Search documentation

Appian

Expressions

Appian 7.7

Toggle Us

An expression is a statement evaluated by the rules engine to determine its value. It's composed of any valid combination of literals, operators, functions, and variables.

The construction of an Appian expression is similar to Excel formulas.

It operates upon Appian data and external data integrated with Appian, such as process variables, node inputs, constants, CDTs, and process proper identified in the expression. It does not operate on data outside of the domain or context in which the expression is running.

You can create an expression by typing it directly into an expression field or through the Expression Editor. Expression fields are marked by an Expression Editor icon.

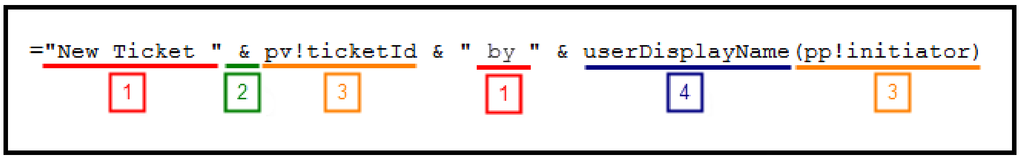
Tu

Re

Ot

Parts of an Expression

Below is an expression with its various parts labeled.



```
="New Ticket " & pv!ticketId & " by " & userDisplayName(pp!initiator)
```

1. **Literal Values**: The values "New Ticket " and " by " are text literals and return in the expression output as written without the quotes.
2. **Operator**: The & operator represents text concatenation and combines text with data.
3. **Variables**: The `pv!ticketId` variable returns the Ticket ID value, and the `pp!initiator` variable returns the user who started the process.
4. **Functions and Rules**: The `userDisplayName` rule takes a user as a parameter and returns the user's first and last name.

The above expression could result in the following: `New Ticket AN-9867 by John Smith`

More information on using these different parts appears in the sections below.

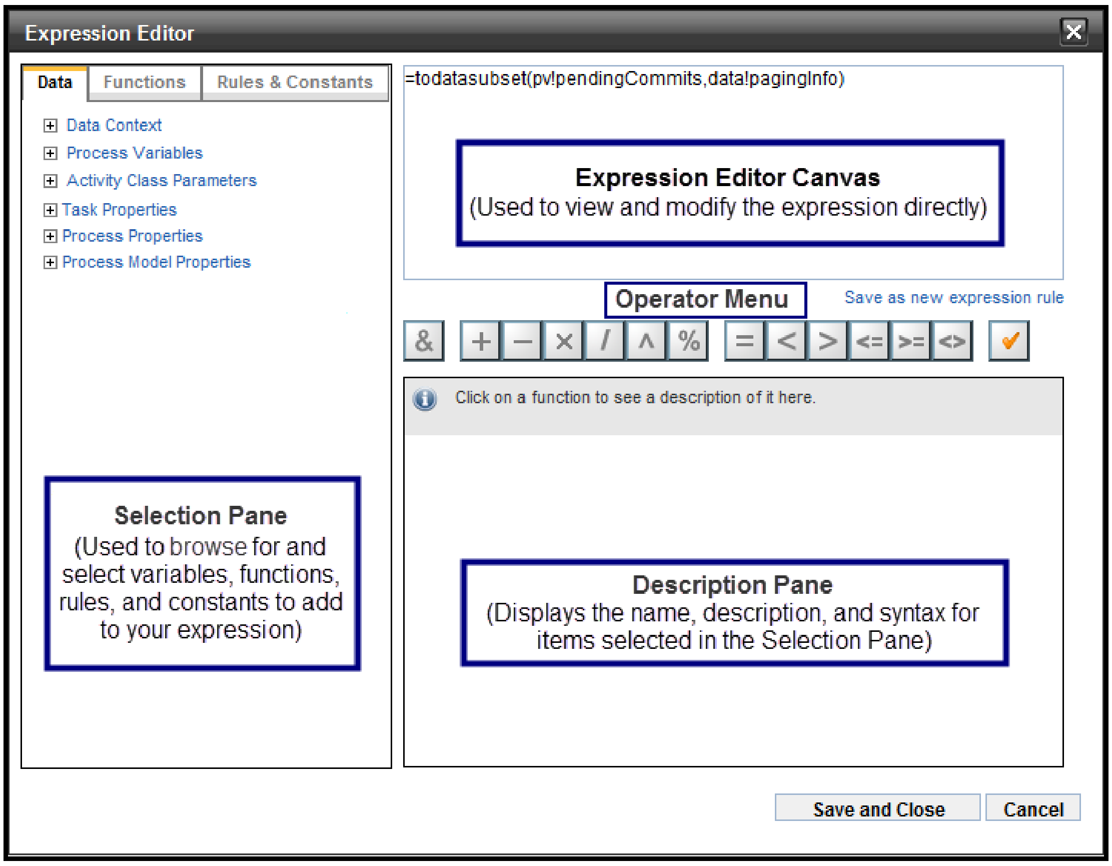
NOTE: An expression must always start with an equal sign (=).

