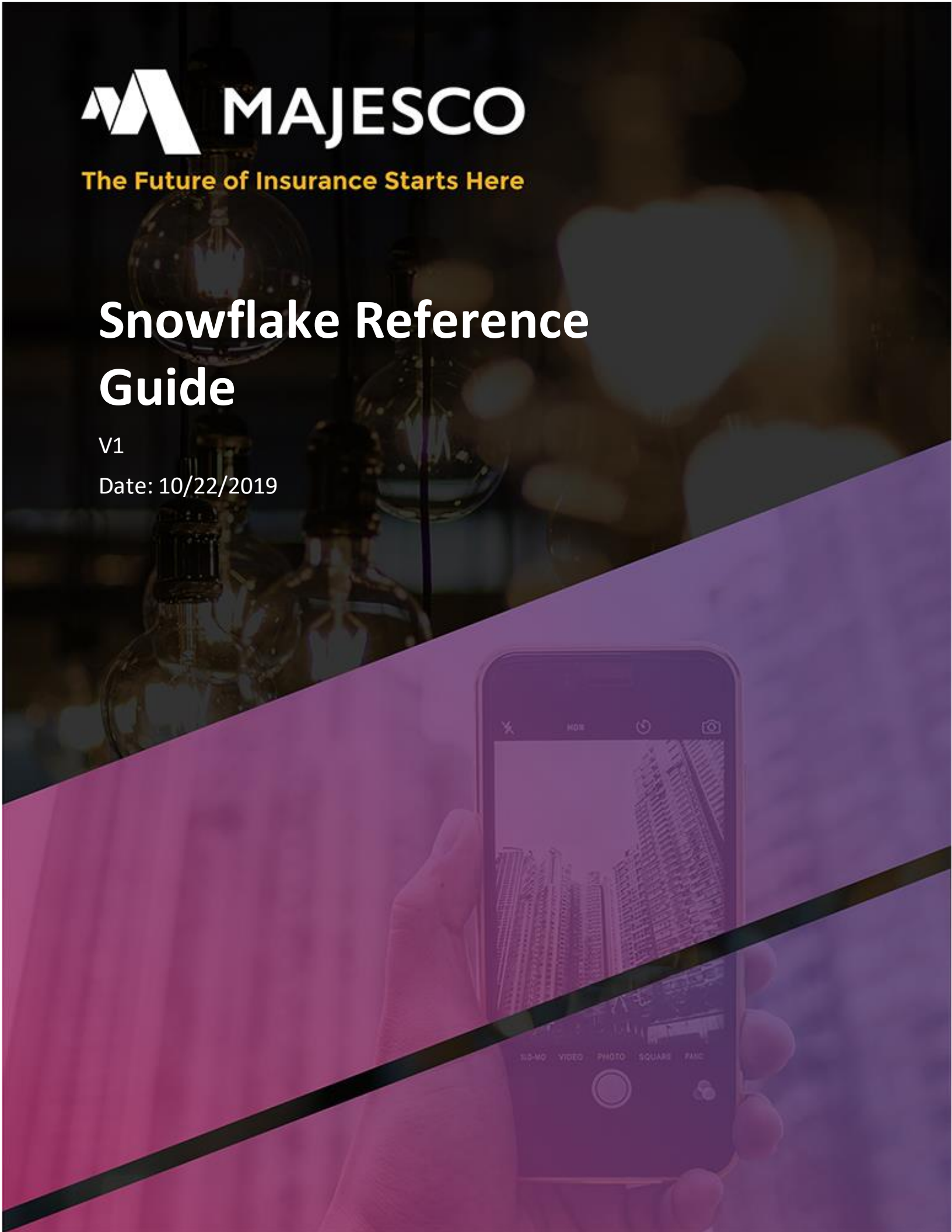
MAJESCO

The Future of Insurance Starts Here

Snowflake Reference Guide

V1

Date: 10/22/2019



Confidentiality Statement

This document contains proprietary and confidential information and data of Majesco Ltd. and its affiliates. This document is provided, on the express condition that such information and data will not be used, disclosed, or reproduced in any form, in whole or in part, for any purpose (other than solely for evaluation purposes by authorized representatives with a need to know), without the express written approval of Majesco. The right to use, disclose, and reproduce such information and data shall be governed by the service agreement between the client and Majesco.

