
In a `send...to` statement, you cannot send an event that has a field of type `action`, `chunk`, `listener`, or `stream`.

Simple statements

The spawn statement

The `spawn` statement creates a copy of the currently executing monitor instance in the current context.

SpawnStatement

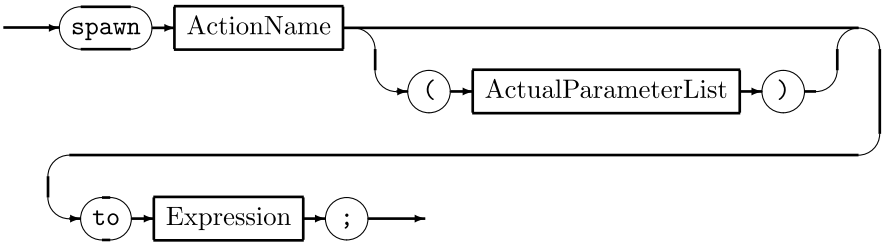


Simple statements

The spawn *action* to *context* statement

The `spawn` *`action`*`()` `to` *`context`* statement creates a copy of the currently executing monitor instance in the specified context. A monitor instance must have a reference for the specified context in order to spawn to that context.

SpawnToStatement



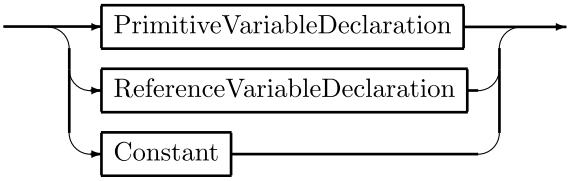
The result type of Expression must be `context`. The `spawn` *`action`*`()to context` statement spawns a new monitor instance in the specified context.

Simple statements

Variable declaration statements

A variable declaration statement can specify a primitive or reference variable.

VariableDeclaration

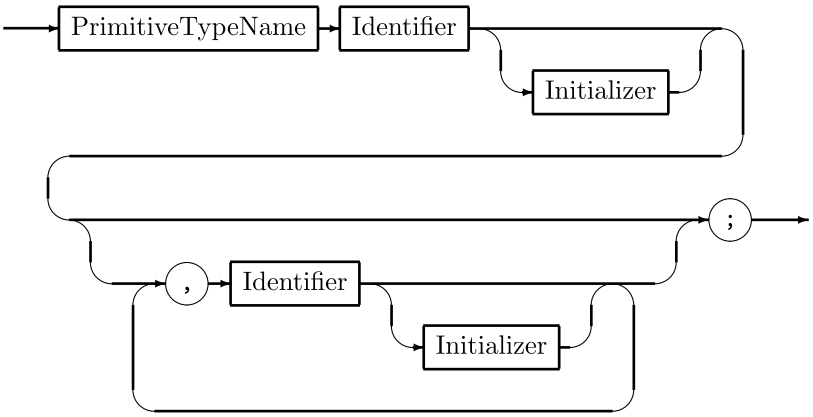


See "Variable declarations" on page 150.

Primitive type variable declarations

A PrimitiveVariableDeclaration specifies a `boolean`, `decimal`, `float`, `integer`, or `string` type variable.

PrimitiveVariableDeclaration

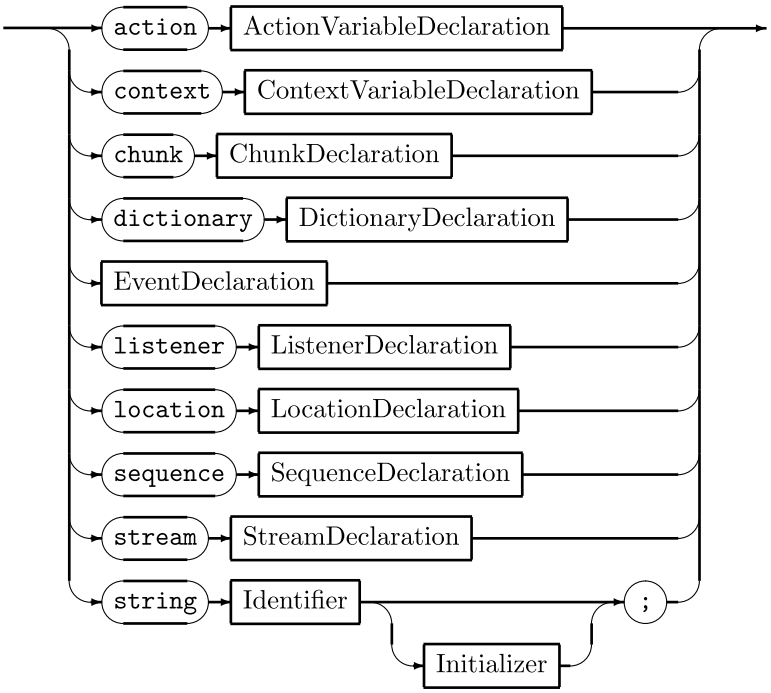


See "Primitive type variable declarations" on page 150.

Reference type variable declarations

A ReferenceVariableDeclaration specifies a reference type variable.

ReferenceVariableDeclaration



A variable declaration statement can appear anywhere in a block. Variables declared in a block are scoped to that block and can be used in statements that follow the declaration.

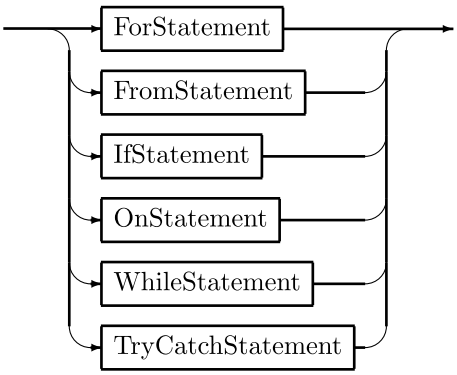
For details, see "Reference type variable declarations" on page 122.

Simple statements

Compound statements

Compound statements enclose other statements or blocks and affect how the enclosed statements are executed.

CompoundStatement

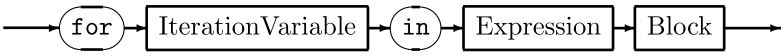


Statements

The for statement

The `for` statement is used to iterate over the members of a sequence and execute the enclosing statment or block once for each member.

ForStatement



Rule components

The expression, which must produce a reference to a `sequence` or a `dictionary`'s key, is evaluated. Then the IterationVariable is assigned a value successively obtained from each element of the sequence, starting with the first, and if the last sequence entry has not been reached, the statement that forms the loop body is executed.

The IterationVariable's type must match the type of the sequence elements.

The loop body is either a single Statement or a Block.

Within the loop body, the `break` statement can be used to cause early termination of the loop by transferring control to the next statement after the loop body. The `continue` statement can be used to transfer control to the end of the body, after which the sequence size is tested to determine if the last entry has been reached. If it has not, then the loop body is executed. The `return` statement can be used to terminate both the loop and the action that contains it.

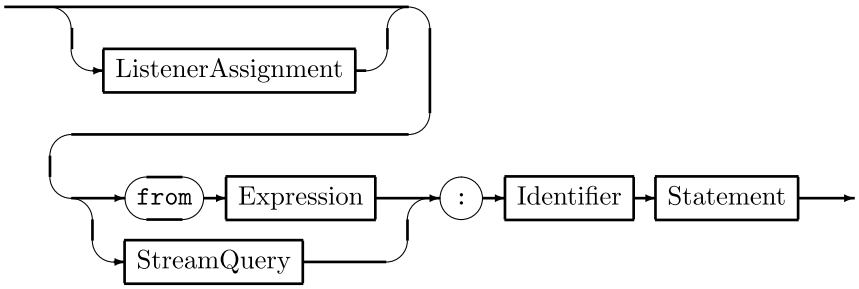
Compound statements

The from statement

The `from` statement is used to create a stream listener. A stream listener watches for items from a stream and passes output items to procedural code.

A `from` statement is similar to an `on` statement, which listens for events processed by the correlator and then executes an event listener action for each matching event or pattern. See "The on statement" on page 125.

FromStatement



Rule components

You can assign the result of a `from` statement to a `listener` variable. This lets you call `quit()` on the stream listener.

A stream listener passes output items from a stream to procedural code. The stream, specified in Expression, can be a reference to an existing stream or a stream source template. Alternatively, it can be the stream created by an in-line stream query.

A colon and an identifier follow the Expression or in-line stream query. This signifies a coassignment — when new items are available from the stream, the stream listener coassigns each output item to the specified variable.

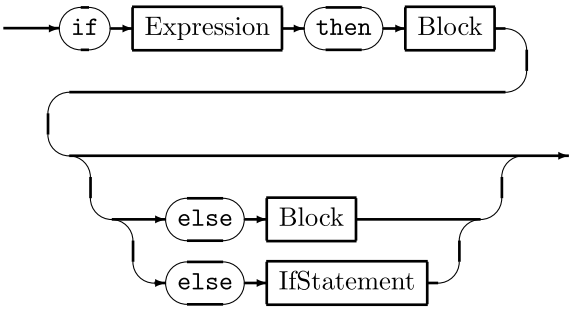
The statement following the identifier can be a single EPL statement or a block of EPL statements. The `from` statement passes the output item to this statement or block and executes the statement or block once for each output item. If the output of the query is a lot that contains more than one item, and you want to execute the statement or block just once for the lot, coassign the output to a `sequence`. See *Developing Apama Applications in EPL*, "Working with Streams and Stream Queries", "Working with lots that contain multiple items".

Compound statements

The if statement

The `if` statement is used to conditionally execute a statement or block.

IfStatement



Rule components

The expression, whose result type must be `boolean`, is evaluated and if its result is `true` the block following the `then` keyword is executed. After the body of the `then` clause has been executed, control is transferred to the next statement following the `if` statement.

If the expression result is `false`, and an `else` clause is present, the statement or block following the `else` is executed. After the body of the `else` clause has been executed, control is transferred to the next statement following the `if` statement.

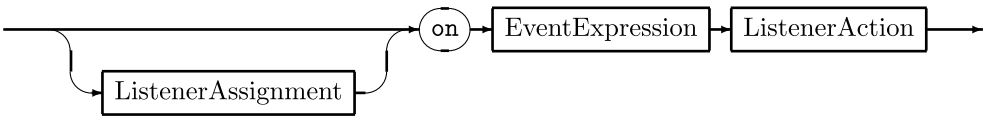
If the expression result is `false`, and the `else` clause is not present, control is transferred to the next statement following the `if` statement.

Compound statements

The on statement

The `on` statement is used to create an event listener that looks for input events that match the pattern specified by an event expression. When a matching event is detected, the event listener fires (also referred to as triggers) and the specified event listener action is executed.

OnStatement



Rule components

The ListenerAssignment clause is used to obtain a reference to the event listener that is created by the `on` statement. One can either define a new variable of type `listener` or specify a reference to an existing `listener` variable.

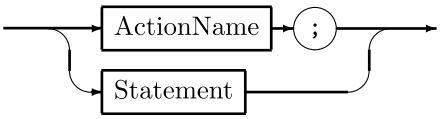
Example

```
listener l := on ...
sequence <listener> aSequence;
aSequence[0] := on ...
```

The EventExpression is an event expression that specifies what events are of interest. See "Event expressions" on page 89.

The ListenerAction defines the processing that will be performed when a matching event is detected and the event listener fires.

ListenerAction



The ListenerAction can be one of the following:

- Name of a monitor action
- A statement
- A block

The ListenerAction is invoked automatically by the correlator when the event expression is satisifed. This may be

- When a matching event is detected.
- If `unmatched` is specified in the expression, the event matches the expression, and there are no matching event listeners that do not specify the `unmatched` keyword.
- If `completed` is specified in the expression, and any matching events have been completely processed by other event listeners.

If an action name is specified instead of a block, then it must be an action that does not return a value and that does not have any parameters.

Compound statements

The while statement

The `while` statement is used to repeatedly evaluate a `boolean` expression and execute a block as many times as the expression result is found to be `true`.

WhileStatement

Rule components

The expression, whose result type must be `boolean`, is evaluated and if the result is `true`, the block is executed. Control then transfers to the top of the loop and the expression is evaluated again. When the expression result is `false`, control is transferred to the next statement following the `while` statement.

The body of the loop must be a block; it must be inside curly braces.

Within the loop body, the `break` statement can be used to cause early termination of the loop by transferring control to the next statement after the loop body. The `continue` statement can be used to transfer control to the end of the body, after which the expression will be evaluated again and the

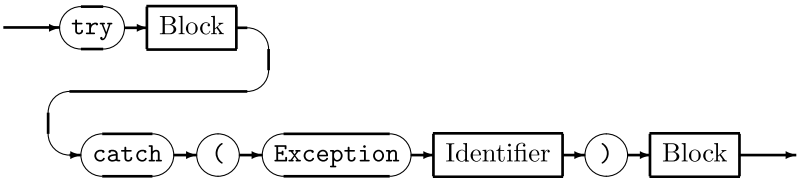
loop body executed if the expression result is `true`. The `return` statement can be used to terminate both the loop and the action that contains it.

Compound statements

The try-catch statement

The try-catch statement is used to handle runtime exceptions.

TryCatchStatement



Rule components

The `catch` clause must specify a variable whose type is `com.apama.exceptions.Exception`.

You can nest try-catch statements in an action and you can specify multiple actions in a `try` block and specify a try-catch statement in any number of actions.

See also: "Catching exceptions" in *Developing Apama Applications in EPL, Defining What Happens When Matching Events Are Found*.

Example

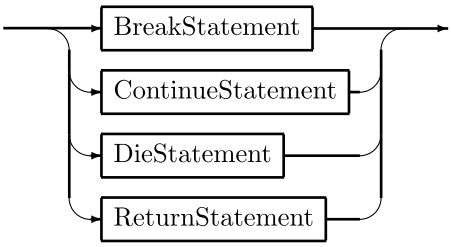
```
using com.apama.exceptions.Exception;
...
action getExchangeRate(
   dictionary<string, string> prices, string fxPair) returns float {
   try {
      return float.parse(prices[fxPair]);
   } catch(Exception e) {
      return 1.0;
   }
}
```

Compound statements

Transfer of control statements

Transfer of control statements alter the normal control path by stopping the sequential execution of statements within a block. All of them end execution of the block that contains them. After a `continue` statement is executed, the containing block might be executed again in a new loop iteration. The `die` and `return` statements also end the action in which they are executed.

TransferOfControlStatement

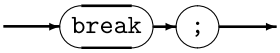


Statements

The break statement

The `break` statement transfers control to the next statement following the loop (`for` or `while` statement) that encloses the `break` statement. A `break` statement can only be used within a `for` or `while` statement. Any statements between the `break` statement and the end of the block are not executed.

BreakStatement

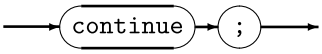


Transfer of control statements

The continue statement

The `continue` statement can be used in a block enclosed by a `for` or `while` statement to end execution of the current iteration and transfer control to the beginning of the loop. When a `continue` statement is executed, control is immediately transferred to the beginning of the inner most enclosing `for` or `while` statement. Any statements between the `continue` statement and the end of the block are not excuted.