Expression Editor

The Expression Editor provides a simple, point-and-click interface for writing expressions.

To access the editor, click the Expression Editor icon next to an expression field or in a rich text editor (where applicable).

The Expression Editor (as shown below) displays in a new window.



Expression Editor

=todatasubset(pv!pendingCommits,data!pagingInfo)

Expression Editor Canvas
(Used to view and modify the expression directly)

Operator Menu

Selection Pane
(Used to browse for and select variables, functions, rules, and constants to add to your expression)

Description Pane
(Displays the name, description, and syntax for items selected in the Selection Pane)

Click on a function to see a description of it here.

To write an expression through the Expression Editor, complete the following:

1. Browse through items in the **Selection Pane** and click them to add them to your expression.
   - Hover your mouse over an item to view information on it in the **Description Pane**.
2. Add operators by clicking them in the **Operator Menu**.
3. Type in any additional values to the **Expressions Editor Canvas**.
4. Click the **checkmark button** to validate the expression.
   - Expressions must pass validation in order to be saved.
5. Click **Save and Close** to save your expression and exit the editor.

Data Tab The Data tab lists any variables available for use in your expressions. Options depend on what you are creating the expression for (such as a form component configuration versus a report metric configuration).

For more information on using variables in expressions, see Variables.

Functions Tab The Functions tab includes all Appian Functions as well as any Custom Function Plug-ins added to a function category.

Rules & Constants Tab The Rules & Constants tab includes any expression rules, query rules, and constants you have viewer rights to.

Expression Best Practices

This section describes a recommended approach that follows Appian Best Practices.

Saving an Expression as an Expression Rule

When you write an expression that can be applied in multiple places around your system, such as one that retrieves information on the most recent meeting event, it is a best practice to create a rule out of it.

An expression rule is a stored expression with a user-defined name, description, definition, and a central location that can be reused within any expressic field. They are accessible through the Expression Editor or can be typed into an expression.

See also: Rules

Saving a Literal Value as a Constant

Similar to expression rules, instead of defining the same literal value multiple times in your system, such as the number of days until a deadline, define th literal value as a constant.

See also: Constants

Literal Values

A literal is a static value stated in the expression, such as 2, 3.14, or "hello".

The following types of literals are supported:

Text String

Expresses a series of text characters. It is stated as text entered between double quotation marks.

- Example: `"Hello World"`

Integer Number

An integer number can be any whole number.

- Example: `82` or `-82`

Decimal Number

A decimal number possesses a decimal point and can express fractional numbers.

- Example: `1.234`

Special Literals

Certain built-in expression values exist in order to concisely express concepts or conditions. Supported special literals include:

- `true` for the Boolean value true.
- `false` for the Boolean value false.
- `null` for an empty (null) value that has no specific data type.

It can be used within any variable as a default value and can be cast to any data type.

Arrays

An array is an ordered list of data items that can be selected by indices computed at run-time. Arrays can be empty or constructed with literal values, variables, or functions.

- Only one-dimensional arrays can be specified.
- Nested arrays are flattened to a one-dimensional array.
  - Example: `{{1, 2}, {3, 4}} = {1, 2, 3, 4}`
- For multi-dimensional arrays, create a complex data type.

When entering an array as part of an expression, you must use braces ({}) to enclose it and separate items using commas (,) unless you are referencir a variable.

- Example: `{2, 3, 9, 1}`

To add text in an array, enclose each item in quotation marks ("").

- Example: `{"a", "b", "c", "d"}`

Accessing Array Items at an Index

In order to select one or more values from an array, use the **index operator []** or **index()**.

For example, with an index operator and an array of `pv!multiple = {10, 20, 30}`:

> `pv!Multiple[2]` yields `20` `pv!Multiple[{2, 3, 2}]` yields `{20, 30, 20}`

- Example with index() and array of `{10, 20, 30}`:

> `index({10, 20, 30}, 2, 1)` yields `20`

Things to consider:

- All arrays are indexed starting at one (1).
- An integer expression can be used in place of a single integer or array of integers.
- Variables and constants with a number (integer) data type or text data type are also accepted by the index operator.

See also: Array Functions

Ad-hoc Data Structures

An ad-hoc data structure, also known as dictionary, is a data structure created at run-time using a more general purpose data structure.

You can pass an ad-hoc data structure as an argument to a function or rule when the input type is "Any Type".

For example, to create an ad-hoc structure with two fields called `label` and `value`, enter the following:

```
{label: "Item", value: "Entry"}
```

To create a list of an ad-hoc structure, enter the following:

```
{{label: "Item one", value: "Entry one"}, {label: "Item two", value: "Entry two"}}
```

Date and Time

Data types Date, Time, and Date and Time are stored as numbers.

- **Date** is an integer (days since the epoch date)
- **Time** is an integer (milliseconds since midnight)
- **Date and Time** is a decimal (fractional days since the epoch date)

For date and time values, the local value (based on the designer time zone) is not stored, instead it is stored in GMT and then converted to the user's tir zone when displayed on the screen.