Warnings and Disclaimer

The information provided is on an as is basis. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book. Screenshots and data used within are for illustration purpose and may vary from the actual application.

Contents

1 Introduction

Snowflake is an adaptive Domain Specific Language (DSL) designed to write business processing logic for a particular business domain such as Insurance. Snowflake adapts very easily to the new or evolving domain object model. This language has been designed with the following objectives in mind:

- It should be simple enough for a business analyst or domain expert to use with minimal training
- It should be aware of the domain or Object Models
- It should be dynamic and not require phases such as compilation, build process etc.
- It should be fast and run as fast as compiled general-purpose language such as Java
- It should be extensible to allow future extension
- It should be modular and any future extensions should not make existing rules invalid

Snowflake can be used to create rules or processing logic in following areas of the system

- LOV (List of values)
- Validation rules
- Error messages
- Object Model rules such as validations, error messages
- UI rules such as conditional display of field/sections
- Navigation rules for conditional routing through screens in a flow
- Pre/Post processing logic between steps in a flow

The Designer provides an intuitive editor where ever rules can be entered with productivity features such as syntax highlighting, auto completion.
This reference guide will provide a walkthrough of the language specification usages and examples.

2 Language Syntax

Snowflake is a case sensitive language. This applies to all the keywords, object model, event references as well.

2.1 Comments

is used to identify a single line comment
/* */ is used to identify a comment for a block of code that can span multiple rows

2.2 Arithmetic Operators

Operation	Symbol
Add	+
Subtract	-
Multiply	*
Divide	/
Mod	mod

2.3 Relational Operators

Relational operators allow the user to compare values for equality. For relational operators, the words and/or symbols can be used as mentioned in table below

Words	Symbol
equals to	=
not equals to	<>
greater than	>

greater than equals to	>=
less than	<
less than equals to	<=

2.4 Special Keywords

Special keywords are used along with relational operators for comparison

Keyword	Description
null	null keyword specifies empty value for a variable

Null keyword can be use in relational operation to compare whether variables/model variable value is empty.

```
if This.Coverage[Code="BI"] = null then
  def cov as new Policy:PersonalAuto.Vehicle.Coverage
  cov.Description = "Bodily Injury"
  cov.Code = "BI"
  This.Coverage = cov
end
```

2.5 Logical Operators

For logical operators must be expressed as word and not symbol.

Words
and
or

2.6 Variables

Local variables can be created that can be used within the business logic. Typically, values will be stored within the object model (for example Quote, Policy, Billing, Claim objects) but can use

local variables for performing business logic when the data does not need to be saved within the quote/policy, billing account, claim). It is advised that interim data such as factors be stored with the quote/policy to satisfy the needs of generating rating worksheets. Example

```
def x as 5
def x as 10.5
def x as “Hello”
def x as true
```

2.7 Assignment

Business logic in rules can assign value to local variable by referencing object data, or update the object field values by referencing field name as assignee. In below example, one can define a local variable and assign Vehicle Objects field value using assignment statement. Once data is updated to local variable, it can be used in various language constructs.

Examples

- Assign field value to local variable and reference the local variable in if-then-else construct

```
def x as Vehicle.Color
if x = "Red" then
   execute rule
end
```

- Assign data to field, this will result in updating the vehicle color field in database

```
Color = "Red"
```

2.8 Flow Control Statement

2.8.1 Conditional statement (if-then-else)

The if-then-else statement, including nesting, can be used to evaluate conditions and invoke a set of instructions based on the results of the condition. Example:

```
if x greater than 5 then
   B = 3
else
   B = 0
end

if z = 6 then
   A = 4
end
```

2.8.2 Looping

Looping statements allow the use to write expression that can iterate over collection of objects.

2.8.2.1 foreach...end

A foreach loop permits code to operate upon a collection of items. Typically, a parent object will iterate over a collection of child object instances.

Example: Using the General Liability example, code can be written to loop through each of the Classification records for a given Location using the following syntax:

```
foreach Classification as cls do
  Premium = Premium + cls.Premium
end
```

Note that, one must use "cls." when referencing variables for the object in which looping is being done. It is not necessary specify Location when referencing variables on the current Location that is being rated.