ContinueStatement



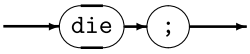
Transfer of control statements

The die statement

The `die` statement terminates the execution of a monitor. When the correlator executes a `die` statement, it terminates only the monitor instance that contains the `die` statement being executed. If the monitor instance that spawned the monitor instance being terminated is still active, that monitor instance is not affected. If that original monitor instance spawned any other monitor instances, those monitor instances are not affected. If the monitor instance being terminated defines an `ondie()` action,

the correlator executes the `ondie()` action for just the monitor instance being terminated, and then terminates the monitor instance.

DieStatement

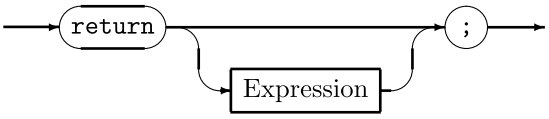


Transfer of control statements

The return statement

The `return` statement ends the execution of an action and control is transferred to the action's caller, at the point following the action call (which might be in the middle of an expression). Any statements between the `return` statement and the end of action are not executed.

ReturnStatement



If the action does not have a `returns` clause then an expression is not permitted in the `return` statement.

If the action has a `returns` clause then an expression whose value is the action's return value is required in the `return` statement. The expression type must match the type specified in the `returns` clause.

Transfer of control statements

Chapter 7: Expressions

In many programs, much work is performed by evaluating expressions, which are combinations of operators, operands, and punctuation. They are used to detect events of interest to the program, perform calculations, comparisons, invoke actions, invoke inbuilt methods, compute parameter values passed to action and method calls, and so on.

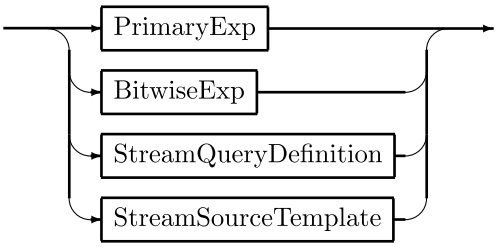
Introduction to expressions

EPL has several kinds of expressions:

- Primary and bitwise expressions are used for computations.
- A stream query definition creates a derived stream from an existing stream.
- A stream source template creates a new stream from an event template.

Event expressions are used in `on` statements for event pattern matching and sequence detection. Event expressions are not ordinary EPL expressions. See "Event expressions" on page 89.

Expression



Primary and Bitwise Expression Notes

When a primary or bitwise expression (called expression for the remainder of this section) is evaluated (that is, it is executed), it will produce a result value if the expression is a variable, a literal, or a combination of values and operators. If the expression is an action or inbuilt method call, then evaluating the expression produces a result value when the action or inbuilt method returns a value, but if the action or inbuilt method does not return a value, then the expression does not produce a result. Note that when an expression includes action or method calls, then evaluating the expression might produce side effects. A side effect is a change in the state of the execution environment. For example, a called action might change the value of a global variable or generate a derived event. If evaluating an expression produces a result, then in addition to a value, the expression result has a type. This is the expression type. An expression's type is always known at compile time.

The elements of an expression are evaluated roughly from left to right, taking into account parentheses and operator precedence. Binary operators have a left operand and a right operand. If an operator is left-associative, its left operand is evaluated first, followed by the right, and then the operation is performed. If an operator is right-associative, its right operand is evaluated first, followed by the left, then the operation is performed. In action calls, the actual parameter list expressions are evaluated from left to right. Many of the operators used in primary or bitwise expressions are polymorphic and can operate on operands of several types. For example, the addition operator performs floating point addition when its operands are of type `decimal` or `float` and performs integer addition when its operands are of type `integer`. Here are some examples of expressions:

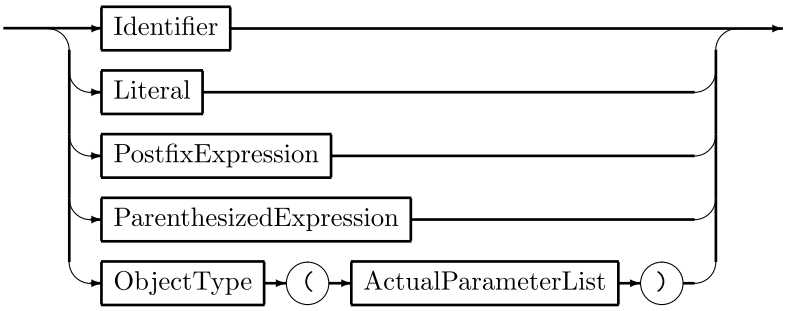
```
i := (a.size() + b[3]) / (n -1);
i := "foo" + s + " " + b.toString() + f.formatFixed(8);
```

Expressions

Primary expressions

The primary expression is the simplest form of expression. It can take the following forms:

PrimaryExp



Identifiers in expressions

In an expression, an identifier is a variable name, an instance method name, a type method name, or an action name.

Literals in expressions

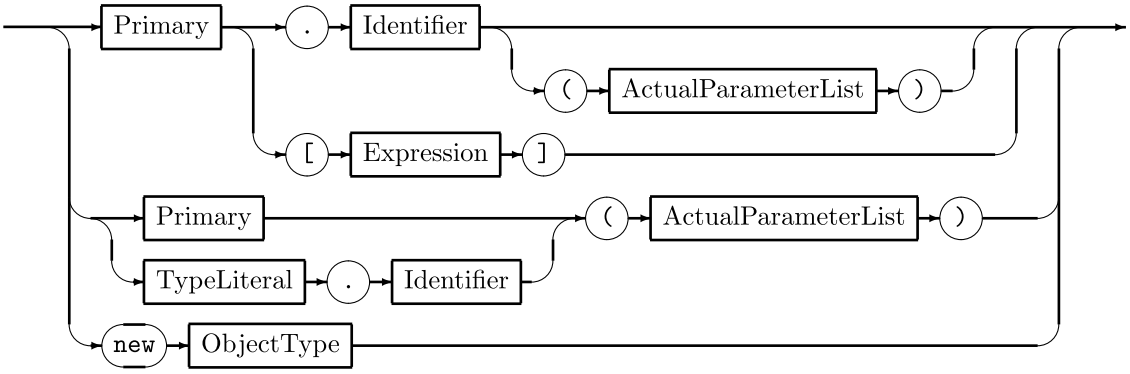
A literal in an expression is a compile-time constant value as described in "Literals" on page 178.

ParenthesizedExpression

Expressions

Postfix expressions

PostfixExpression



Rule components

A primary followed by a `"."` symbol, and an identifier must represent a variable reference, an action call, or a method call. Action and method calls are described in "Action and method calls" on page 133.

An expression enclosed by the `[` and `]` symbols denotes a subscript operation for a `sequence` or `dictionary`. This can be used on the right or left side of an assignment statement.

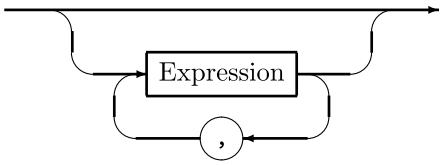
The `new` operator is used to create an instance of a reference type or event type.

Expressions

Action and method calls

An action call within an expression transfers control to the statements within the action body during expression evaluation and temporarily suspends the expression evaluation. If the action has parameters, then their values are copied to the action's formal parameter variables. When the control flow reaches the action's end or the action executes a `return` statement, control is transferred back to the expression and evaluation continues.

ActualParameterList



Rule components

The ActualParameterList is a comma-separated list of expressions. The entire list is enclosed in parentheses. It forms the set of parameter values that are passed when the action is called. Each expression value is copied to the corresponding parameter variable specified in the action definition's FormalParameters and the expression result type must match the parameter variable's type. The number and order of actual parameters passed by a caller must also match those listed in the action definition's formal parameters.

The action or method being invoked in the expression must return a value. The action's `return` type becomes the expression result type.

Postfix expressions

The subscript operator [ ]

The subscript operator takes one operand. The operand can be an `integer` index into a `sequence` or a key type index of a `dictionary`. The subscript operator produces a result of the same type as the sequence's entry type or dictionary's item type.

Postfix expressions

The new object creation operator

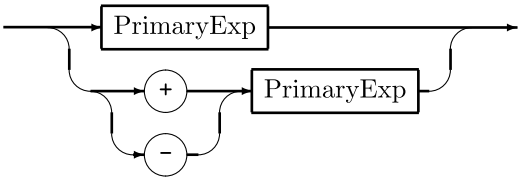
The operator `new` produces a result whose type is the type of the object parameter. It has one operand, the name of the type of object to be created.

Postfix expressions

Unary additive operators

The unary additive operators are used to perform arithmetic on one right operand of type `decimal`, `float` or `integer`. The result type of the unary arithmetic operators is the same as the type of the operand.

UnaryArithmeticExp



Rule components

Both of the unary arithmetic operators have one operand, which must be an expression of type `decimal` , `float` or `integer`. The result type is the same as the type of the operand.

Unary inverse

The unary additive inverse operator produces a result that is its right operand value with the sign reversed. If the operand value is negative, the result value is positive. If the operand value is positive, the result value is negative. If the operand value is zero, the result value is zero.

Unary identity

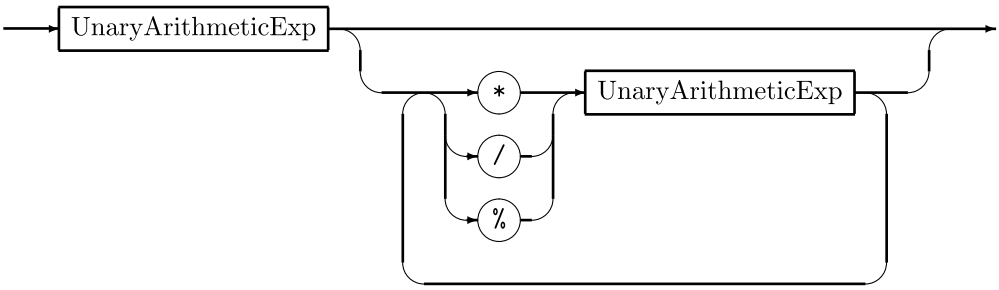
The unary additive identity operator `+` produces a result that is its right operand value.

Expressions

Multiplicative operators

The multiplicative operators are used to perform arithmetic on two operands of matching type, both `decimal` or both `float` or both `integer`.

MultiplicativeExp



Rule components

The left and right operands must both be expressions of type `decimal` or both be of type `float` or both be of type `integer`.

The result type of the multiplicative operators is the same as the type of the operands.

Expressions

Multiplication operator

The multiplication operator `*` produces a result by computing the numeric product of its two operands. If the two operands are both expressions of type `integer`, then integral multiplication is performed and the result is of type `integer`. If the two operands are both of type `decimal` or both of type `float`, then floating-point multiplication is performed and the result type is the same as the operand type.

Multiplicative operators

Division operator

The division operator `/` produces a result by computing the numeric quotient of its two operands. The left operand value, the dividend, is divided by the right operand value, the divisor.

Operand type

If both operands are of type `integer`, any fractional part of the result value is discarded. In other words, the result is truncated toward zero. For example, the expression `13/5` yields a result of `2`.

If both operands are of type `integer`, then integral division is performed and the result is of type `integer`. If both operands are of type `decimal` or both are of type `float`, then floating-point division is performed and the result type is the same as the operand type.

If the right operand's value is zero, a runtime error is raised.

Multiplicative operators

Remainder operator

The remainder operator % produces a result by computing the numeric remainder from dividing the left operand value by the right operand value. For example, the expression `13%5` yields a result of `3`.

Operand type

If both operands are of type `integer`, then the integral remainder is computed and the result is of type `integer`. If both operands are of type `decimal` or both of type `float`, then the floating-point remainder is computed and the result type is the same as the operand type.

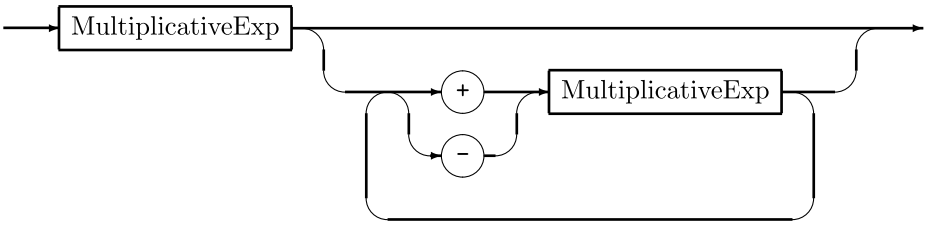
If the right operand's value is zero, a runtime error is raised.

Multiplicative operators

Additive operators

The additive operators perform addition, subtraction, and string concatenation.

AdditiveExp



The additive operators are used to perform arithmetic on two operands of matching type -- both of type `decimal`, both of type `integer` or both of type `float`. The result type of the additive operators is the same as the type of the operands.

Expressions

Addition operator

The addition operator + produces a result by computing the numeric sum of its left and right operands. If the two operands are both expressions of type `integer`, then integral addition is performed and the result is of type `integer`. If the two operands are both of type `decimal` or both of type `float`, then floating-point addition is performed and the result type is the same as the operand type.

Additive operators

Subtraction operator

The subtraction operator – produces a result by computing the numeric difference between the left and right operands by subtracting the value of the right operand from the left. If the two operands are both expressions of type `integer`, then integral subtraction is performed and the result is of type `integer`. If the two operands are both of type `decimal` or both of type `float`, then floating-point subtraction is performed and the result type is the same as the operand type.

Additive operators

String concatenation operator

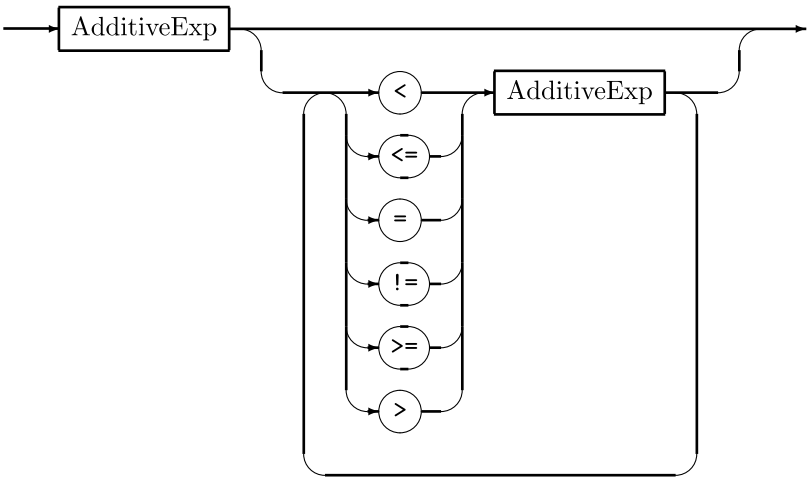
The string concatenation operator + produces a result by "adding" two strings together. The result is a new string whose value is the value of the right operand, an expression of type `string`, appended to the value of the left operand, an expression of type `string`. The result type of the `string` concatenation operator is `string`.

Additive operators

Relational operators

The relational operators are used to determine the equality, inequality, or relative values of their left and right operands.