To advance either **Date** or **Date and Time** by a day, add 1 to the value.

To advance **Time** by a minute, add `60*1000` to the value. Adding one to Time only advances it by a millisecond.

Example Input	Yields
`date(2012, 4, 30) - date(2012, 4, 25)`	`5 days`
`datetime(2012, 4, 25, 12) - datetime(2012, 4, 25, 10)`	`0.0833 days –or- 2.0 hours`
`date(2012, 4, 25) + 5`	`4/30/2012`

Operators

There are many operators that can be used in expressions to perform data manipulation. The operators provided are divided into two different categories.

Arithmetic Operators

To Perform...	Use	Example
Addition	+	`10+8` yields `18`
Subtraction/Negation	-	`10–8` yields `2` - OR - if value is `97`, then –value is `–97`
Multiplication	x or *	`2*5` yields `10`
Division	/	`10/5` yields `2`
Exponentiation	^	`2^8` yields `256`
Percentage (divide by 100)	%	`97%` yields `0.97`

Using arithmetic operators with arrays and tointeger():

Example Input	Yields
`{1, 2, 3} * 10`	`{10, 20, 30}`
`{1, 2, 3} + {1, 2, 3} + {1, 2, 3}`	`{3, 6, 9}`
`{1, 1, 1, 1, 1} + {1, 2}`	`{2, 3, 2, 3, 2}`
`tointeger({}) + 1`	`1`

BEST PRACTICE: Apply arithmetic to lists directly, rather than through MNI, to make your process model look cleaner and work faster.

NOTE: If operating on two arrays that are not the same length, the shorter list will be repeatedly extended until it is the same length as the longer list. This applies to Date and Date/Time values as well:

- `pv!list_of_dates+1` returns the next day for the entire list.

Comparison Operators

To Perform...	Use	Example
Less than	<	`10<2` yields `False`
Greater Than	>	`10>2` yields `True`
Less Than or Equal	<=	`10<=2` yields `False`
Greater Than or Equal	>=	`10>=2` yields `True`
Not Equal to	<>	`10<>2` yields `True`
Equal to	=	`10=2` yields `False`

Using comparison operators with arrays, functions, and text inputs:

- Array elements are compared item by item.
- When comparison operators are used on an array, multiple true or false results are returned.
- If you want a single comparison of one list versus another, use the exact function.
- If you need a text comparison that DOES NOT require case-insensitiviy, consider using the `exact()` function rather than the `=` operator to improve performance.
- To test any element in the compared list, enclose the comparison using the or() function.

Example Input	Yields
`1={1, 2, 3}`	`{true, false, false}`
`{1, 2, 3}<>{4, 2, 6}`	`{true,false,true}`
`exact({1, 2, 3},{1, 2, 3})`	`true`
`1=tointeger({})`	`false`
`0=tointeger({})`	`true`
`"Hello"="HELLO"`	`true (case-insensitive)`
`exact("Hello","HELLO")`	`false (case-sensitive)`

SAIL Operators

The only SAIL operator, the save operator represented by `<<`, is now deprecated. The a!save() function should be used to save modified or alternativ values in SAIL `saveInto` fields. Designers familiar with the << operator can refer to the following table to understand how a!save() works:

Save Operator	a!save()
`saveInto: local!a << fn!isnull`	`saveInto: a!save(local!a, isnull(save!value))`
`saveInto: local!a << append(ri!list, _)`	`saveInto: a!save(local!a, append(ri!list, save!value))`
`saveInto: { local!a << append(ri!list, _), local!b << rule!ucReturnFirstInput(2, _) }`	`saveInto: { a!save(local!a, append(ri!list, save!value)), a!save(local!b, 2) }`

See also: SAIL Components and SAIL Design: Saving User Inputs

For information on the save operator, refer to its entry in the 7.5 documentation.

Operator Precedence

The precedence of operators evaluated in an expression follows the standard Order of Operations:

1. Operator expressions inside parentheses

2. Exponentiation
3. Multiplication and Division
4. Addition and Subtraction

So, if an expression includes two or more operators, the operator higher on the list is applied first, then the second highest, and so on. In order to ensu an operation occurs before another, enclose it within parentheses.

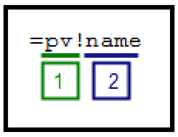
Variables

A variable identifies the data to use when evaluating an expression. It uses syntax similar to an Excel's sheet!cell syntax, where the Appian data domain the sheet and a named variable is the cell syntax. Appian variables are always of a specific data type.

The variables you can use in an expression depend on the context in which the expression is designed and evaluated. They are listed in the Data tab of the Expression Editor and include Appian variables and user-defined process variables.

For a full listing or variables delivered with Appian, see Process and Report Data.

In the following example, the expression returns the value of a process variable:



1. **domain**: Optional if the desired domain is the default domain for the execution context. When a domain isn't specified, it is inferred based on the context in which the expression is evaluated.
2. **name**: Name you assign (or assigned by Appian) to your variable when you create it. It is not case sensitive. If the variable name contains characters other than letters/numbers/underscores or begins with an underscore, the domain and variable name must be enclosed in single quotes. For example, `'pv!variable.a-name'`.