Please note that only the Classification records related to the current Location are referenced using foreach. If necessary, there is a Find method that allows the user to find one or more records of a particular object type within the entire policy based on criteria (across all Locations).

2.8.2.2 for...end Statement

A for statement permit code to iterate/execute the block for specified value. Syntax of for block is

```
for <variable> as <startcounter> to <endcounter> increment by <incrementcounter> do
  <statements>
end

# Increment by keyword is optional and default increment counter is 1.
# e.g. Below for block will iterate for 10 times

def a as 0

for x as 1 to 10 do
  a = a + 5
end

return a
```

2.9 Branching Statements

The branching statements allow one to define expression to exit from the execution blocks blocks created by flow control statements.

2.9.1 Return Statement

Return statement is used to end the execution of the rule and return the value from execution block to caller.

In below example, code will verify the Color field of the vehicle, if Color is “Red” then it will return value as "Supported" or else return value as "Not Supported" to the caller. In a scenario where user is writing the Snowflake expression in Rule When construct, user can use only "execute rule" expression.

```
if x = "Red" then
   return "Supported"
else
   return  "Not Supported"
end
```

In a scenario where user is writing the Snowflake expression in Rule When construct, user can use only "execute rule" expression.

```
def x as Vehicle.Color
if x = "Red" then
   execute rule
end
```

2.10 Break Statement

Break statement is used in for or foreach statement block to break the execution of repeating block.

Example

```
def a as 5, z

for x as 1 to a * 2 increment by 2 do
   z = x * 5

   if x >= 4 then
      break
   end
end
```

```
return z
# Result of above execution block will be 25
```

2.11 Continue Statement

Continue statement is used to skip the execution of current iteration and move to the next iteration of block.

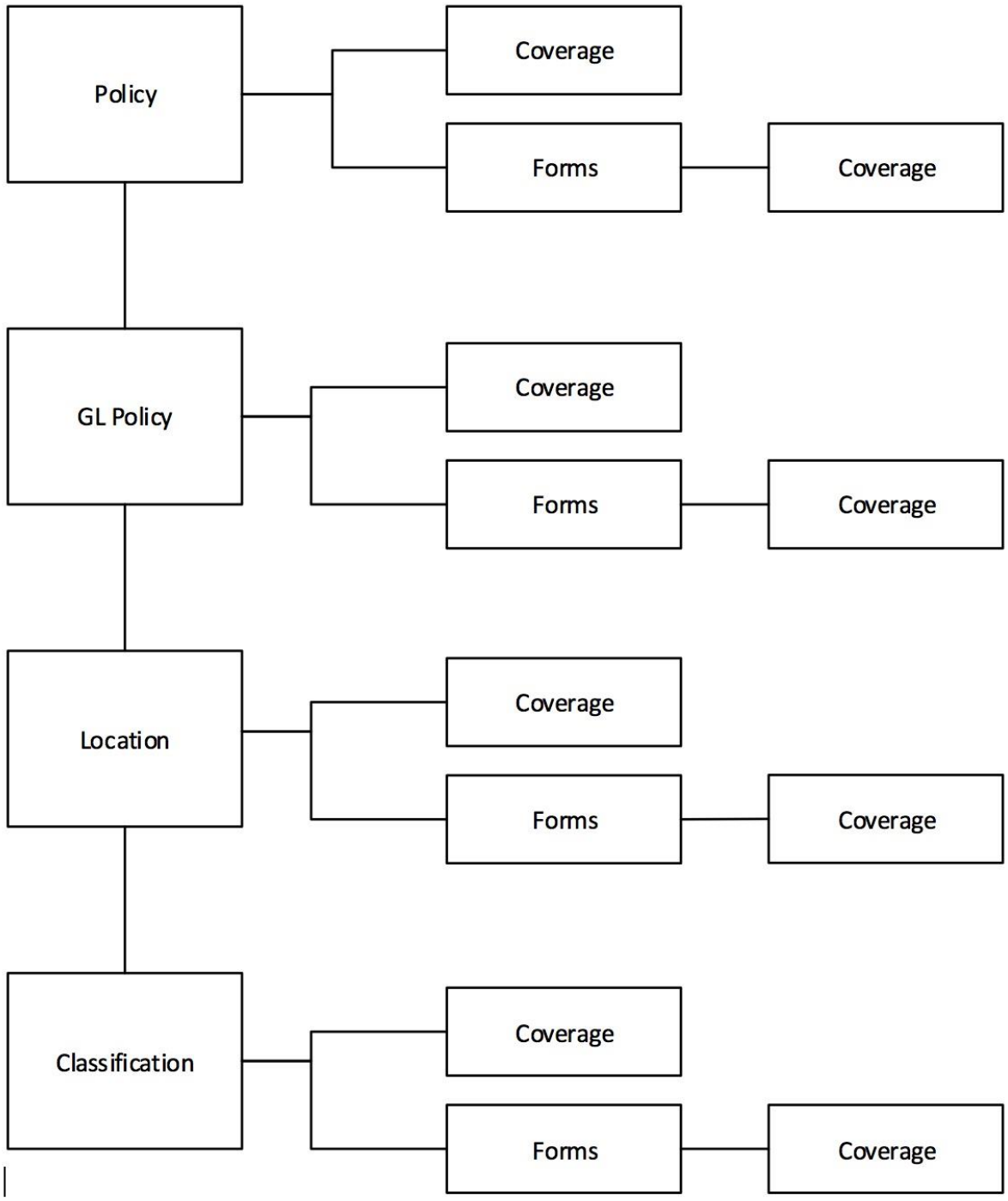
Example

```
def a as 5, y

for x as 1 to a do
   if x = 2 then
      continue
   end
   y = x * 5
end
return y

# Result of the above execution block will be 5
```