CompareExp



The left and right operands must be expressions of the same type and the type must be allowed for that operator. You can use each relational operator on `decimal`, `float`, `integer`, and `string` types (see "Primitive and string types" on page 18). On `boolean` types, you can use the `=` and `!=` relational operators.

The result type of all six relational operators is `boolean`.

Expressions

Less-than operator

The less-than operator `<` produces the result `true` if the left operand's value is smaller than the right operand's value and `false` otherwise.

Relational operators

Less-than-or-equal operator

The less-than-or-equal operator `<=` produces the result `true` if the left operand's value is smaller than or equal to the right operand's value and `false` otherwise.

Relational operators

Equality operator

The equality operator `=` produces the result `true` if the left operand's value is equal to the right operand's value and `false` if they are not equal.

Relational operators

Inequality operator

The inequality operator `!=` produces the result `true` if the left operand's value is not equal to the right operand's value and `false` if they are equal.

Relational operators

Greater-than-or-equal operator

The greater-than-or-equal operator `=>` produces the result `true` if the left operand's value is larger than or equal to the right operand's value and `false` otherwise.

Relational operators

Greater-than operator

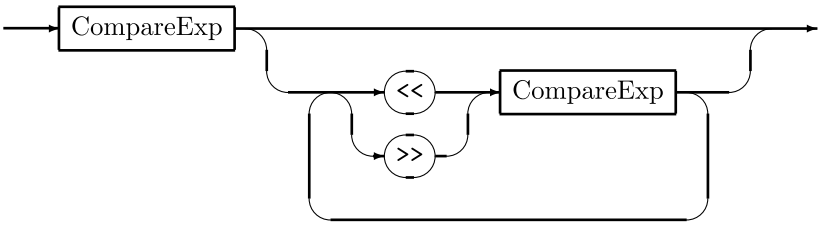
The greater-than operator `>` produces the result `true` if the left operand's value is larger than the right operand's value and `false` otherwise.

Relational operators

Shift operators

The shift operators `<<` and `>>` perform a shift of an integral value, moving bits in the result a specified number of positions to the right or left. The result type of both shift operators is `integer`.

ShiftExp



Rule components

The left operand is an expression of type `integer` whose value is to be shifted. The right operand is the shift count, an expression of type `integer` whose value is the number of bits the left operand value is to be shifted.

The shift count must be a nonnegative value less than 64. If the shift value is zero, then the result value is equal to the left operand value. Values less than zero or greater than 63 will produce unpredictable results and should not be used.

Expressions

Left shift operator

The left shift operator << produces a result by moving the left operand value's bits to the left and filling the vacated bits on the right with 0 bits. Bits that are moved beyond the leftmost bit (the sign bit) position are discarded.

Example

The following table illustrates this using 64-bit binary values.

`i := 42;`	`0000000000000000000000000000000000000000000000000000000000101010`
`i << 24`	`0000000000000000000000000000000000101010000000000000000000000000`

Shift operators

Right shift operator

The right shift operator >> produces its result by moving the left operand value's bit to the right. The vacated bits on the left are filled with 0 bits if the left operand value is zero or positive and filled with 1 bits if the left operand value is negative. Bits that are moved to beyond the rightmost bit (the least significant bit) position are discarded.

Examples

The following tables illustrate this using 64-bit binary values.

`i := 42;`	`0000000000000000000000000000000000000000000000000000000000101010`

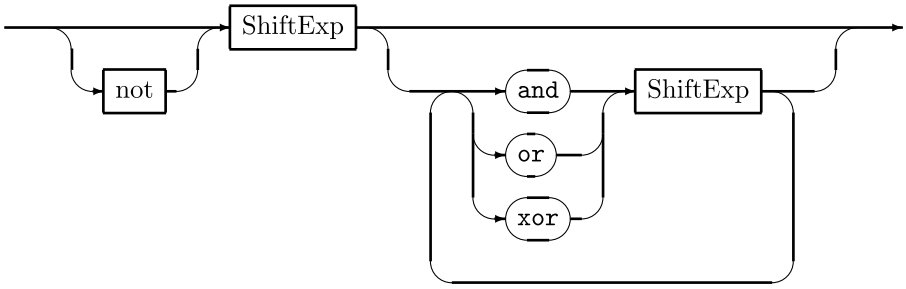
`i >> 24`	`00000000000000000000000000000000000000000000000000000000000000000000000000000000`
`i := -42;`	`11111111111111111111111111111111111111111111111111111111111111111111111111010110`
`i >> 24`	`11111111111111111111111111111111111111111111111111111111111111111111111111111111`

Shift operators

Logical operators

The logical operators `and`, `or`, `xor` and `not` perform Boolean arithmetic on their operands.

LogicalExp



Rule components

The logical operators' left and right operands are expressions whose result type must be `boolean`. The result type of all four operators is `boolean`.

Expressions

Logical intersection (and)

The `and` operator produces a result of `true` if both of its operand values are `true` and `false` otherwise.

When the correlator evaluates a logical `and` expression it evaluates the left operand first. If the left operand evaluates to false then the correlator does not evaluate the right operand since the expression cannot be true. For example:

```
a and b
```

If `a` is false then whether or not `b` is true the expression will be false so the correlator does not evaluate `b`. This lets you write code such as the following:

```
if (dict.hasKey(k) and dict[k] = "someValue")
```

If `k` is not in the dictionary then the left operand evaluates to false and so the entire logical expression is false. The correlator never evaluates `dict[k] = "someValue"`, which would cause an error if `k` is not in the dictionary.

Logical operators

Logical union (or)

The `or` operator produces a result of `true` if either of its operand values is `true` and `false` otherwise.

When the correlator evaluates a logical `or` expression it evaluates the left operand first. If the left operand evaluates to true then the correlator does not evaluate the right operand since the expression will always be true. For example:

```
a or b
```

If `a` is true then regardless of what `b` evaluates to the expression will be true so the correlator does not evaluate `b`.

Logical operators

Logical exclusive or (xor)

The `xor` operator produces a result of `true` if either of its operand values is `true` and the other is `false` and `false` if both are `true` or both are `false`.

Logical operators

Unary logical inverse (not)

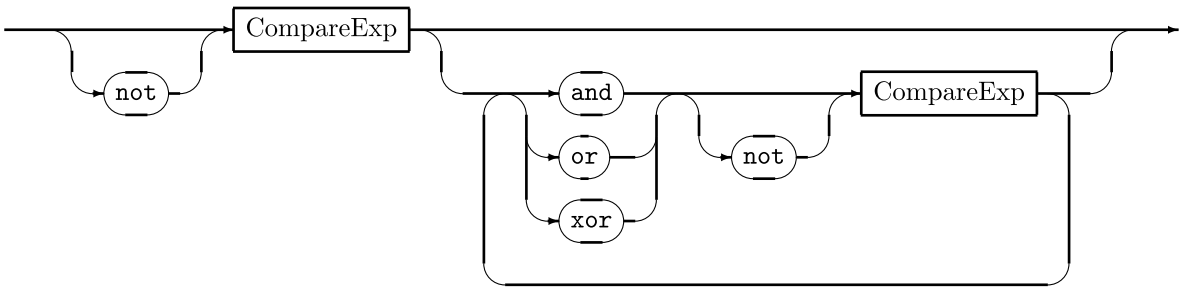
The unary `not` operator produces the result `true` if its right operand value is `false`, and `false` if the operand value is `true`.

Logical operators

Bitwise logical operators

The bitwise logical operators examine one bit at a time in their operands and compute the corresponding bit value in the result.

BitwiseExp



Rule components

The bitwise operators `and`, `or`, and `xor` are binary operators that have a left and right operand. The bitwise operator `not` is a unary operator that has only a right operand.

The result type of all four bitwise operators is `integer`. Note that EPL integers are 64 bits wide.

Expressions

Bitwise intersection (and)

The bitwise intersection operator `and` produces a result by comparing all 64 bits of its left and right operands, which must be expressions of type `integer`, one bit at a time. For each bit in the two operands, the corresponding bit in the result value is set to `1` if both operand bit values are `1` and set to `0` if either operand bit value is `0`.

Example

The following table illustrates this using 64-bit binary values.

`a := 42;`	`0000000000000000000000000000000000000000000000000000000000101010`
`b := 642;`	`0000000000000000000000000000000000000000000000000000001010000010`
`a and b`	`0000000000000000000000000000000000000000000000000000000000000010`

Bitwise logical operators

Bitwise union (or)

The bitwise union `or` produces a result by comparing all 64 bits of its left and right operands, which must be expressions of type `integer`, one bit at a time. For each bit in the two operands, the corresponding bit in the result value is set to `1` if either or both operands bit values is `1` and set to `0` if both operand bit values are `0`.

Example

The following table illustrates this using 64-bit binary values.

`a := 42;`	`0000000000000000000000000000000000000000000000000000000000101010`
`b := 642;`	`0000000000000000000000000000000000000000000000000000001010000010`
`a or b`	`0000000000000000000000000000000000000000000000000000001010101010`

Bitwise logical operators

Bitwise exclusive (xor)

The bitwise exclusive `or` operator `xor` produces a result by comparing all 64 bits of its left and right operands, which must be expressions of type `integer`, one bit at a time. For each bit in the two operands, the corresponding bit in the result value is set to `1` if either operand's bit value is `1` and the other is `0` and set to `0` if both operand bit values are `0` or both are `1`. In other words, the result bit is `1` if both bit values are different and `0` if they are the same.

Example

The following table illustrates this using 64-bit binary values.

`a := 42;`	`0000000000000000000000000000000000000000000000000000000000101010`
`b := 642;`	`0000000000000000000000000000000000000000000000000000001010000010`
`a xor b`	`0000000000000000000000000000000000000000000000000000001010101000`

Note that the expression `a xor b` yields the same result as `not (a and b)`.

Bitwise logical operators

Unary bitwise inverse

The unary bitwise `not` operator produces a result by computing the bitwise complement or inverse of its right operand, which must be an expression of type `integer`. For each bit in the operand's value, the corresponding bit in the result value is set to `1` if the operand's bit value is `0` and `0` if the operand's bit value is `1`.

Example

The following table illustrates this using 64-bit binary values.

`b := 42;`	`0000000000000000000000000000000000000000000000000000000000101010`
`not b`	`1111111111111111111111111111111111111111111111111111111111010101`

Bitwise logical operators

Expression operator precedence

The following table lists the primary and bitwise expression operators in order by their precedence, from lowest to highest. See also "Event expression operator precedence" on page 96.

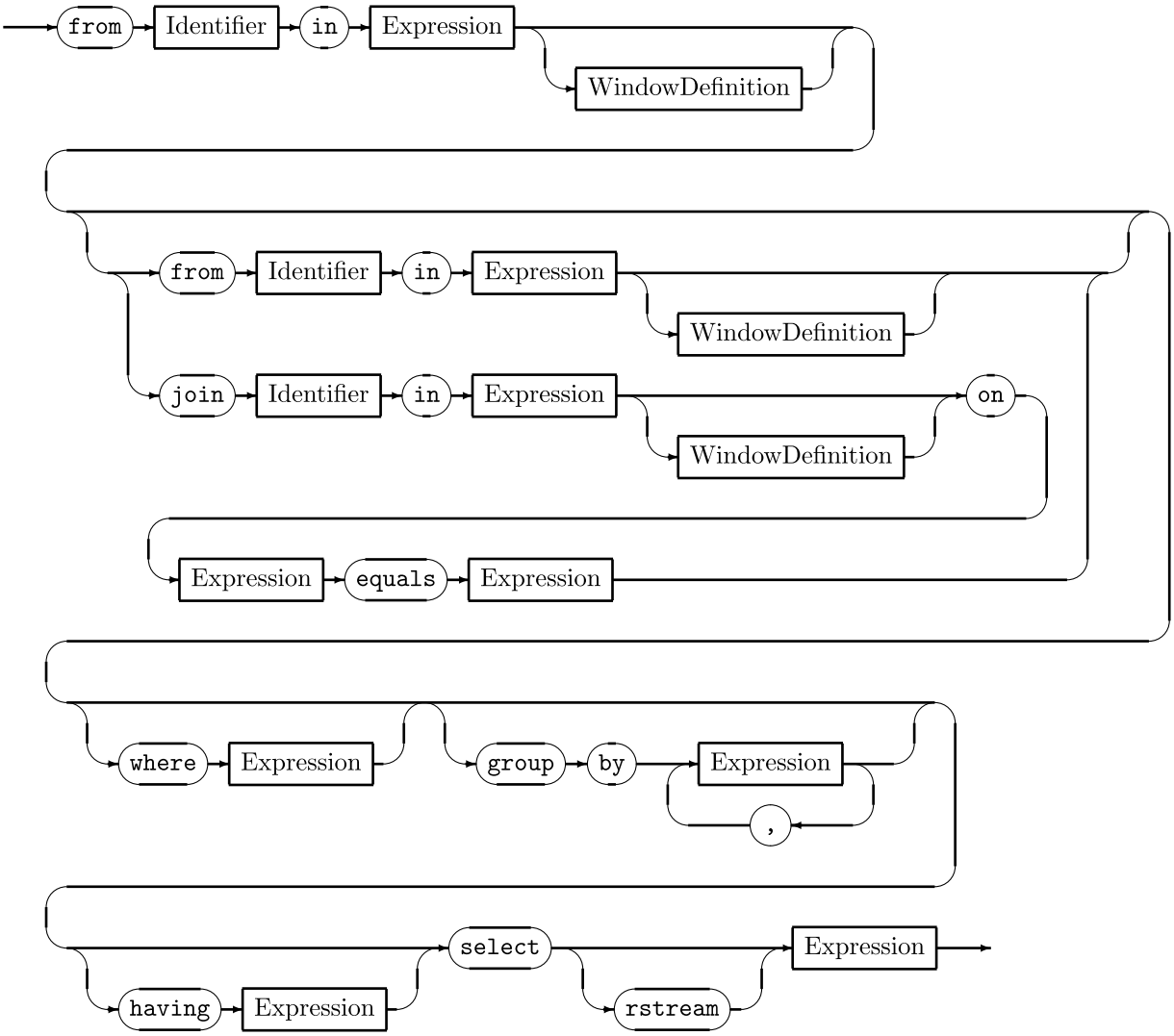
Operation	Operator	Precedence
Logical or bitwise union	`or`	1
Logical or bitwise exclusive or	`xor`	2
Logical or bitwise intersection	`and`	3
Unary logical or bitwise inverse	`not`	4
Relational	`<, <=, >, >=, !=, =`	5
Additive	`+, -`	6
String concatenation	`+`	6
Multiplicative	`*, /, %`	7
Unary additive	`+, -`	8
Name qualifier (Dot)	`.`	9
Object constructor	`new`	9
Subscript	`[ ]`	9
Action call	`ActionName()`	10
Parenthesised expression	`( )`	10
Stream query	`from`	10
Stream source template	`all`	10

Expressions

Stream queries

A stream query defines an operation that the correlator applies continuously to one or two streams of items. The output of a stream query is a continuous stream of derived items, `stream<X>`, where *X* is the type returned by the expression in the `select` clause.