Variables of complex and custom data types use the dot operator to access field values. You can also use the `index()` function.

Let's say you have a custom data type "Person" with the following structure:

```
Person
 |- firstName (Text)
 |- lastName (Text)
 |- homeAddress (Address)
   |- city (Text)
```

You create a variable (process variable, node input, or local variable) called "personA" of type Person. To access the "firstName" value of the variable, enter any of the following:

`index(pv!personA, "firstName", "")` `pv!personA["firstName"]` `pv!personA.firstName`

NOTE: Appian recommends using `index()` to access the index value of a CDT in cases where the variable may have a null value, or the field name is nc available. The `index()` function allows you to assign a default value through its default value parameter. See also: index() function.

The dot notation is useful when accessing nested fields of custom data type, provided that the nested field is not null.

To access the "city" value of the "personA" variable, enter any of the following:

`index(pv!personA, "homeAddress", "city", "")` `pv!personA.homeAddress.city`

Text Concatenation

When writing an expression that includes text and data, use an ampersand (`&`) to string (or concatenate) the information together.

For example, to display a salutation when using process variables for the title and name data:

`="Dear "& pv!title & pv!name &","` returns `Dear John Smith,`

Comments

Comments are a way for designers to leave notes in the expression to explain what the expression does.

You start a comment with `/*` and end it with `*/`. Any content between these two symbols is ignored when the expression evaluates.

For example:

`"Dear "& pv!title /* pv!tile should contain either "Mr" or "Mrs" */ & pv!name &","`

Includes a comment but still evaluates to the following:

> "Dear "& pv!title & pv!name &","

Functions, Rules, and Data Types

Appian Functions, custom function plug-ins, rules, and data types are stored within the Appian system.

You can use Appian Functions, custom function plug-ins, and rules in expressions to perform operations using arguments you pass to them. The expression then returns a result based on your arguments.

You can use data types in expressions to construct a complex data type value by creating a type constructor.

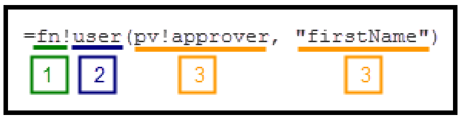
Both options of passing arguments and creating type constructors are explained in detail below.

See also: Appian Functions, Rules, Data Types

Passing Arguments

Functions and rules are defined by their logic and parameters. These parameters accept arguments that determine how the function will evaluate.

For example, the `user()` function retrieves the first name of a user using a process variable and a literal text as arguments through the following syntax



```
=fn!user(pv!approver, "firstName")
```

1. **domain**: Defines where the operation definition is stored. When not specified in the expression, the name is searched for in rules, then Appian Functions, then custom functions.
2. **name**: Given name of the Appian Function, custom function, rule, or data type to use.
3. **arguments**: Values supplied to the function parameters. Supported arguments are literal values, arrays, variables, functions, rules, data types, and expressions. The Expression Editor lists the expected argument types within the function description.

Passing arguments can be done by position or keyword.

By Position

Passing arguments by position is required for Appian functions and custom function plug-ins and is best for rules that take three or fewer arguments.

To pass arguments by position, enter the values in order.

For example, the syntax for the `joinarray()` function is the following:

joinarray(*array*, *[separator]*)

To pass values using positional arguments, enter the following:

```
=joinarray({1,2,3,4},"|")
```

This evaluates with array as `{1,2,3,4}` and separator as `|`, returning `1|2|3|4`.

Required Arguments

You must enter values for every required argument in a function or rule or the expression results in an error. For rules, all inputs are required when passing by position.

Optional Arguments

To pass a value for an optional argument, enter values for all arguments defined before the parameter you're entering a value for, even if they are also optional. If not, the expression may apply the argument to the wrong parameter and result in an error or undesired results.

Optional arguments are surrounded by brackets `[]` in the function documentation.

For example, the `toxml()` function has four parameters, three of which are optional.

toxml(*value*, *[format]*, *[name]*, *[namespace]*)

To use the default for *[format]* and *[namespace]*, but specify a value for *[name]*, you must also configure the *[format]* parameter.

For example:

`=toxml(pv!somePersonNameCDT, false(), "person")` evaluates with the following:

- *value* = `pv!somePersonNameCDT`
- *[format]* = `false()`
- *[name]* = `"person"`
- *[namespace]* = default value

Whereas, `=toxml(pv!somePersonNameCDT, "person")` would evaluate with the following:

- *value* = `pv!somePersonNameCDT`
- *[format]* = `"person"`
- *[name]* = default value

- *[namespace]* = default value

Unlimited Arguments

Some functions take an unlimited number of arguments, such as sum(). This is denoted by an ellipsis in their function description with the Expression Editor.

For example:

sum(*addend*, **...**)

By Keyword

Passing arguments by keyword is only supported in system functions and rules. It is best used for system functions and rules that take four or more arguments.