2.12 Model Aware Syntax

Snowflake adapts to a new or evolving domain object model. Its ability to reference the domain objects and its fields with easy to use syntax for basic CRUD operations on the objects makes Snowflake a very powerful domain specific language. The domain objects can be defined or extended through object designer and Snowflake adapts to these changes. For understanding these powerful capabilities and syntax structure, we will use the following example from an hypothetical policy object model:

2.12.1 Example Model



2.12.2 This and Parent

The user will be able to reference the model of an object being evaluated (such as policy, quote, claim, etc.). If the logic was invoked for a child object, users can always reference objects that are an ancestor, child or peer of the current object.

In the General Liability example, there will be a root Policy object with a child of General Liability. General Liability will have a child of Location. Location will have a child of Classification.

If there is business logic associated with the Location object, it can reference its own attributes by simply typing the attribute name. The user can optionally use the keyword `This` to represent the current object. If the user wants to access attributes of its parent, they can use the `Parent` keyword.
For example, if the user is working on an algorithm associated with the Location and wants to access information regarding the General Liability parent object, the `Parent` keyword can be used. The `Parent` keyword will allow the user to access all of the behaviours of the parent object, including access of its children, just as if you were within code for the object itself.
The user will be able to select from the long format of the object and attribute names so it will be more familiar to them, as opposed to the database table and column names. The long format of each will be captured in the metadata definition during the process of creating the object model.

Examples of referencing a policy object and its fields:

Object Reference	Description
Policy	Reference to the entire policy object
Policy.EffectiveDate	Reference to the EffectiveDate from Policy object
This.EffectiveDate	Reference the current object's EffectiveDate
EffectiveDate	Reference the current object's EffectiveDate. This is optional!
Global.EffectiveDate	Based on the current context, reference the nearest field from the current object model that has it's Global Name property set as EffectiveDate

Policy.PrimaryInsured	Reference to the PrimaryInsured object or field on the Policy object
Policy.PrimaryInsured.InsuredName	Reference to the PrimaryInsured.InsuredName field on the Policy.PolicyInsured object
Policy.AdditionalInsured	Reference to the AdditionalInsured object or field on the Policy object
Policy.AdditionalInsured [InsuredType=vInsuredType (and/or…)]	Reference to the AdditionalInsured object on the Policy object that uses filter criteria to return a list of AdditionalInsureds
Policy.AdditionalInsured [InsuredType=vInsuredType (and/or…)].Name	Reference to the AdditionalInsured.Name object on the Policy object that uses filter criteria to return a list of AdditionalInsured.Name
Parent.RiskState	If current context is Insured object and if you want to access the Risk state from parent, you can use Parent keyword to refer parent object field.