StreamQueryDefinition



Rule components

A `from` clause specifies a stream that the query is operating on.

An item in a stream can be an event, a simple type (`boolean`, `decimal`, `float`, `integer` or `string`) or a `location` type. The first *`Identifier`* is the identifier that represents the current item in the stream you are querying. You use this identifier in subsequent clauses in the stream query.

The first *`Expression`* identifies the stream that you want to query.

A stream query window definition is optional. If you do not specify any window then the stream query operates on only the items that arrive on the stream for a given activation of that query. See "Stream query window definitions" on page 146.

A subsequent `from` clause indicates a cross-join operation.

Alternatively, a subsequent `join` clause indicates an equi-join operation. An equi-join has a key expression for each of the two streams that are being joined. Two items are joined into an output item only if the values of their key expressions are equal.

A `where` clause qualifies the items produced from a window or a join operation.

A `group by` clause organizes the qualified items, or the items produced from a window or join operation.

A `having` clause filters the output items produced from the projection.

The required `select` clause specifies how to generate the output items.

Semantic constraints

`from` *`Identifier`* `in` *`Expression`* `join` *`Identifier`* `in` *`Expression`*

The identifier can be any legal identifier and, within the stream query's scope, is associated with items from the source stream and therefore has their type. In a joined stream query, the two identifiers must be distinct.

The expression's result must be a value of some stream type. The correlator evaluates the expression outside the stream query's scope. For example:

```
stream<A> a := all A();
from a in a …
```

This is legal, because the identifier `a` is not in scope for evaluation of the expression `a`.

`on` *`Expression1`* `equals` *`Expression2`*

The correlator evaluates both expressions within the stream query's scope.

`Expression1` must contain the first item identifier and cannot contain the second. *`Expression2`* must contain the second item identifier and cannot contain the first.

The two expressions must return the same type, and that type must be a comparable type.

`where` *`Expression`* `group by` *`Expression`*`, ` *`Expression`*`, ...`

The item identifier or identifiers are in scope and should be used in these expressions. The `where` expression must return a `boolean` value. The `group by` expressions can return any comparable types.

`having` *`Expression`*

The item identifier or identifiers are in scope and can be used in this expression. The presence of this clause implies that the projection must be an aggregate projection. The expression must return a `boolean` value.

You can use one or more aggregate functions in the `having` expression. In fact, you can use aggregate functions only in `having` expressions and `select` expressions.

`select [rstream]` *`Expression`*

The item identifier or identifiers are in scope and can be used in this expression. The expression must return a value.

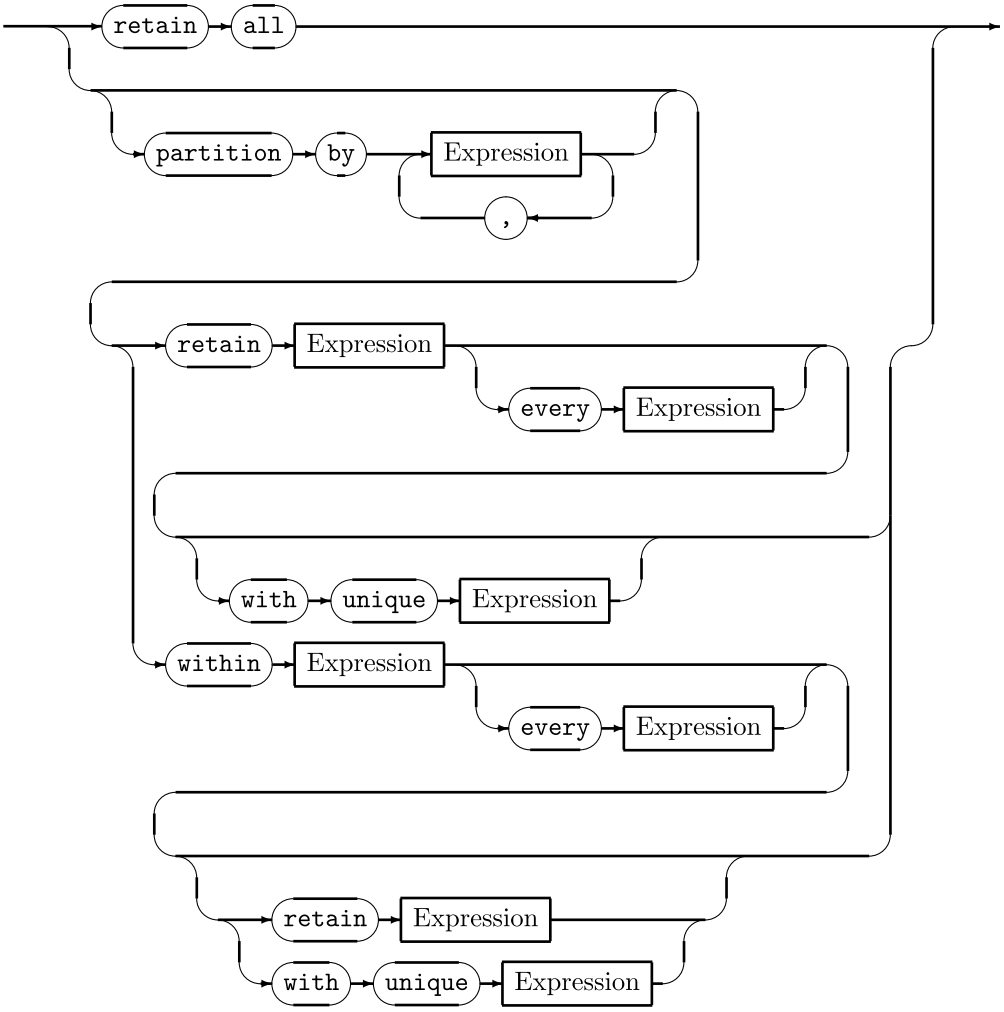
You can use one or more aggregate functions in a `select` expression. In fact, you can use aggregate functions only in `having` expressions and `select` expressions. If you specify an aggregate function you cannot specify the `rstream` keyword.

Expressions

Stream query window definitions

In a stream query, the optional window definition specifies which items in a stream to operate on.

StreamQueryWindowDefinition



Typically, stream queries process a window over a stream. A stream is an ordered sequence of items over time. A window specifies which items to operate on. Windows can contain a portion of the stream based on number of items, time of item arrival, content of item, or other criteria.

Rule Components

When the stream query window definition is `retain all`, the window contains all items that have ever been in the stream. Conceptually, once an item enters a `retain all` window, it remains in the window indefinitely, or until the stream query is terminated. The `retain all` clause specifies an unbounded window. Unbounded windows have restrictions on their use:

- You cannot have a partitioned or batched unbounded window.
- You cannot perform a join operation on an unbounded window.
- You cannot speciy an unbounded window when you use `rstream` in the `select` clause of a stream query.

When you use a custom aggregate function in a stream query that contains an unbounded window, you cannot use a bounded aggregate function. You should also be aware that, if you use a badly

implemented custom aggregate function in a stream query that contains an unbounded window, then this can result in uncontrolled memory usage.

A `partition by` clause divides the input data into several partitions and then applies the stream query window definition separately to each partition. The `partition by` expressions must be comparable types.

The `retain` clause specifies the maximum number of items to be retained by the window. The `retain` expression must be an `integer` expression. In a size-based window, as each new item arrives in the stream, it is added to the window. After the number of items in the window reaches the window size limit specified in the `retain` clause, the arrival of a new item causes removal of the oldest item from the window.

The `within` clause specifies the number of seconds to keep each new item in the window. The `within` expression must be a `float` expression. In a time-based window, as each new item arrives in the stream, it is added to the window. As soon as an item has been in the window for the number of seconds specified by the `within` expression, the correlator removes the item from the window.

By default, the contents of a window change upon the arrival of each item. The `every` keyword can be used to control when the contents of the window change, which causes the items to be added to the window in batches of several items at once. Time-based windows can be controlled to update only every *p* seconds and size-based windows can be controlled to update only every *m* events.

The contents of the window can also depend on the content of individual items in the stream. Specify `with unique`*`Expression`* to limit the window to containing only the most recent item for each key value identified by the expression.

Semantic constraints

In a stream query window definition for one of a joined stream query's input streams, it is always an error to refer to the other input stream's item identifier.

`partition by` ***`Expression`*****`,`** ***`Expression`*****`, ...`**
You should use the item identifier in each expression. Expressions can return any comparable types.

`retain` ***`Expression`*** **`[every`** ***`Expression`*****`]`**
You cannot use the item identifier in these expressions. These expressions must return `integer` values.

`within` ***`Expression`*** **`[every`** ***`Expression`*****`]`**
You cannot use the item identifier in these expressions. These expressions must return `float` values.

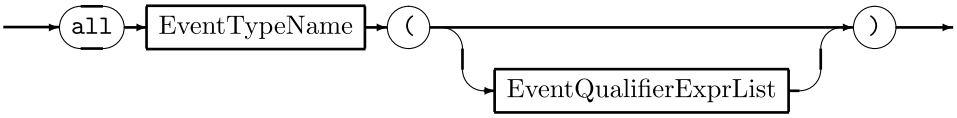
`with unique` ***`Expression`***
You should use the item identifier in this expression. The expression can return any comparable type.

Stream queries

Stream source templates

A stream can be created from an event template using the `all` keyword. This is referred to as a stream source template.

StreamSourceTemplate



A stream source template is the `all` keyword followed by a single event template. The output of a stream source template is a continuous stream of items, `stream<X>`, where *X* is the type specified by the event template.

Expressions

Chapter 8: Variables

Variables are names that are bound to data values (in the case of primitive types) or the location of data values (in the case of reference types). Variables are declared by specifying a type, a name, and optionally, an initial value. With the exception of the `string` type, once declared, new values can be computed and assigned to variables as needed. Strings are immutable and variable assignment causes a new string value to be created and bound to the `string` variable.

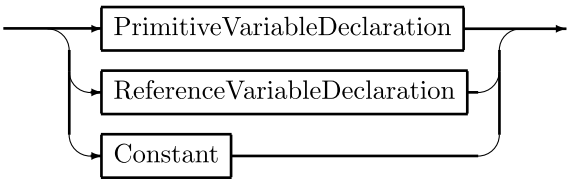
Variable declarations

Before a variable or a named constant value can be referenced in a program, it must be declared.

The declaration gives the variable or named constant a unique name, a type and, optionally except for constants, an initial value. There are three forms of variable declarations, the PrimitiveVariableDeclaration, the ReferenceVariableDeclaration and the Constant. You must supply a literal value for a constant. Aside from the types, the main difference is that the declarations for the primitive types are simpler than for reference types.

Variable declarations in actions and blocks are statements that are executed when the program's control flow reaches them.

VariableDeclaration

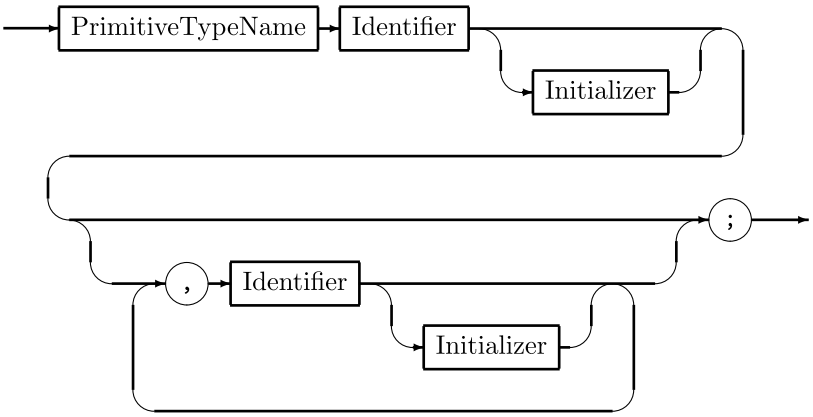


Variables

Primitive type variable declarations

The PrimitiveTypeName is one of the primitive types: `boolean`, `decimal`, `float`, `integer`, or `string`.

PrimitiveVariableDeclaration



Rule components

The Identifier following the type name becomes the variable's name. The variable name must be unique with respect to other variables declared within the same scope in which the variable declaration occurs. Variable names cannot be the same as any of the keywords listed in "Lexical Elements" on page 163.

Example

```
boolean a;
boolean b;
boolean c, d;
integer i;
float f; s
string s1;
string s2;
```

The diagram is also true for reference variable declarations, however additional detail about reference variable declarations is in "Reference-type variable declarations" on page 152.

Variable declarations

Primitive-type initializers

Variable of the primitive types can be given an initial or starting value by including an Initializer after the variable name.

Initializer

Rule components

The expression result type in an Initializer must be of the same type as the variable being initialized. The expression's result value is used as the variable's initial or starting value.

Examples

```
boolean a := true;
```

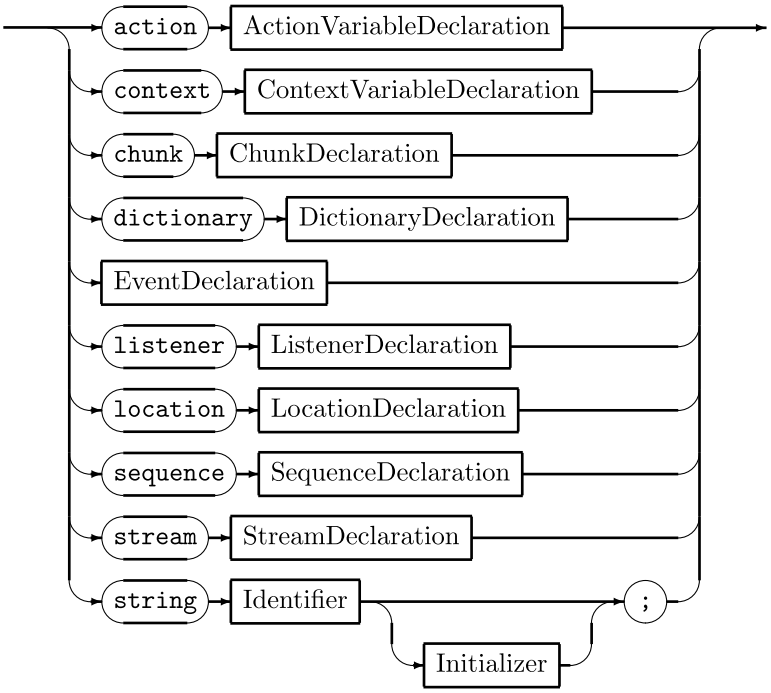
```
boolean b := false;
boolean c := true, d := false;
integer i := 42;
float f := 100.6180339887;
string s1 := "abcdefghijklmnopqrstuvwxyz";
string s2 := s1;
```

Variable declarations

Reference-type variable declarations

ReferenceVariableDeclarations differ in structure depending on the particular reference type. See "Reference types" on page 37. Each kind of declaration contains an Identifier that becomes the variable's name. The variable name must be unique with respect to other variables declared within the same scope in which the variable declaration occurs. Variable names cannot be the same as any of the keywords listed in "Lexical Elements" on page 163.