See also: System Functions

To pass arguments by keyword, specify the name of the parameter, followed by a colon, then the argument value.

Appian recommends entering a line break after each keyword argument.

For example:

A rule called `feedMessageForNewCase` returns the feed message for a case management application using the following definition:

```
="Priority " & ri!priority & ": " & ri!caseSummary & " [#" & ri!caseId & "]"
```

It includes the inputs `priority`, `caseSummary`, and `caseId`.

To pass arguments by keyword, you could enter the following:

```
=rule!feedMessageForNewCase(
  priority: pv!priority,
  caseSummary: pv!summary,
  caseId: pv!id
)
```

To evaluate into the following:

Priority 1: Basic users cannot connect to server [#100005]

Keywords do not need to be in the order the arguments are defined.

For example, the following still results in the text above:

```
=rule!feedMessageForNewCase(
  caseId: pv!id,
  caseSummary: pv!summary,
  priority: pv!priority
)
```

Optional Arguments

All arguments are optional when passing by keyword. If you do not pass an argument for a parameter, the parameter receives a null value of the parameter type.

For example:

```
=rule!feedMessageForNewCase(
  caseId: pv!id,
  caseSummary: pv!summary
)
```

Evaluates with the following:

- *[caseId]* = `pv!id`
- *[caseSummary]* = `pv!summary`
- *[priority]* = `null`

Keyword Requirements

The keyword (or rule input name) for a parameter is defined by the designer. They are searched first by the case-sensitive name, then case-insensitive, such that both rules below will evaluate:

```
=rule!feedMessageForNewCase(
  caseId: pv!id,
  caseSummary: pv!summary
)

=rule!feedMessageForNewCase(
  CASEId: pv!id,
  CASESummary: pv!summary
)
```

If a keyword is not matched with a parameter name, the argument is ignored and the parameter receives a null value.

When specifying the keyword, use single quotes around it if it contains characters other than letters/numbers/underscores or begins with an underscor (similar to namespaces).

For example:

```
=rule!person(
  '_firstName': "John"
)
```

NOTE: Passing arguments by keyword is not supported when creating rules for the Web Content Channel, Portal reports, or events. If used, it will cau the process or task to pause by exception.

Passing Functions, Rules, and Data Types as Arguments

Appian functions, custom functions, rules, and data types are supported as arguments.

The syntax for passing them is similar to passing variables by inserting the object's domain and pointing to its name similar to the following:

- `fn!sum`
- `rule!myrule`
- `type!Person`

For example, passing the `sum()` function to the `reduce()` function:

```
=reduce(fn!sum, 0, {1, 2, 3}) yields 6
```

Passing a rule isnumbereven to the any() function:

```
=any(rule!isnumbereven,{-1,0,1,2}) yields true
```

Passing a data type for a type comparison:

```
=if(typeof(ri!input) = type!User, user(ri!input, "email"), group(ri!input, "groupName"))
```

When using a function, rule, or data type in an expression, remember that function and rule names are NOT case-sensitive, whereas data types ARE case-sensitive.

Specifying the namespace of the data type improves readability but is optional when the name of the data type is unique.

If the data type name is not unique, the system returns a validation error indicating the namespace must be specified. When specifying the namespace, use single quotes around the domain and data type name if the namespace contains characters other than letters/numbers/underscores or begins with an underscore.

For example:

```
 'type!{http://www.appian.com/ae/types/2009}User'
```

Deleted Rules or Deleted Data Types

If you pass a rule or data type to an expression and the rule or data type is later deleted, the expression will still evaluate. If you inspect an import package, however, and it contains expressions that require the deleted rules or data types, the rules or data types will not be listed as missing dependencies.

See also: Deleting Rules and Deleting Data Types

NOTE: The following Appian functions cannot be passed as an argument:

- `and()` function, such as `apply(fn!and . . .)`
- `if()` function, such as `apply(fn!if . . . )`
- `load()` function, such as `apply(fn!load . . . )`
- `or()` function, such as `apply(fn!or . . .)`
- `with()` function, such as `apply(fn!with . . . )`

Constructing Data Type Values

To construct values for complex system and custom data types in an expression, create a type constructor. Type constructors accept an argument fc each data type field and return a value of the specified data type.

Creating a type constructor is similar to passing arguments to functions except the domain is required for the type constructor to evaluate and data type names are case-sensitive.

For example:

If a Person CDT has the following structure:

```
Person
 |- firstName (Text)
 |- lastName (Text)

=type!Person(
  firstName: "John",
  lastName: "Smith"
)
```

returns `[firstName=John, lastName=Smith]`

Entering the namespace of the data type is optional. If the name of the data type is unique, the namespace is looked up when the expression is saved and shown when the expression is viewed again.

For example, enter the following rule and save it:

```
=type!Person(
  firstName: "John",
  lastName: "Smith"
)
```

When you view it again, it shows up as the following:

```
='type!{https://cdt.example.com/suite/types/}Person'(
  firstName: "John",
  lastName: "Smith"
)
```

If the data type name is not unique, the system prompts you to enter the fully qualified name including namespace when saving the expression. Remember to use single quotes around the full name since the namespace contains special characters.