2.12.3 Data Access With Predicates/Criteria

Predicates in Object aware statement, allows to access the Object values only if Predicate conditions are satisfied.

Consider a scenario where there are multiple coverage's stored under vehicle Object. Now you want to access a Coverage object with Coverage Code as "BI", If Coverage with "BI" code is not available then we want to create the new Coverage Object instance and assign it to existing vehicle instance. Below statement will allow you to select the specific coverage object under vehicle object and if its not available then we can create a new object instance and assign/map it to existing vehicle instance.

```
if This.Coverage[Code="BI"] = null then
  def cov as new Policy:PersonalAuto.Vehicle.Coverage
  cov.Description = "BodilyInjury"
  cov.Code = "BI"
  This.Coverage = cov
end
```

2.13 Select Statement

Select statement is used in Report building to get the data from transaction tables. Syntax of select statement is as below

```
from {in memory} <modelName> select <fieldReference>, <fieldReference> where <fieldCondition>
```

keywords used in { } are optional keywords. In above expression "in memory" is an optional keyword and can be used to execute the select expression against in memory database. By default select expression will get executed against permanent database.

For example, to show a report of the list of Quotes in pending status, we can used the Select statement in report module as below

```
from Policy
 select QuoteNumber, EffectiveDate, ExpirationDate, Insured.MailingAddress.State
 where Status = "Pending"
```

2.14 Creating "new" Objects

The `new` object statement is used to create a new local object instance that can be later assigned to the existing object instance for saving data into the database.
Syntax of the new object instance is

```
def <variablename> as new <ModelName>:<RelationPath>
def <variablename> as new <ModelName>:Root # To reference fields from root of the object model
```

Example: Consider that we want to verify a location using a third party location verification service. Let us also assume that we have access to General Liability > Location and we would like to verify the current location instance by sending information to this third party service. The third party service expects the location information in specific data format, we can use below new statement to create the instance of third party model and set the location data from existing General Liability instance.

```
def tpLocation as new ThirdPartyLocation:Root
/*
This DSL statements will create a new local instance of
ThirdPartyLocation object model which can be later referenced using
local variable named "tpLocation"
*/


tpLocation.VerifyAddress = GLLocationAddress
# Refering the root instance of the model instantiated above

return tpLocation
```

2.15 Linking Child Objects With Using Statement

New Association statement is used to relate one object record with another object. E.g. In General Liability example using DSL statements, we can create a local General Liability object instance using def DSL statement. Once this instance is available, we can add this instance to existing policy instance using new association statement.

Syntax of new association statement is

```
new <fieldReference> using <variableReference>
```

Example, we can use below Snowflake block to create a general liability instance and associate it with existing policy instance using DSL statements. Consider that currently you are defining a integration block with reference to Policy instance.

```
def gl as new Policy:GeneralLiability

gl.State = "NY"

gl.Type = "Commercial"

new this.GeneralLiability using gl
```

2.16 Deleting Links

Delete association statement is used to delete an association between two object instancecs. For example, if you want to delete a Coverage from under General Liability instance, we can use the delete association statement to delete the existing coverage