ReferenceVariableDeclaration

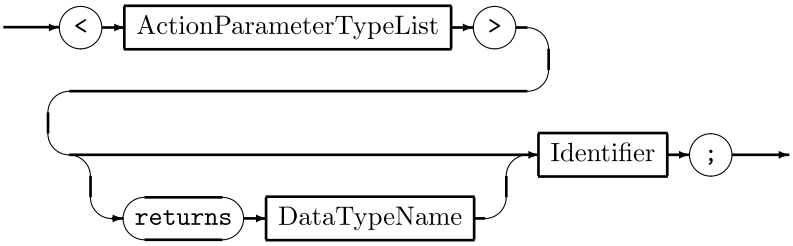


Variable declarations

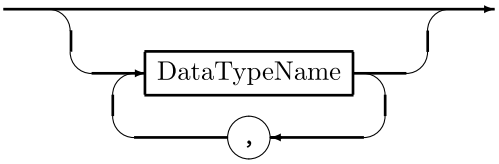
Action variable declarations

You can assign an action to an `action` variable of the same `action` type. An action is of the same type as an `action` variable if they have the same argument list (the same types in the same order) and return type (if any).

ActionVariableDeclaration



ActionParameterTypeList



Rule components

The format for defining an `action` variable is as follows:

```
action<[type1 [, type2]...]> [returns type3] name;
```

Follow the `action` keyword with zero, one or more parameter types enclosed in angle brackets and separated by commas. The angle brackets are required even when the action takes no arguments.

Optionally, follow the parameter list with a `returns` clause. Specify the `returns` keyword followed by the type of the returned value.

Finally, specify the name of the variable.

Example

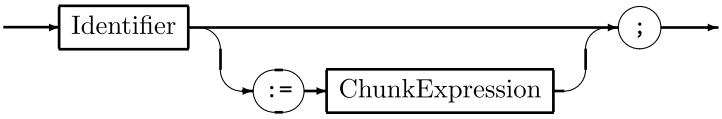
```
action<string> a;
action<integer, integer> returns string b;
```

Reference-type variable declarations

Chunk variable declarations

Variables of the type `chunk` are not used directly in EPL but can be passed as parameters in calls to plug-ins. See "Plug-ins" on page 105. Initializers are allowed for chunk variables.

ChunkDeclaration



Examples

```
chunk c;
chunk c1, c2;
```

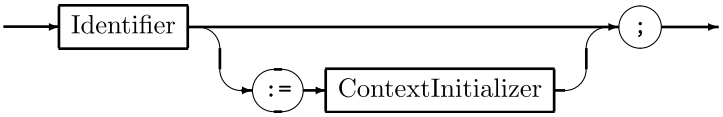
Reference-type variable declarations

Context variable declarations

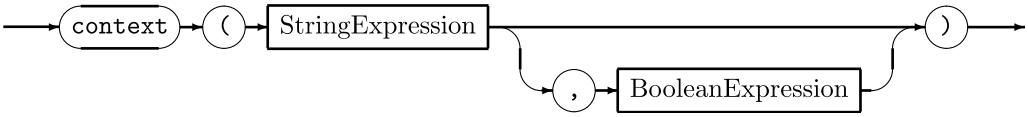
Use one of the following constructors to create a context:

```
context(string name)
context(string name, boolean receivesInput)
```

ContextVariableDeclaration



ContextInitializer



Rule components

The optional `receivesInput` Boolean flag controls whether the context is public or private:

- A public context can receive external events on the default channel, which is the empty string (`""`) as well as events that are sent on channels that the context is subscribed to.
- A private context can receive only those events that are sent on channels that the context is subscribed to. The default is that a context is private.

Example

The following example creates a reference, `c`, to a private context whose name is `test`:

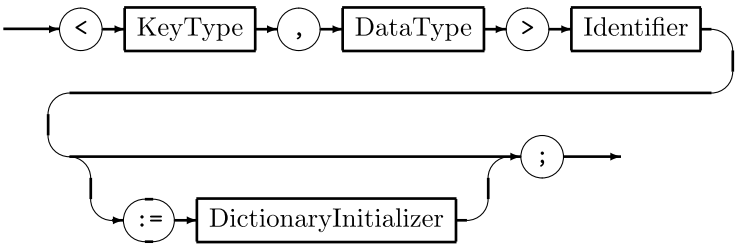
```
context c:=context("test");
```

Reference-type variable declarations

Dictionary variable declarations

A DictionaryDeclaration specifies a key type, a data type, an identifier, and an optional initializer.

DictionaryDeclaration



Rule component

The dictionary's KeyType and DataType, separated by a comma, are enclosed in angle brackets. The KeyType defines the type of the dictionary's key values. The DataType defines the type of the dictionary's data values.

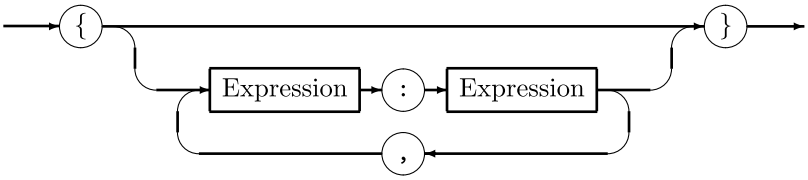
The Identifier becomes the dictionary variable's name.

The initial dictionary size is determined by the number of entries in the ditionary's initializer if one is present and zero otherwise.

Examples

```
dictionary<string, integer> dayNumbers;
dictionary<integer, string> dayNames;
```

DictionaryInitializer



Rule component

A DictionaryInitializer is a comma-separated list of expression pairs separated by colons. The entire list is enclosed in curly braces. Each expression value pair becomes an element in the dictionary. In each pair, the first expression value is the element's key value and the second is the element's data value.

The first expression result type in each pair must match the dictionary's key type. The second must match the dictionary's data type.

Examples

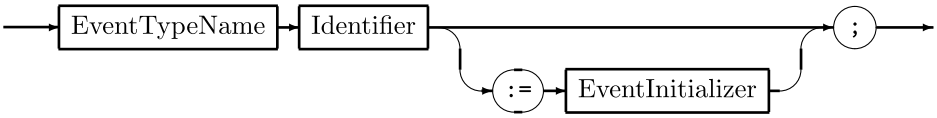
```
event PhoneticAlphabet
{
   action getPhoneticHexValues() returns dictionary <string, string>
   {
      dictionary <string, string> dict :=
         {"A":"Alfa", "B":"Bravo", "C":"Charlie", "D":"Delta",
          "E":"Echo", "F":"Foxtrot"};
      return dict;
    }
}

dictionary <string, integer> dayNumbers :=
   {"Sunday": 1, "Monday":2, "Tuesday": 3, "Wednesday": 4,
    "Thursday": 5, "Friday": 6, "Saturday": 7};

dictionary <integer, string> dayNames :=
   {1: "Sunday", 2: "Monday", 3: "Tuesday",
    4: "Wednesday", 5: "Thursday", 6: "Friday",
    7: "Saturday"};
```

Reference-type variable declarations

Event variable declarations

Event variable declarations, unlike variables of other reference types, do not begin with the keyword `event`, but instead begin with the EventTypeName that is part of the event definition that defines the event and its fields and actions. See "Event definitions" on page 79.

EventDeclaration



Examples

```
event etype1
{
   integer i;
}

event etype2
{
   boolean     b;
   integer     i;
   float       f;
   string      str;
   location    l;
   etype1      n;
}

etype2 e := etype2(true, -10, 1.73, "abc",
   location(1.0, 1.0, 5.0, 5.0), etype1 (1));
```

Reference-type variable declarations

Listener variable declarations

Listener variable values are references to listeners and cannot be initialized in the way other variables can. Instead, you must use the `on` statement or `from` statement to assign a value to a `listener` variable.

ListenerDeclaration

Rule component

The Identifier becomes the `listener` variable's name.

Example

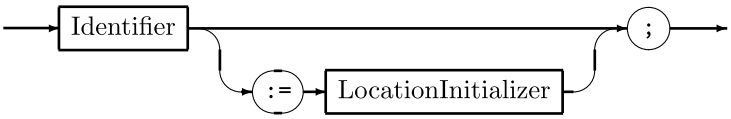
```
listener l;
```

Reference-type variable declarations

Location variable declarations

The Identifier becomes the location variable's name.

LocationDeclaration



Examples

```
location rect;
location point;
```

LocationInitializer

The four Expressions in the location initializer are separated by commas and must have a result type of `float`, are the coordinates of the two points `x1`, `y1`, and `x2`, `y2`, forming the location's enclosing boundary rectangle.

Examples

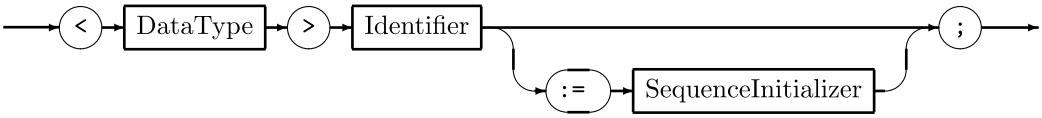
```
location rect := location(1.0, 1.0, 5.0, 5.0);
location point := location(1.0, 1.0, 1.0, 1.0);
```

Reference-type variable declarations

Sequence variable declarations

A value of type `sequence` contains a set of data elements or entries that are all of the same type.

SequenceDeclaration



Rule components

The DataType enclosed in angle-brackets defines the type of the element values contained in the sequence. You can use any primitive or reference type.

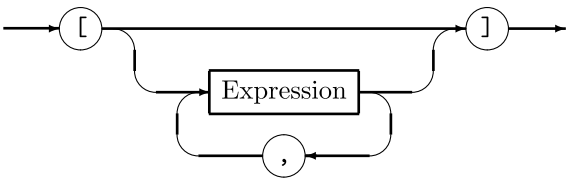
The Identifier becomes the sequence variable's name.

When you have a sequence of sequences or a sequence of dictionaries, the possibility that the declaration will contain two adjacent > characters arises. To distinguish them from the right shift operator >>, you must include a space between them.

Examples

```
sequence <integer> s;
sequence <string> w;
```

SequenceInitializer



Rule components

A SequenceInitializer is a comma-separated list of expressions enclosed in square brackets. Each expression value becomes an element in the sequence and the initial sequence size is determined by the number of expressions in the initializer.

The expression result types must all match the type specified in the sequence declaration.

Examples

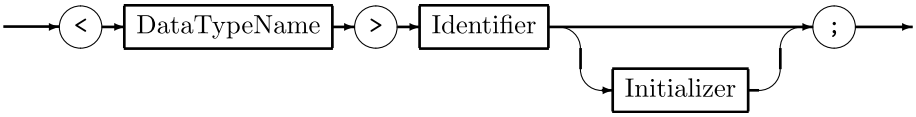
```
sequence <integer> s := [1, 3, 5, 7, 11, 13, 17];
sequence <string> w := ["one", "three", "five", "seven"];
```

Reference-type variable declarations

Stream variable declarations

A `stream` variable references a linearly ordered flow of items that have been processed by the correlator. The items in a stream are always ordered according to the order in which the correlator processed them. When more than one item arrives in a lot the correlator preserves the order in which it processed them.

StreamDeclaration



Rule components

An item in a stream can be any Apama type. A particular stream can contain only one type of item.

The optional initializer can be a stream source template, a stream query definition, or any other expression that returns a stream. A stream source template is an event template that specifies the `all` keyword and no other qualifiers. See "Stream source templates" on page 148. A stream query definition specifies a query that the correlator applies continuously to one or two streams. See "Stream queries" on page 144.

Examples

```
stream<Tick> ticks := all Tick(symbol=”APMA”);
stream<integer> derived := from a in sA select a.i;
```

Reference-type variable declarations

Variable scope

The parts of a program in which a particular variable can be referenced (that is, its value used or a new value assigned) is called the scope of the variable. In EPL, variables can have scopes that include

- All monitors — these are global variables that are part of EPL, also called predefined variables.
- The monitor within which they are declared.
- The action within which they are declared.

- The block within which they are declared.
- The event within which they are declared.
- The custom aggregate function in which they are declared.
- The stream query within which they are identified.

Regardless of the scope of a variable, it cannot be referenced in statements or expressions until after it has been declared or specified as an item identifier in a stream query. Further, variables scoped to actions or blocks cannot be referenced until a value has been assigned.

Within a scope at a particular level, variables declared at that level must have unique names. They can, however, have names that are the same as variables defined at an outer scope and in that case the variables declared at the inner level hide or mask the ones defined at the outer level(s) until the end of their scope.

Variables

Predefined variable scope

Predefined variables are defined by the correlator and are accessible in all monitors. See "Provided variables" on page 160.

Variable scope

Monitor scope

A variable that is defined in a monitor is visible and can be referenced in all parts of the monitor. Such variables are also called global variables.

Variable scope

Action scope

A variable that is declared in an action (also called a local variable) can only be referenced within the action. A variable that is a formal parameter of an action can only be referenced within the action. If a local variable declared in an action has the same name as a global variable declared at the monitor level, the local variable hides the global variable until the end of the action.

Variable scope

Block scope

A variable that is declared within a block can only be referenced within the block. A block is one or more statements enclosed within curly braces (the characters { and }). If a local variable declared in a block has the same name as a global variable declared at the monitor level, or a local variable

declared at the action level, the block's local variable hides the global variable or the action's variable, or both if all three have the same name, until the end of the block (the closing }).

Variable scope

Event action scope

The fields of an event are part of the event declaration. An event field's scope depends on where it is declared. When an event also includes action definitions, the statements in the action can reference the event's fields as simple identifiers. From the point of view of an event's action, the fields can be said to be scoped to the event.

Variable scope

Custom aggregate function scope

A variable that is declared in a custom aggregate function (also called a local variable) can only be referenced within the custom aggregate function. If a local variable declared in a custom aggregate function has the same name as a global variable declared at the monitor level, the local variable hides the global variable until the end of the custom aggregate function.

Variable scope

Provided variables

The EPL execution environment provides several variables. You can use these variables in the same way as variables you declare yourself, except that you cannot assign values to them. Instead, the correlator automatically assigns values to these variables.

Variables

currentTime

Purpose

The `currentTime` variable is a read-only `float` global variable that contains a timestamp value with the current time and date as read from the correlator's clock. Timestamps are encoded as the number of seconds and fractional seconds elapsed since midnight, January 1, 1970 UTC and do not have a time zone associated with them.

The current time is the time indicated by the most recent clock tick. Use the `currentTime` variable to obtain the current time. The value of the `currentTime` variable is always changing to reflect the correlator's current time.

If you have multiple contexts, it is possible for the current time to be different in different contexts. A particular context might be doing so much processing that it cannot keep up with the time ticks on its queue. In other words, if contexts are mostly idle, then they would all have the same current time.

In a context, the current time is never the same as the current system time. In most circumstances it is a few milliseconds behind the system time. This difference increases when the context's input queue grows.