NOTE: Appian recommends using keyword parameters with type constructors as shown in the above example to ease CDT change management.

See above: Passing Arguments

Deleted Data Types

When you save an expression that uses a data type value, and the data type is subsequently deleted, the expression continues to reference the delete data type.

For example, if the data type Person is deleted, the expression in the examples above will show up as the following:

```
='type!{https://cdt.example.com/suite/types/}Person^1'(
  firstName: "John",
  lastName: "Smith"
)
```

See also: Removing Data Types

Optional Arguments

All arguments are optional in a type constructor. Fields that are not assigned a value are set to null.

For example:

```
=type!PagingInfo(
  startIndex: 1,
  batchSize: 2
)
```

returns `[startIndex=1, batchSize=2, sort=]`

Complex Arguments

For fields that are themselves a complex data type, type constructors can be used to define their values.

For example:

When a Person CDT has the following structure:

```
Person
    |- firstName (Text)
    |- lastName (Text)
    |- address (Address)
        |- street (Text)
        |- city (Text)
```

```
=type!Person(
  firstName: "John"
)
```

returns `[firstName=John, lastName=, address=]`

```
=isnull(type!Person(firstName: "John").address)
```

returns `true`

```
=type!Person(
  firstName: "John",
  address: type!Address(
    street: "123 Abc St",
    city: "Reston"
  )
)
```

returns `[firstName=John, lastName=, address=[street=123 Abc St, city=Reston]]`

Best Practice: For readability when creating saving it as a rule, enter line breaks after each argument when passing by keyword.

Extra Arguments

If you enter keywords that don't match any fields in the data type, the keywords are ignored.

NOTE: Type constructors are not supported when creating rules for the Web Content Channel, Portal reports, or events. If used, it will cause the process or task to pause by exception.

Evaluation Order

Functions, rules, and type constructors are evaluated in argument order through nested functions and expressions.

A nested function is a function used as the argument for another function. For example, in the following expression, the `supervisor()` function is nested within the `user()` function:

```
=user(supervisor(pp!initiator), "firstName")
```

The `supervisor()` function is evaluated first, followed by the `user()` function.

If more than one function is used in an expression outside of nested functions, the functions are evaluated left to right. For example, in the following expression, both the `calisworkday()` and `calisworktime()` functions are nested in the `and()` function.

```
and( calisworkday( datetime(2011, 12, 13, 20, 0, 0), "Second"), calisworktime(  datetime( 2011, 12, 13, 20, 0, 0), "Third"))
```

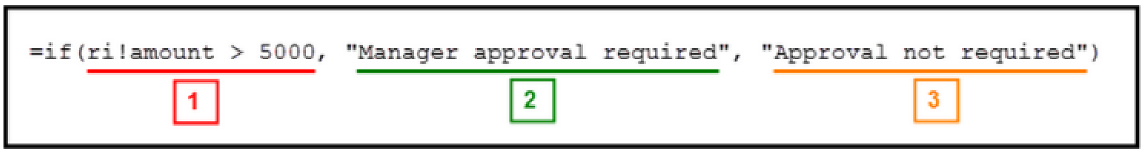
The `calisworkday()` function is evaluated first, followed by the `calisworktime( )` function, and then the `and()` function.

When an expression refers to the same Java Function multiple times with the same arguments, the function is evaluated once. Java Functions include custom functions and Scripting Functions (except for the `linktoxxx()` and `loggedinuser()` functions).

See also: Custom Function Plug-Ins Best Practices

Special Case: if()

When the `if()` function is used in an expression, the condition is evaluated first, then either the true value or the false value.



```
=if(ri!amount > 5000, "Manager approval required", "Approval not required")
```

1. **Condition**: A test that determines whether the true value or the false value will be returned.
2. **True Value**: Value to be returned if the condition evaluates to true.
3. **False Value**: Value to be returned if the condition evaluates to false.

When the condition is an array of booleans, both sides are evaluated. The true values from the "true" side are returned, and the false values from the "false" side are returned.

For example:

```
=if({10, 20, 30, 40, 50} < 30, {10, 20, 30, 40, 50}, {0, 0, 0, 0, 0}) returns {10 ,20, 0, 0, 0}
```

See also: if() function

Advanced Evaluation

The advanced evaluation options below apply mainly to creating dynamic behavior in user interfaces. They work, however, anywhere in the system, except analytics reports and process event nodes.

Partial Evaluation of Rules and Type Constructors

With partial evaluation, you can evaluate arguments in a rule or type constructor and defer complete evaluation of the function until the remaining arguments are available. In computer science publications, this is commonly called partial application or partial function application.

For example, when creating a Tempo report using expressions, you may want to create multiple filters the user can interact with in order to narrow dov results in a grid, such as filtering items to those updated between two dates.

In this case, you would set up the expression with two date components (such as startDate and endDate) each passing its value to a rule or data type that requires multiple arguments. The startDate and endDate inputs would save their values into a query rule that expects both a start date and an enc date to filter by.