Syntax of delete association is

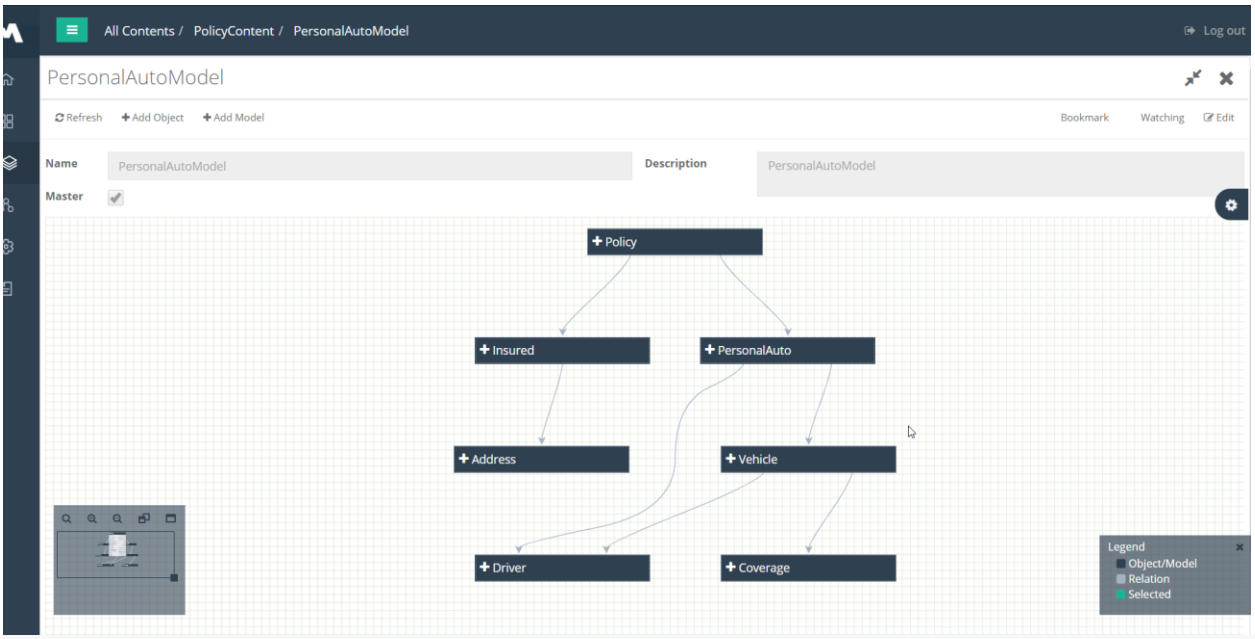
```
del <fieldReference>
```

Example, if you are having access to the GeneralLiability instance, you can select a specific coverage object instance and delete it using delete statement

```
del coverage[Code="PREMOPS"]
```

2.17 Sample Code With Multiple Snowflake Expression

Consider a Object Model with below hierarchy.




```
def x
x = lookup RefCoverage to return RefCoverage.Code as code , RefCoverage.Description as desc

if This.Vehicle = null then
   def veh as new PersonalAutoModel:PersonalAuto.Vehicle
   This.Vehicle = veh
end

foreach x as y do
                def code as y.code
   if This.Vehicle.Coverage[Code=code] = null then
      def cov as new PersonalAutoModel:PersonalAuto.Vehicle.Coverage
      cov.Description = y.desc
      cov.Code = y.code
      This.Vehicle.Coverage = cov
   end
end

foreach This.Vehicle.Coverage as cov do
   if cov.Code <> null then
      def limit
      limit = lookup RefLimit to return RefLimit.Value as value having RefLimit.CoverageCode = cov.Code
      cov.Limit = limit.value
                              cov.Premium = limit.value / 100
   end
end

def totalPremium as 0

foreach This.Vehicle.*.Premium as prem do
   if prem <> null then
      totalPremium = totalPremium + prem
```

```
  end
end

This.Premium = totalPremium
```

Above block is written on a route navigation while going from Personal Auto information page to vehicle page. User want to achieve below operation using above code block.

1. If Vehicle instance is not available, create a new vehicle instance and assign it to current Personal Auto object
2. Get the list of Coverage's available in Reference table [named RefCoverage]
3. Create the coverage instance under Vehicle object using number of coverage's available under RefCoverage reference table.
4. Get default limit value from Reference Table [named RefLimit] and Assign it to the vehicle coverage instance
5. While saving the limit value, calculate the premium using arithmetic operation and save it to coverage's premium field.
6. Find out the total coverage premium by looping over all the coverages available under vehicle and save it at Personal Auto's premium field.

3 Functions

Functions can take zero or more variables or functions as input to execute a logic and return the result that can be assigned to another variable or used in any of the statements where a variable can be used.

Functions are great way to eliminate the duplication of common logic. This improves the productivity and make the code more readable and maintainable.