When a listener executes an action, it executes the entire action before the correlator starts to process another event. Consequently, while the listener is executing an action, time and the value of the `currentTime` variable do not change. Consider the following code snippet,

```
float a;
action checkTime() {
   a := currentTime;
}
// ... Lots of additional code
// A listener calls the following action some time later
action logTime() {
   log a.toString(); // The time when checkTime was called
   log currentTime.toString(); // The time now
}
```

In this code, an event listener sets `float` variable `a` to the value of `currentTime`, which is the time indicated by the most recent clock tick. Some time later, a different event listener logs the value of `a` and the value of `currentTime`. The values logged might not be the same. This is because the first use of `currentTime` might return a value that is different from the second use of `currentTime`. If the two event listeners have processed the same event, the logged values are the same. If the two event listeners have processed different events, the logged values are different.

The correlator maintains a clock that advances at a fixed interval (default) of 0.1 seconds. The clock does not advance while an event is being processed.

Provided variables

Event timestamps

Purpose

The correlator defines an arrival timestamp for every event it receives. The arrival time value is set from the main context's clock when an event is received by the correlator, just before it is placed on the input queue of each public context.

You can access the arrival timestamp by calling the event's inbuilt `getTime()` method (see "event" on page 49). After the correlator creates an event or after you coassign an event, the `getTime()` method returns the time in the context when the event was created or coassigned. An event's arrival timestamp has the same scope as the event itself.

Provided variables

self

Purpose

The predefined variable `self` is an event reference that can be used to refer to an event instance within the event's definition.

Within an event action body, you can use the `self` variable to refer an event instance of that event type. In other words, the scope of `self` is each action body in the event definition. For example:

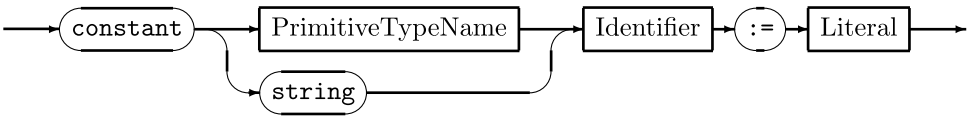
```
event Circle
{
   float radius;
   location position;
   action area () returns float
   {
      return (float.PI * radius * radius);
   }
   action circumference () returns float
   {
      return (2.0 * float.PI * self.radius);
   }
}
```

Provided variables

Specifying named constant values

A constant is a named literal and its value cannot be changed during runtime.

Constant



You can declare an identifier for a constant value in an event type definition or in a monitor. A constant appears in memory once. Spawning a monitor that contains a constant does not make copies of the constant.

The type of a constant must be `boolean`, `decimal`, `float`, `integer`, or `string`.

The name you assign to a constant must be unique within the event type or monitor that contains the constant definition.

The literal that you assign to the constant must be the specified type.

When you define a constant event field, you can refer to that constant from outside the event. Qualify the name of the constant with the event name, for example, `MyEvent.myConstant`.

You cannot declare a constant in an action, directly in a package, or in a custom aggregate function.

Variables

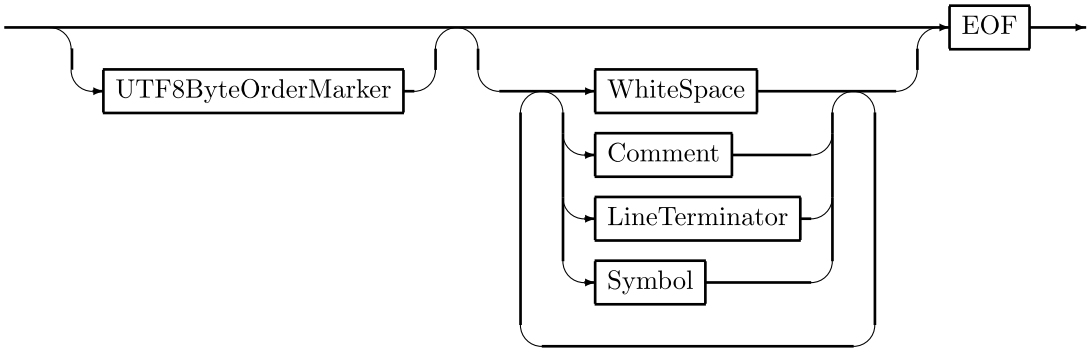
Chapter 9: Lexical Elements

The lexical rules of the EPL grammar describe how sequences of characters are used to form the basic elements of the language, that is, identifiers, constants (string, numeric, and so on), operators, separators, white space, comments, and language keywords. These elements, after discarding any white space and comments, form the symbols used in the syntactical grammar of the language.

Program text

A program's source text is composed of an optional UTF-8 byte-order marker followed by characters that form a sequence of symbols, white space, comments, and line terminators, up to the end of file (denoted by the EOF symbol).

Program Text



UTF8ByteOrderMarker

The UTF-8 byte order marker is a sequence of three consecutive bytes with the values `0xEF`, `0xBB`, and `0xBF` respectively, appearing at the beginning of a file containing EPL source text. The UTF-8 character encoding format does not need a byte-order marker to indicate the byte order because UTF-8 is by definition a bytewise encoding. A UTF-8 byte-order marker at the start of a file just indicates that the program text is encoded in the UTF-8 format. It is inserted automatically by some text editors, such as Notepad on Windows systems.

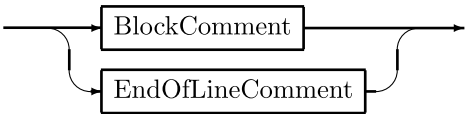
A program's source text can be encoded as Unicode UTF-8, as 7-bit ASCII (which is a proper subset of UTF-8), or various other encodings. The comiler will convert the source text from the locale's encoding to UTF-8 if necessary. In practice, this really only affects comments, white space, and string literals because all other EPL constructs are limited to the ASCII subset. "Identifiers" on page 168, for example, are limited to only a few of the many possible Unicode characters.

Lexical Elements

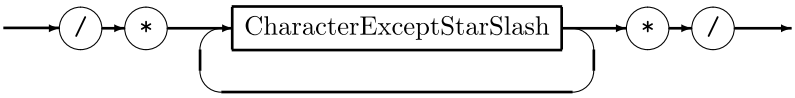
Comments

Comments are explanatory notes or text intended for human readers to help them unerstand what a program or section of a program does.

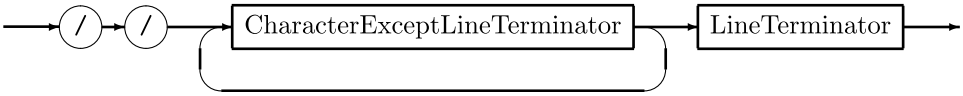
Comment



BlockComment



EndOfLineComment



There are two kinds of comments: block comments and end-of-line-comments.

Block comments begin with the character sequence slash-asterisk `/*`, which is followed by any number of other characters and line breaks, followed by a closing asterisk-slash `*/` sequence. The entire contents of all block comments are ignored.

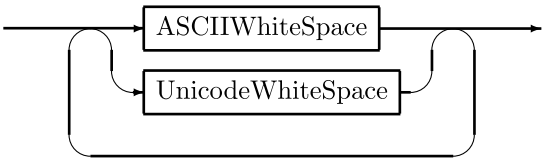
End-of-line comments begin with two consecutive slash characters `//` followed by any number of characters up to and including the end of the current line. The entire contents of all end-of-line comments are ignored.

Lexical Elements

White space

White space characters are characters such as spaces and tabs that are used between symbols to separate them. White space characters are sometimes required between symbols when they would otherwise be misinterpreted or unrecognizable. For example, the symbol `/` is used as the division operator and the symbol `*` is used as the multiplication operator, but the character pair `/*` with no white space between them marks the beginning of a block comment.

WhiteSpace



Though they act as separators between symbols, white space characters are otherwise ignored and discarded during program compilation.

Judicious use of white space improves a program's readability.

The ASCIIWhiteSpace characters and their encodings are listed below:

Table 1. ASCIIWhiteSpace characters and their encodings

Code Point	UTF-8 Encoding	ASCII Encoding	Name
`0x0020`	`0x20`	`0x20`	Space
`0x0009`	`0x09`	`0x09`	Horizontal Tab
`0x000c`	`0x0c`	`0x0c`	Form Feed
`0x001c`	`0x1c`	`0x1c`	File Separator
`0x001d`	`0x1d`	`0x1d`	Group Separator
`0x001e`	`0x1e`	`0x1e`	Record Separator
`0x001f`	`0x1f`	`0x1f`	Unit Separator

The UnicodeWhiteSpace characters, as defined by the Unicode character dictionary, and their encodings are listed below:

Table 2. UnicodeWhiteSpace characters and their encodings

Code Point	UTF-8 Encoding	Name
`0x0085`	`0xc2 0x85`	unnamed control character
`0x00a0`	`0xc2 0xa0`	NO-BREAK SPACE
`0x1680`	`0xe1 0x9a 0x80`	OGHAM SPACE MARK
`0x180e`	`0xe1 0xa0 0x8e`	MONGOLIAN VOWEL SEPARATOR

Code Point	UTF-8 Encoding	Name
0x2000	0xe2 0x80 0x80	EN QUAD
0x2001	0xe2 0x80 0x81	EM QUAD
0x2002	0xe2 0x80 0x82	EN SPACE
0x2003	0xe2 0x80 0x83	EM SPACE
0x2004	0xe2 0x80 0x84	THREE-PER-EM SPACE
0x2005	0xe2 0x80 0x85	FOUR-PER-EM SPACE
0x2006	0xe2 0x80 0x86	SIX-PER-EM SPACE
0x2007	0xe2 0x80 0x87	FIGURE SPACE
0x2008	0xe2 0x80 0x88	PUNCTUATION SPACE
0x2009	0xe2 0x80 0x89	THIN SPACE
0x200a	0xe2 0x80 0x8a	HAIR SPACE
0x2028	0xe2 0x80 0xa8	LINE SEPARATOR
0x2029	0xe2 0x80 0xa9	PARAGRAPH SEPARATOR
0x202f	0xe2 0x80 0xaf	NARROW NO-BREAK SPACE
0x205f	0xe2 0x81 0x9f	MEDIUM MATHEMATICAL SPACE
0x3000	0xe3 0x80 0x80	IDEOGRAPHIC SPACE

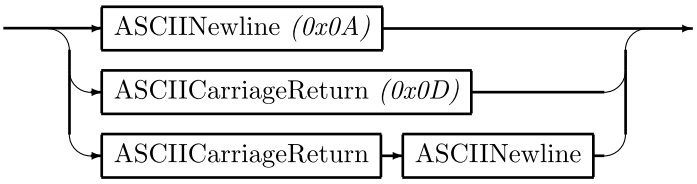
All white space characters appearing between two symbols are ignored. However, note that white space appearing within string literals is not igored. See "Literals" on page 178.

Lexical Elements

Line terminators

Line terminators are used to mark the end of a line of source text. Different operating systems use different characters or character sequences to mark the end of a line.

LineTerminator



The following terminators are used on various operating systems:

Operating System	Line Terminator
Mac OS X	ASCII Carriage Return (`0x0D`)
UNIX	ASCII Newline (`0x0A`)
Linux	ASCII Newline (`0x0A`)
Windows	ASCII Carriage Return (`0x0D`) followed by ASCII Newline (`0x0A`)

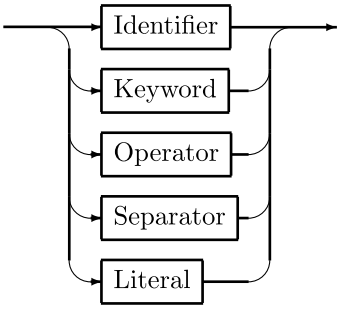
In general, any number of line terminators can be used between any two symbols in a program and they are treated the same as other white space. A line terminator appearing at the end of an end-of-line comment terminates the comment.

Lexical Elements

Symbols

Symbols (also called tokens, atoms, or lexemes) are the elements and words of the language, consisting of identifiers, keywords, operators, separators, and literals. Symbols are composed of one or more characters, excluding white space, comments, and line terminators.

Symbol



Sometimes you must use at least one white space character between two symbols in order to make them distinguishable from each other and from another symbol. For example, the symbol `>>` is the right-shift operator and the symbol `>` is used to indicate the end of the element type in a sequence declaration. Since you can have a sequence of sequences, such a declaration could have two adjacent symbols. Since `>>` in a sequence declaration looks just like the right-shift operator, you have to write them with a white space character between them, thusly: `> >`. On the other hand, the expression `a-b` (subtract the value of the variable named `b` from the value of the variable named `a`) is unambiguous

and no extra white space characters are needed. If you wrote it as `a - b` it would mean the same thing.

Lexical Elements

Identifiers

An identifier is a sequence of allowed characters

Characters

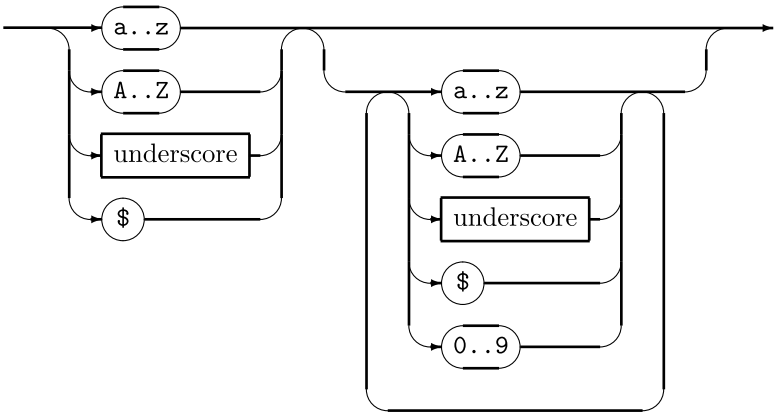
An identifier is a character sequence composed of a combination of the following characters:

- The 26 letters of the Roman alphabet in upper and lower case
- Digits `0` through `9`
- Underscore (_) character
- Dollar sign (`$`) character

The first character may not be a digit. Identifiers are case sensitive. An identifier cannot have the same spelling as a keyword. For example, the word `action` is a keyword and cannot be used as an identifier. See "Lexical Elements" on page 163 for a list of the EPL keywords.

The length of an identifier is limited by available memory. In practice, this means you can make them as long as you want, but very long identifiers are hard to type and harder to read.

Identifier



Lexical Elements

Keywords

In EPL, reserved words are referred to as keywords. You must escape them to use them as identifiers in your code.

Relevant topics

- "List of EPL keywords" on page 169

- "List of identifiers reserved for future use" on page 170
- "Escaping keywords to use them as identifiers" on page 171

Lexical Elements

List of EPL keywords

The table below lists the reserved words called keywords. EPL keywords are case sensitive. You cannot use keywords as identifiers in EPL programs unless you prefix them with a hash symbol (#). Some keywords are flagged with an asterisk (*). You can safely use these keywords outside the scope of a stream query. Inside a stream query, you cannot use these keywords as identifiers unless you prefix them with a hash symbol (#). See "Escaping keywords to use them as identifiers" on page 171.