Normally, if all required arguments are not passed, the expression would not evaluate and the interace would not render.

In order to have your interface render and later accept arguments to the remaining parameters, you need to set up your function for partial evaluation

To construct a partial function, create the expression as usual and leave an underscore character for arguments to be evaluated later.

Passing Arguments by Position

Deferring one argument: `index(_, {1, 1, 3}, 0)`

Deferring two arguments: `index(_, _, 0)`

Passing Arguments by Keyword

Deferring one argument: `a!pagingInfo(startIndex:_, batchSize:10)`

Deferring two arguments: `a!pagingInfo(startIndex:_, batchSize:_)`

Add the arguments to be evaluated later on to the end of the function and enclose them in parentheses as shown in the examples below:

Filling Blank Arguments by Position

Arguments are applied to the underscores in the order they are listed from left to right.

Partial function that takes one argument:

- `index(_, {1, 1, 3}, 0)({10, 20, 30})`
- `a!pagingInfo(startIndex: _, batchSize: 10)(1)`

Partial function that takes two arguments:

- `index(_, _, 0)({10, 20, 30}, {1, 1, 3})`
- `a!pagingInfo(startIndex: _, batchSize: _)(1, 10)`

Filling Blank Arguments by Keyword

Arguments are applied by matching up their keywords. They can only be passed by keyword to partial functions that are defined by keyword.

For example, `a!pagingInfo(1, _)(batchSize: 10)` is not allowed.

Partial function that takes one argument: `a!pagingInfo(startIndex: _, batchSize: 10)(startIndex: 1)`

Partial function that takes two arguments: `a!pagingInfo(startIndex: _, batchSize: _)(startIndex: 1, batchSize: 10)`

When the blank arguments are filled for the examples above, the functions will evaluate as follows:

Passing Arguments by Position: `index({10, 20, 30}, {1, 1, 3}, 0)`

Passing Arguments by Keyword: `a!pagingInfo(startIndex: 1, batchSize: 10)`

Arguments that make up a partial function are evaluated immediately. For example, in `concat(now(), _)` , `now()` returns the time at which the expressio is evaluated to return a partial function.

Functions with Unlimited Parameters

Functions that take an unlimited number of inputs, such as `sum()` , `product()` , and `difference()` , can also be partially evaluated by using the underscor syntax.

Inputs are applied for each underscore in order, and leftovers are added at the end as shown in the examples below:

```
sum(1, _)(2, 3, 4) returns sum(1, 2, 3, 4)

sum(1, _)(pv!array) returns sum(1, 2, 3, 4) where pv!array is {2, 3, 4}

sum(_, 2)(1, 3, 4) returns sum(1, 2, 3, 4)

sum(_, 2, _)(1, 3, 4) => sum(1, 2, 3, 4)

myrule(_)(1, 2) returns myrule(1, 2)
```

Best Practice: Appian recommends including the exact number of underscores when the number of arguments is known even though multiple arguments can be passed to a partial function when it is marked with only one underscore. This helps with the readability and maintainability of the expression.

For example, in the following expressions:

- Accepted: `myrule(_, 2, _)(1, 3, 4, 5)` which equates to `myrule(1, 2, 3, 4, 5)`
- RECOMMENDED: `myrule(_, 2, _, _, _)(1, 3, 4, 5)` which equates to `myrule(1, 2, 3, 4, 5)`

NOTE: Rules created for the Web Content Channel, operators, and the following functions do not support partial evaluation:

- Operators, such as `"Hello"=_`
- `and()` function, such as `and(isleapyear(1996),_)`
- `if( )` function, such as `if(isleapyear(1996),1,_)`
- `load()` function, such as `load(local!a:20, _)`
- `or()` function, such as `or(isleapyear(1996), _)`
- `with()` function, such as `with(a:10, _, a+b)`

For a function recipe that showcases a looping function that uses partial evaluation, see also: Boolean Results

Indirectly Evaluating Arguments

Rules can execute a function, rule, data type, or partial function passed as an argument indirectly by way of a rule input. For example, a rule that forma data for a Grid component can indirectly evaluate a rule input function based on the PagingInfo value passed to the rule each time a user interacts with the grid.

See also: Grid Tutorial

To do this, create a rule with an input of type Any Type and define the rule as `=ri!inputName()` where `inputName` is the name of your input based on what it will represent. For example `gridDataFn`.

You can then pass 0, 1, or more parameters to this rule depending on the number of parameters that the function, rule, or data type entered as the `inputName` expects.

For example, if you have a rule called `myrule` with the inputs `ri!inputName` (Any Type) and `ri!input` (Any Type), and it is defined by `=ri!inputName(ri!input)` the following expressions evaluate as shown:

`myrule( fn!sum, {1, 2, 3})` returns `6`

`myrule( fn!average, {1, 2, 3})` returns `2`

`myrule( fn!sum( _, 4), {1, 2, 3})` returns `10`

```
myrule(
  a!pagingInfo(
    startIndex: _, batchSize: 10),
   1
)
```

returns `[startIndex=1, batchSize=10, sort=]`

This functionality is especially useful when defining reusable rules that encapsulate the logic for a SAIL interface.