General syntax for using functions is

```
variable = Function Name (variable1, variable2, Another Function(...), ...)
```

Following functions are available in Snowflake out of the box

3.1 String Functions

Function	Description
trim left	Return a string containing a copy of a specified String with no leading spaces
trim right	Return a string containing a copy of a specified String with no trailing spaces
trim	Return a string containing a copy of a specified String with no leading and trailing spaces
upper	Return a string or character containing the specified string converted to uppercase
lower	Return a string or character containing the specified string converted to lowercase
title	Return a string or character containing the specified string converted to title case
left pad	Return a string containing a copy of a specified String padded from left by specified pad string
right pad	Return a string containing a copy of a specified String padded from right by specified pad string
length	Return an integer that contains the number of characters in the String

replace	Replace single instance of old string with new string
replace all	Replace all instances of old string with new string
replace first	Replace first instance of old string with new string
replace last	Replace last instance of old string with new string
left	Return a string containing a specified number of characters from left side of specified string
mid	Return a string containing a specified number of characters from specified string
right	Return a string containing a specified number of characters from right side of specified string
compare	Returns -1, 0 , 1 based on the result of a string comparison. Return 0 if both string are equals, -1 if string 1 sorts ahead of string 2, 1 if string 2 sorts ahead of string 1.
contains	Return true if string contains the specified sequence of character value.
not contains	Return true if string does not contains the specified sequence of character value
concat	Join two or more strings

1. trim left

```
Snowflake Expression: return trim left(" Hello ")
Output: “Hello “
```

2. trim right

```
Snowflake Expression: return trim right(" Hello ")
Output: “ Hello”
```

3. trim

```
Snowflake Expression: return trim (" Hello ")
Output: “Hello”
```

4. upper

Snowflake Expression: return upper("Hello")
Output: “HELLO”

5. lower

Snowflake Expression: return lower ("HELLO")
Output: “hello”

6. title

Snowflake Expression: return title("this is a Test for Sentence.lets see how it works.")
Output: “This Is A Test For Sentence.Lets See How It Works.”

7. left pad

Snowflake Expression: return left pad("Sample",5,"000")
Output: “Sample”

8. right pad

Snowflake Expression: return right pad("Sample",15,"000")
Output: “Sample000000000”

9. length

Snowflake Expression: return length ("Hello")
Output: “5

10. replace

Snowflake Expression: return replace ("SampleTextSampleTeSamplext","Text","")
Output: “SampleSampleTeSamplext”

11. replace all

Snowflake Expression: return replace all ("SampleTextSampleTeSamplext","\[a-z\]","-")
Output: “S-----T---S-----T-S-------”

12. replace first

Snowflake Expression: return replace first("SampleTextSampleTextSampleText"," Text","")
Output: “SampleSampleTextSampleText”

13. replace last

DSL: return replace last("SampleTextSampleTextSampleText"," Text","")
Output: “SampleTextSampleTextSample”

14. left

Snowflake Expression: return left("HelloWorld", 6)
Output: "HelloW"

15. mid

Snowflake Expression: return mid("HelloWorld", 3,3)
Output: “loW“

16. right

```
Snowflake Expression: return right("HelloWorld", 3)
Output: “rld“
```

17. compare

```
Snowflake Expression: return compare("Hello","hello")
Output: “-1“
```

18. compare

```
Snowflake Expression: return compare("Hello","Hello")
Output: “0“
```

19. contains

```
Snowflake Expression:
if contains ( "shashikant",  "kant" ) then
 return true
end
Output: true
```

20. not contains

```
Snowflake Expression:
if not contains ( "shashikant",  "mhatre" ) then
 return true
end
Output: true
```

21. concat

```
Snowflake Expression:
def insName as Insured.Name
return concate ( "Hello !",  insName )
Output: Hello ! Shashikant
```

3.2 Number Functions

Function	Description
ceiling	Return the nearest number value greater than or equals to specified number value
floor	Return the nearest number value less than or equals to specified number value
absolute	Return the absolute value of number
round	Return the value of number rounded to a specific number of digit

1. ceiling

```
Snowflake Expression: return ceiling ( 100.25 )
Output: 101
```

2. floor

```
Snowflake Expression: return floor ( 100.25 )
Output: 100
```

3. absolute

```
Snowflake Expression: return absolute ( 100.25 )
Output: 100.25
```

4. round

```
Snowflake Expression: return round ( 10.2 * 20.4 )
Output: 208
```

3.3 Date Functions

Function	Description
current date	Return current system date
date	Return a date instance for a specified date string, provide date string is specified as "date-month-year" format
date with format	Return a date instance for a specified date string and provided date format string
date difference	Return a number value for number of days, months or year between two days.
change date	Return a new date instance by adding number of days, months or year in specified date.