Table 3. EPL keywords

action	aggregate	all	and
as	at	boolean	bounded
break	by *	call	catch
chunk	completed	constant	context
continue	currentTime	decimal	dictionary
die	else	emit	enqueue
event	every *	false	float
for	from	group *	having *
if	import	in	integer
join *	largest *	location	log
monitor	new	not	on
optional	or	package	partition *
persistent	print	retain *	return
returns	route	rstream *	select *
send	sequence	smallest *	spawn
static	stream	streamsource	string
then	throw	to	true

try	unbounded	unique *	unmatched
using	wait	where *	while
wildcard	with *	within	xor

Some reserved keywords are actually operators. Nevertheless, the restriction still applies. Some Apama tools, such as the Event Modeler, generate code based on EPL and in such code there might be symbols that resemble identifiers but contain hash (#) characters, which are not allowed in identifiers. These "identifiers" are placeholders that are later replaced with valid identifiers that do not contain the hash character.

The `string join()` method is still supported. That is, you can still use the following and you do not receive a warning: `string.join()`. Also, note that the `join` keyword has a query scope and `join` is also a reserved word for use outside queries in a future release.

Note that `ondie`, `onload`, and onunload are not reserved keywords. They are the names of special actions. While you can use "`ondie`", "`onload`", and "`onunload`" as identifiers, doing so is not recommended.

Keywords

List of identifiers reserved for future use

EPL might use the identifiers listed in the table below as keywords in a future release. In this release, if you use one of these reserved words, the correlator logs a warning.

List of reserved identifiers

In this table, some identifiers are flagged with as asterisk (*). These identifiers are reserved as keywords only within stream queries. That is, the correlator logs a warning only if you use this identifier inside a stream query. To use one of these identifiers inside a stream query without logging a warning, prefix it with a hash symbol (#). See "Escaping keywords to use them as identifiers" on page 171.

Table 4. Identifiers reserved for future use

abstract	ALL *	AND *
assert	bignum	BY *
byte	case	char
class	default	enum
EQUALS *	eval	EVERY *
except	extends	FALSE *
finally	FROM	GROUP *

HAVING	immutable	implements
IN *	instanceof	interface
join	JOIN	LARGEST *
native	NOT *	null
OR *	otherwise	PARTITION *
private	protected	public
RETAIN *	RSTREAM *	runtime
SELECT *	SMALLEST *	sortedsequence
switch	sync	SYNC *
synchronized	table	throws
transient	TRUE *	UNIQUE *
void	volatile	WHERE *
window	WITH *	WITHIN *

Keywords

Escaping keywords to use them as identifiers

You can use a keyword as an identifier if you escape it with a hash symbol (#). For example:

```
package com.company.#monitor.client;
using com.company.#monitor.server.Event;
```

In a stream query, you can use a query-scope keyword as an identifier if you prefix it with a hash symbol (#). For example:

```
event Tick
{...
   string partition;
   ...
}
from t in all Tick() partition by t.#partition retain 5 ...
```

You can define a JMon event type that has a field name that is the same as an EPL keyword. To refer to that field in EPL, prefix it with a hash symbol. For example:

```
class MyEvent extends Event {
   int integer;
   ...
}
on all MyEvent(#integer = 5): m { ... }
```

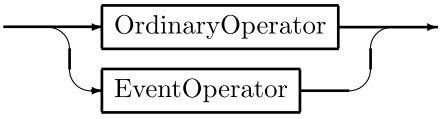
To avoid warning messages if you use a reserved word as an identifier, escape the reserved word with a hash symbol (#).

Keywords

Operators

Operators are symbols used in expressions and statements to perform a computation on or test a relation between data values or, in event expressions, to detect sequences and patterns of events. As you will see, the same symbol is sometimes used for different operations, depending on the context in which the operator is used. For example, the `and` operator is used both in logical expressions, and event sequencing and the `*` operator is used both for integer and floating point multiplication and to match any value in event templates.

Operator



This section discusses the following topics:

See also:

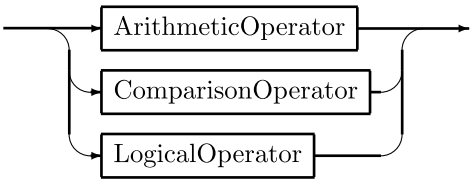
- "Event expression operator precedence" on page 96
- "Expression operator precedence" on page 143

Lexical Elements

Ordinary operators

The ordinary operators are used in primary and bitwise expressions. See "Expressions" on page 130 to perform calculations and comparisons on variables, data values, and other constructs. "Types" on page 17 provides information about the operators that you can use with values of each type.

OrdinaryOperator



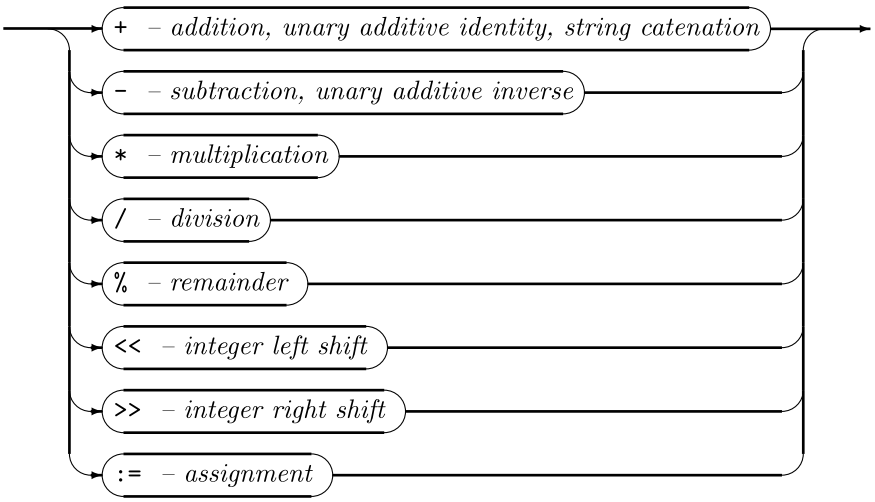
The ordinary operators are grouped into three subcategories: arithmetic, relational, and logical.

Operators

Arithmetic operators

Arithmetic operators are for use in expressions.

ArithmeticOperator

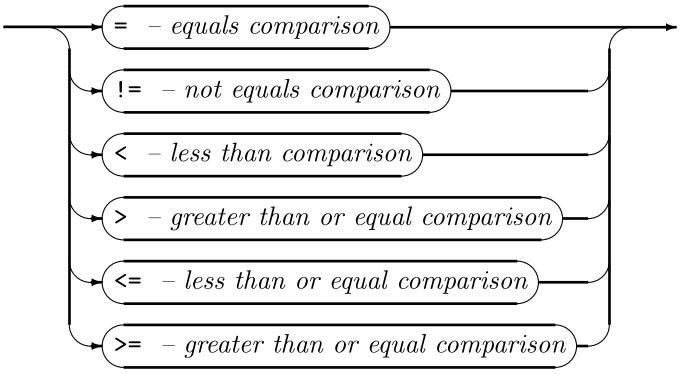


Ordinary operators

Comparison operators

Comparison operators are for use in expressions.

ComparisonOperator

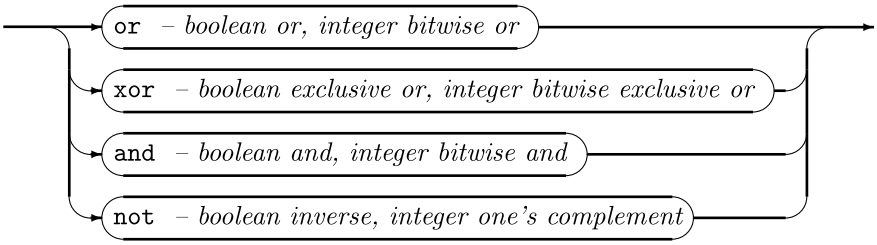


Ordinary operators

Logical operators

Logical operators are for use in expressions.

LogicalOperator

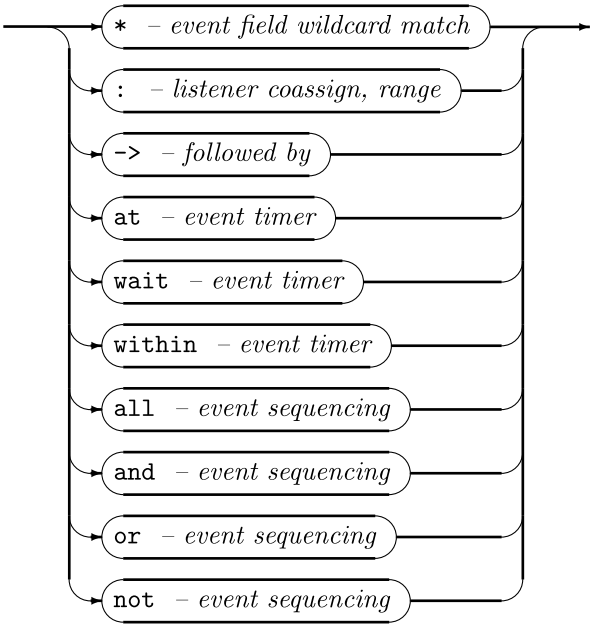


Ordinary operators

Event operators

Event operators are special operators that are used in the `on` statement's event expression. An `on` statement defines an event listener. See "Event expressions" on page 89.

EventOperator



Operators

Expression operators

Another way to categorize the operators supported in EPL is as as expression operators and field operators. You can use the following operators wherever you can specify an expression. Note that they are all binary operators.

Table 5. Expression operators

Operator	Operation	Description
+	Addition	Returns a `decimal`, `float` or an `integer` according to the operands, or concatenation in the case of `string` operands
-	Subtraction	Returns a `decimal`, `float` or an `integer` according to the operands
%	Modulus	Returns an integer and is a valid operator only for integers
/	Division	Returns a `decimal`, `float` or an `integer` according to the operands
*	Multiplication	Returns a `decimal`, `float` or an `integer` according to the operands
>	Greater than	Returns a `boolean` value indicating whether the condition expressed is true or false
<	Less than	Returns a `boolean` value indicating whether the condition expressed is true or false
>=	Greater than or equal to	Returns a `boolean` value indicating whether the condition expressed is true or false
<=	Less than or equal to	Returns a `boolean` value indicating whether the condition expressed is true or false
=	Equivalence	Returns a `boolean` value indicating whether the condition expressed is true or false
!=	Not equals	Returns a `boolean` value indicating whether the condition expressed is true or false
or	Logical `or`, Bitwise `or`	On `boolean` types, On `integer`s
and	Logical `and`, Bitwise `and`	On `boolean` types, On `integer`s
xor	Logical `xor`, Bitwise `xor`	On `boolean` types, On `integer`s
not	Logical `not`	On `boolean` types

Operators

Field operators

Field operators can appear within event templates to define a field value.

Description

The `on` keyword creates an event listener that watches the series of events processed by the correlator for individual events or patterns of particular events. You define the sequence of interest in an event expression made up of one or more event templates. The first part of an event template defines the event type of the event the event listener is to match against, while the section in brackets describes further filtering criteria that must be satisfied by the contents of events of that type for there to be a match. Event template field operators define what values, or range of values, are acceptable for a successful event match.

The value that a field operator applies to can be the result of an expression. Therefore, it is possible to have `>`, `<`, `>=`, `<=` , and/or `=` present in both their roles, as expression operators and as field operators, within an event template. This is not a problem, since the latter are unary while the former are binary and the semantics are quite different.

The following table describes the field operators:

Table 6. Field operators

Operator	Description
`[`*value1*`:`*value2*`]`	Specifies a range of values that can match. The values themselves are included in the range to match against. For example: `on stockPrice(*, [0 : 10]) doSomething();` This example will invoke the `doSomething()` action if a `stockPrice` event is received where the price is between 0 and 10 inclusive. You can apply this range operator to `decimal`, `float`, `integer` and `string` types.
`[`*value1*`:`*value2*`)`	Specifies a range of values that can match. The first value itself is included in the range to match against while the second value is excluded from the range to match against. For example: `on stockPrice(*, [0 : 10)) doSomething();` This example will invoke the `doSomething()` action if a `stockPrice` event is received where the price is between 0 and 9 inclusive (assuming the field was of `integer` type). You can apply this range operator to `decimal`, `float`, `integer` and `string` types.
`(`*value1*`:`*value2*`]`	Specifies a range of values that can match. The first value is excluded from the range to match against while the second value is included. For example: `on stockPrice(*, (0 : 10]) doSomething();` This example invokes the `doSomething()` action if a `stockPrice` event is received where the price is between 1 and 10 inclusive (assuming the

Operator	Description
	field was an `integer`). This operator can apply to `decimal`, `float`, `integer` and `string` types.
`(value1:value2)`	Specifies a range of values that can match. The values themselves are excluded from the range to match against. For example: `on stockPrice(*, (0 : 10)) doSomething();` This example will invoke the `doSomething()` action if a `stockPrice` event is received where the price is between 1 and 9 inclusive (assuming the field was of `integer` type).You can apply this range operator to `decimal`, `float`, `integer` and `string` types.
`> value`	All values greater than the value supplied will satisfy the condition for a match.You can apply this operator to `decimal`, `float`, `integer`, and `string` types. When used with a `string`, the operator assumes lexical ordering.
`< value`	All values less than the value supplied will satisfy the condition for a match. You can apply this operator to `decimal`, `float`, `integer`, and `string` types. When used with a `string`, the operator assumes lexical ordering.
`>= value`	All values greater than or equal to the value supplied will satisfy the condition for a match. You can apply this operator to `decimal`, `float`, `integer`, and `string` types. When used with a `string`, the operator assumes lexical ordering.
`<= value`	All values less than or equal to the value supplied will satisfy the condition for a match. You can apply this operator to `decimal`, `float`, `integer`, and `string` types. When used with a `string`, the operator assumes lexical ordering.
`= value`	All values equal to the value supplied will satisfy the condition for a match. You can apply this operator to `decimal`, `float`, `integer`, and `string` types. When used with a `string`, the operator assumes lexical ordering.

`value`	With one exception, only a value equivalent to the value supplied will satisfy the condition for a match. The exception is a `location` type field. A `location` value consists of a structure with four `float`s representing the coordinates of the corners of the rectangular space being represented. A listener that is watching for a particular value for a `location` field matches when it finds a `location` field that *intersects* with the `location` value specified in the listener's event expression. In the following example, the listener matches each `A` event whose `loc` field specifies a location that intersects with the square defined by `(0.0, 0.0, 1.0, 1.0)`.
	`location l := location(0.0, 0.0, 1.0, 1.0);`
	`on all A(loc = l) ...`

Operator	Description
*	Any value for this field satisfies the condition for a match.

Operators

Separators

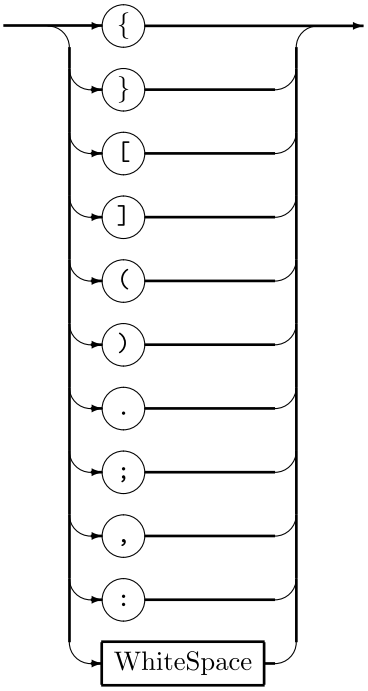
Separators are symbols that are used in certain statements and expressions.