For example, an expression rule is set up to configure a grid component in a UI, and the rows in the grid represent vehicle data. The rule defines the gri columns, label, instructions, etc., and takes as its input a reference to the source of the data. This data will sometimes come from process variables or from an external database through query rules. The rule that returns the grid component with vehicle data will then be called as follows:

If the data comes from process variables:

```
=showVehicleGrid( todatasubset( pv!vehicles, _))
```

If the data comes from a query rule:

```
=showVehicleGrid( getVehiclesForClient( pv!clientId, _))
```

If the data comes from a rule that calls a query rule:

```
=showVehicleGrid( determineVehiclesToShow( _, pv!someInput1, pv!someInput2))
```

In the above examples, the rule `showVehicleGrid` takes a partial function that expects one parameter. showVehicleGrid then applies a PagingInfo value t the partial function to return one page of data in a data subset.

NOTE: Rules created for the Web Content Channel do not support indirect evaluation.

HTML Tags and Attributes

Certain tags and attributes are supported for expressions and render when the evaluated output is displayed.

For example:

`"Book Title: <b>" & pv!bookTitle & "</b>"` returns Book Title: **To Kill a Mockingbird**

If HTML tags are used elsewhere, the system prevents the browser from rendering the tags by escaping them.

- HTML tags are always escaped (removed) when an expression is used to define a report column.

For example, when entered in a text field:

`Book Title: <b>One Flew Over the Cuckoo's Nest</b>` returns `Book Title: &lt;b&gt; One Flew Over the Cuckoo's Nest &lt;/b&gt;`

If an unsupported attribute of a supported HTML tag is used, the attribute will be stripped out and dropped.

For example:

`<a href='#" onclick='doSomething()'>" & pv!bookTitle & "</a>` returns `[Lord of the Flies](http://)` (without the `onclick` attribute)

You can configure the system to log every time HTML is changed by Customizing your Application Logging for HTML Filtering.

Supported HTML Tags and Attributes

The following HTML tags and attributes are supported in expressions. Multiple supported attributes can be used in a supported tag.

`<b>` , `<strong>` , `<i>` , `<em>` , `<u>` , `<ol>` , `<ul>` , `<li>` , `<big>` , `<small>` , `<br>` , `<hr>` , `<thead>` , `<tbody>` , `<tfoot>` , `<tr>` , `<th>` , `<table>` , `<td>` , `<font>` , `<a>` , `<img>`

Supported `<table>` Attributes: `<table cellspacing>` , `<table cellpadding>` , `<table border>`

- Only integers are accepted cellspacing and cellpadding values.

Supported `<td>` Attributes: `<td width>` , `<td valign>`

Supported `<font>` Attributes: `<font color>` , `<font face>` , `<font size>`

- Only color names and color codes are accepted for the color attribute.
- Face values are tested against the following regular expression: `[\w;, \-]+`
- Size values are tested against the following regular expression: `(\+|-){0,1}(\d)+`

Supported `<a>` Attributes: `<a href>` , `<a name>` , `<a target>`

- Target values are tested against the following regular expression: `[a-zA-Z0-9-_]+`

Supported `<img>` Attributes: `<img src>` , `<img name>` , `<img alt>` , `<img height>` , `<img width>` , `<img align>` , `<img hspace>` , `<img vspace>` , `<img border>`

- Source (src) attributes are tested as to whether they're offsite or onsite URLs.

Configuring Additional HTML Tags and Attributes

Appian recommends you **do not** add or update the default list of allowed HTML tags and attributes for increased security against cross-site scripting attacks. However, depending on their needs of their environment and their security requirements, some Appian users list additional HTML tags and attributes in **safehtml.xml**.

This file is stored in `<APPIAN-HOME>/ear/suite.ear/lib/appian-security.jar` . It is possible to extract this file and whitelist additional HTML tags or attributes.

After modifying it, save the file under the `<APPIAN-HOME>/ear/suite.ear/web.war/WEB-INF/classes/resources/appian/security/antisamy` directory.

NOTE: Appian is not responsible for the impact of any changes made to the `safehtml.xml` that we provide you. Modifying this file may have serious implications with regard to the security of your Appian environment.

Permissions Used During Evaluation

In order for an expression to execute and logic to evaluate when the system attempts to derive information at runtime, the initiator must have sufficien user rights.

For example, if an expression is initiated by a user who does not have sufficient rights to access a resource requested by the expression, it will encounte an exception that halts the evaluation.

When creating an expression for a process model, the designer or task owner may require the rights instead.

See also: User Contexts for Expressions in Process

Testing Expressions

The validate button in the Expression Editor only checks for the following:

- Invalid Function, Rule, and Referenced Literal Objects
- Failed Casts
- Duplicate Keywords
- Invalid Parameters
- Incorrect Number of Required Arguments

To test the logic of your expressions, create it through the Rules Designer.

See also: Testing a Rule