1. current date

```
DSL: return current date

Output: “2018-03-29“
```

2. date

```
DSL: return date ("03/28/2018", "dd/MM/yyyy")

Output: “2018-03-29“
```

3. date as

```
DSL: return date ("29-Mar-2018")

Output: “29-Mar-2018“
```

4. date difference

```
DSL: return date difference (date("02-May-2017") , date("02-May-2018"), days)

Output: “365“

DSL: return date difference (date("02-May-2017") , date("02-May-2018"), years)

Output: “1“

DSL: return date difference (date("02-May-2017") , date("02-May-2018"), months)

Output: “12“

DSL: return date difference (date("02-May-2017") , date("02-May-2018"), weeks)

Output: “52“

DSL: return date difference (date("02-May-2018") , date("02-May-2018"), years)

Output: “0“
```

5. change date years

```
DSL: return change date years (date("02-Mar-2018"), 1)

Output: “02-Mar-2019“
```

6. change date months

```
DSL: return change date months (date("02-Mar-2018"), 3)

Output: “02-Jun-2018“
```

7. change date days

```
DSL: return change date days (date("02-May-2018"), 30)

Output: “01-Jun-2018“
```

4 Language Extensions

Language extensions are specialized functions that depend upon other configurations and adds significant capability to overall system.

4.1 Table Lookup

Performing reference table lookups is a common function when implementing premium calculation logic and validating field values. This involves providing inputs for a given table and having the system return a value. The language will provide an easy method of performing table lookups. When the user enters the Lookup method, they will be prompted for the name of the table, and once entered/selected, they will be prompted for each of the inputs. The results of the lookup will be returned into a variable.

```
lookup <TableName> to return <TableName.KeyValue1> {as <alias1>}, <TableName.KeyValue2> {as <alias2>} ...
having <TableName.KeyValue3 = Value>
```

Example: Consider following reference table defined in the Designer

Key	Factor
0	1.84
1	5.51
4	11.40
9	29.41

Assume that the value of the Key is 3. In that case we will get following results.

Type	Description	Outcome for Key=3
Exact Match =	The rate table key field value must exactly match the value being passed into the rate table from the rate request	null

Less Than <	The system compares if the value being passed is less than the lowest value in the table and then works its way up to compare against the next highest value in the table	11.4
Less Than or Equal To <=	The system compares if the value being passed is less than or equal to the lowest value in the table and then works its way up to compare against the next highest value in the table	11.4
Greater Than >	The system compares if the value being passed is greater than the highest value in the table and then works its way down to compare again the next lowest value in the table	5.51
Greater Than or Equal To >=	The system compares if the value being passed is greater than or equal to the highest value in the table and then works its way down to compare again the next lowest value in the table	5.51

4.2 Return Multiple Columns

Following example shows how to return multiple values in a lookup statement

```
lookup Gender to return Gender.Code, Gender.Description
# Here Gender is a reference table with Code and Description columns
```

4.3 Joining Multiple Reference Tables

Following example shows joining the city and state reference tables to return the results from the join.

```
  lookup RefCity, RefState
to return RefCity.Code, RefCity.Description
  having RefCity.State = RefState.State
    and RefCity.Country = COUNTRY
    and RefState.State = STATE
```

4.4 User Profile Lookup

Sometimes we need to access the current logged in user details (e.g. Role, User Name, Last Login Time) to perform certain DSL operation. This can be achieved using special DSL syntax.

Accessing Current logged in user details using expression -

- Access the current logged in User's roles.
  `security.CURRENT_ROLE`
- Access the current logged in User's Name
  `security.USER_NAME`
- Access the current logged in User's last login time
  `security.USER_LAST_LOGIN`