Separator uses

Separators are used to:

- Keep the various parts from bumping into each other, for example commas between parameter values in an action call
- Group related elements together, for example the left and right braces at the beginning and end of a block of statements

Separator

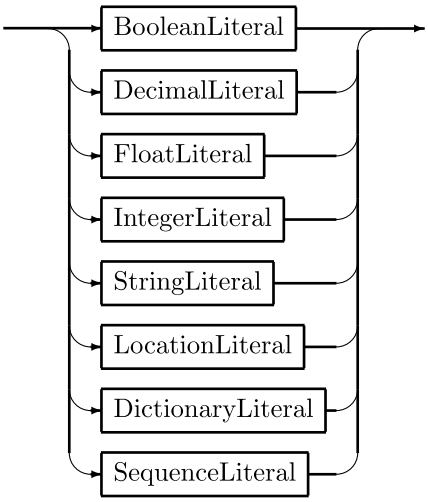


Lexical Elements

Literals

A literal is a source text representation of a constant value of a primitive type, or a `location`, `dictionary`, or `sequence` type.

Literal



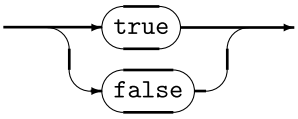
You might want to declare a constant for a frequently used literal so that you can refer to it by name. See "Specifying named constant values" on page 162.

Lexical Elements

Boolean literals

There are two Boolean literal values: `true` and `false`.

BooleanLiteral



Example

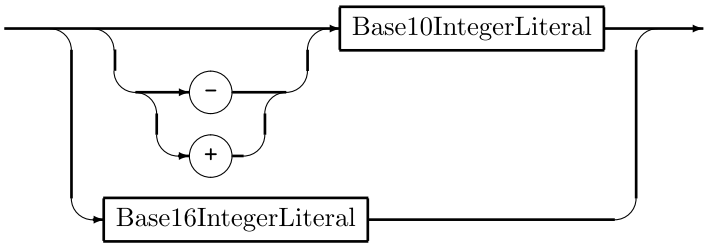
```
a := true;
b := false;
```

Literals

Integer literals

Integer literal values can be written either base 10 (decimal) or base 16 (hexadecimal).

IntegerLiteral

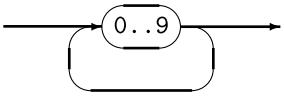


Literals

Base 10 literals

Base 10 integral literal values are a sequence of one or more of the digits `0` through `9`.

Base10IntegerLiteral



Examples

```
i := 0;
i := 11;
i := 1023;
i:= 9223372036854775807;
```

The value can optionally be preceded by a sign. If the sign is omitted, `+` is assumed.

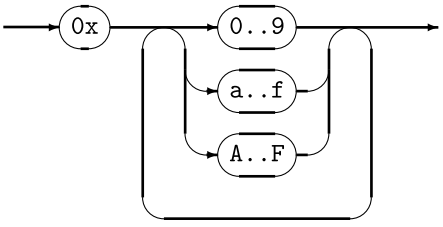
The number 9223372036854775807 or ($2^{63} - 1$) is the largest base 10 integer literal value that can be represented.

Integer literals

Base 16 literals

Base 16 integral literal values begin with the characters `0x`, and consist of a combination of the decimal digits `0` through `9` and the hexadecimal digits `a` through `f` and `A` through `F`.

Base16IntegerLiteral



Examples

```
j := 0x0;
j := 0x0d;
j := 0x0aFF;
j := 0x7fffffffffffffff;
```

The number `0x7fffffffffffffff` or (2^{63} `- 1`) is the largest base 16 integer literal value that can be represented.

You cannot specify a negative hexadecimal literal. The correlator treats hexadecimal literals as unsigned integers. For example, the following is illegal:

```
-0x43af
```

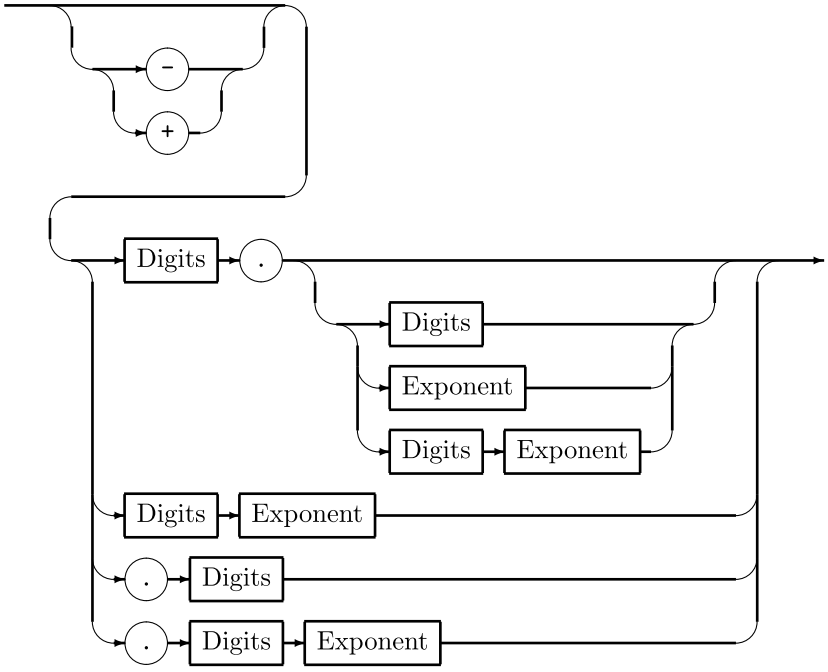
Integer literals

Floating point and decimal literals

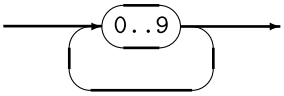
There are several forms of floating point literals.

DecimalLiteral

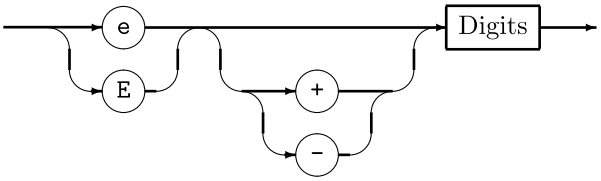
FloatLiteral



Digits



Exponent



Floating-point literal values can take one of the following forms:

- Optional sign, integer digits followed by an exponent.
- Optional sign, integer digits, a decimal point, and an optional exponent,
- Optional sign, integer digits, a decimal point, fraction digits, and an optional exponent.
- Optional sign, a decimal point, fraction digits, and an optional exponent.

If the sign is omitted, '+' is assumed. If the exponent is omitted, e0 is assumed.

The exponent is the letter 'e' followed by an optional sign, and one or more decimal digits.

Examples

```
f := 0.0;
f := 1.;
f := 200128.00005
f := 3.14159265358979;
f := 1e4;
f := 1e-4;
f := 10000e0;
f := .1234;
f := .1234e4;
f := 1.E-32;
f := 1.E-032;
f := 6.0221415E23;
f := 1.7976931348623157e308;
```

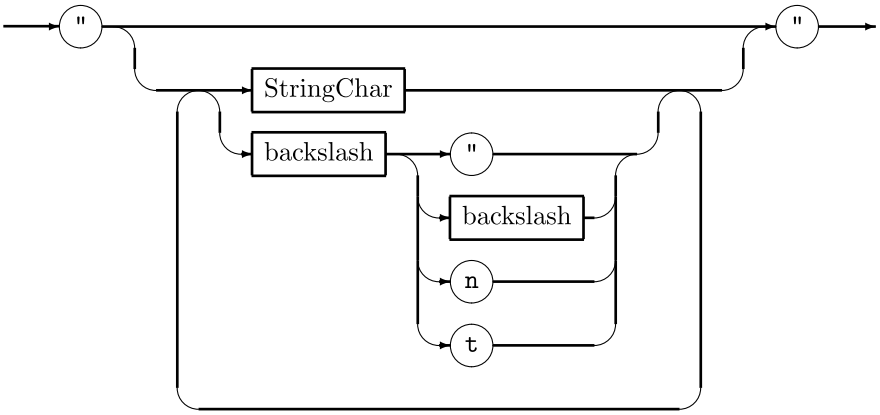
The largest positive floating point literal value that can be represented in EPL is `1.7976931348623157 * 10^308`. The smallest positive nonzero value that can be represented is `2.2250738585072014 * 10^-308`. If you write a floating-point literal whose value would be outside the range of values that can be represented, the compiler raises an error.

Literals

String literals

A string literal is a sequence of characters enclosed in double quotes.

StringLiteral



The backslash character is used as an escape character to allow inclusion of special characters such as newlines and horizontal tabs.

The symbol StringChar above means all the characters other than the ASCII doublequote " and the ASCII backslash `\`.

To include a double quote in a string literal, precede it with a `\` character which serves as an escape character, which means "do not treat this quote as the end of the string literal". To include a newline, use `\n`. To include a tab character, use `\t`. To include a single `\` character, use two: `\\`. The compiler will remove the extra backslashes.

Examples

```
s := "Hello, World!";
s := "\ta\tstring\twith\ttabs\tbetween\twords";
s := "a string on\n two lines";
s := "a string with \\ a backslash and a \" quote";
```

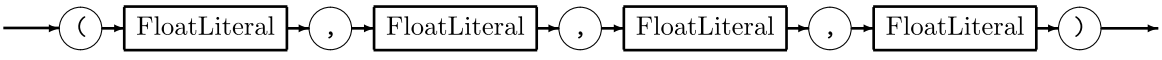
The length of a string literal is limited only by available memory at compile time andruntime. In practice, this means you can make them as long as you need.

Literals

Location literals

The four FloatLiterals form the location's corner point coordinates, `x1`, `y1` and `x2`, `y2`.

LocationLiteral

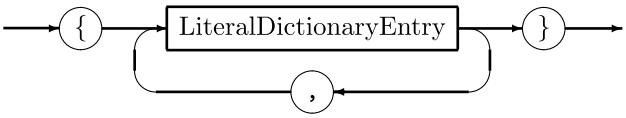


Literals

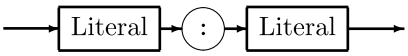
Dictionary literals

Each LiteralDictionary entry in the comma-separated list forms one dictionary entry consisting of a key value and item value pair.

DictionaryLiteral



LiteralDictionaryEntry



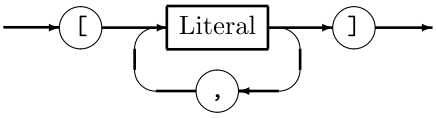
The first Literal in a dictionary entry is the key value and the second is the item value. All key values must be the same type. All item values must the same type. Both must be of a type that matches the types specified in the dictionary variable's definition.

Literals

Sequence literals

Each Literal in the comma separated list forms one entry in the sequence literal. The types must all be identical and must match the sequence type.

SequenceLiteral



Literals

Names

Names are used in EPL programs to refer to the various different kinds of entities in the program. Actions, variables and reference variable members, parameters, monitors, methods, aggregate functions, events, packages, and plug-ins all have names.

Description

Names are either simple or qualified. Simple names consist of a single identifier. Qualified names consist of a sequence of identifiers separated by . symbols.

Every name has a scope, which is the part of a program's text where the name can be used as a simple identifier. The scope is determined by where in the program the name is declared. See "Variable scope" on page 158.

Do not create EPL structures in the `com.apama` namespace. This namespace is reserved for future Apama features. If you do inadvertently create an EPL structure in the `com.apama` namespace, the

correlator might not flag it as an error in this release, but it might flag it as an error in a future release.

Name Precedence

When there are duplicate unqualified names for types, the correlator searches for the associated definition in the following order, and uses the first one it finds:

1. The monitor-internal type definitions, for example, event type definitions and custom aggregate function definitions
2. Definitions that have been brought in with a `using` declaration in the current file
3. Definitions in the current package (this could be the root namespace if a package was omitted)
4. The root namespace

If you try to create a package-level type that has the same name as a definition brought in with a `using` declaration, it causes a compiler error and the code does not inject. For example:

```
package foo;
using bar.Bar;
event Bar { // Causes an error when injecting as Bar has already been
            // defined by a "using" declaration.}
```

You cannot define a type that has the same fully-qualified name as another type.

If two types have the same name but are in different packages, either one can take precedence over the other depending on their ordering in the precedence list. The correlator uses the first match it finds even if that results in an error when a lower-priority match would have worked. For example:

```
X x;
```

This causes an error if, for example, there is an aggregate function called `x` in the current package even if there is an event type called `x` in the root namespace.

Lexical Elements

Chapter 10: Limits

EPL enforces the limits described in the following table.

EPL Limit	Value
Lowest integer	-2^{63} (–9223372036854775808)
Highest integer	$2^{63} - 1$ (9223372036854775807)
Integer precision	64 bits (about 18 decimal digits)
Maximum integer left shift	63 bits
Maximum integer right shift	63 bits
Lowest negative floating point value	$-1.7976931348623157 \times 10^{308}$
Highest negative nonzero floating point value	$-2.2250738585072014 \times 10^{-308}$
Lowest positive nonzero floating point value	$2.2250738585072014 \times 10^{-308}$
Highest positive floating point value	$1.7976931348623157 \times 10^{308}$
Floating point precision	About 15 decimal digits
Lowest negative decimal floating point value	$-9.999999999999999 * 10^{384}$
Highest negative nonzero decimal floating point value	-10^{-398}
Lowest positive nonzero decimal floating point value	10^{-398}
Highest positive decimal floating point value	$9.999999999999999 * 10^{384}$
Decimal precision	Exactly 16 decimal digits
Maximum identifier length	Limited by available memory
Maximum number of entries in a sequence	Limited by available memory
Maximum number of entries in a dictionary	Limited by available memory
Maximum number of characters in a string	Limited by available memory
Maximum number of active listeners	Limited by available memory, typically many tens of thousands

EPL Limit	Value
Maximum number of active monitors	Limited by available memory
Maximum number of fields in an event	2^{16} (65536)
Maximum number of actions in an event	2^{16} (65536)
Maximum indexed fields in an event	32
Memory address space available to EPL runtime	On 32-bit systems, limited to about two gigabytes. The correlator stops if it runs out of memory.
Maximum number of active stream queries	Limited by available memory
Maximum stream window size	Limited by available memory

Chapter 11: Obsolete Language Elements

As EPL has evolved, some older language constructs have been supplanted by more useful and flexible ones. The new constructs can accomplish the same effects and more and their use is preferred. Nevertheless, existing programs may still use the obsolete constructs, which are described in this section.

Old style listener calls

Do not specify the following:

```
on A() foo;
```

Instead, specify the following:

```
on A() foo();
```

Obsolete Language Elements

Old style spawn statements

Do not specify the following:

```
spawn actionName;
```

Instead, specify the following:

```
spawn actionName();
```

Obsolete Language Elements