Overview of EL

a Reference Domain Specific Language (DSL) Compiler

DTRules DSL Compiler

Contents

Introduction

The Formal Language Compiler is implemented using Flex and Cup. Flex is a Java implementation of Lex, and Cup is a Java implementation of YACC. Both projects are open source, and both are rather long lived and stable projects.

Nothing about DTRules is dependent directly on the compiler. Any compiler that can generate the Postfix versions for each condition and each action in a set of decision tables can be used with DTRules. Once the decision table XML has been "compiled" i.e. has the postfix versions of the conditions and actions generated and added to the XML, that XML can be used directly by the DTRules interpreter.

This means that no component of the compiler is required to deploy an application which uses a particular set of decision tables. The compiler would only be required to create changes to a set of decision tables.

This compiler can compile either a CONDITION (a statement that results in a boolean) or an ACTION (consisting of one or more statements).

Modifications

There are a number of modifications to the decision tables which are desired to make the flow of the decision tables friendlier to both developers and to business analysts. These include:

- An Optional Context section
- An Optional Initial Actions section
- Execute First
- Execute All
- An Optional Policy Statements section

The Context section describes how the table is to be applied to a list of Entities
The Initial Actions section describes the actions needed to be performed, within the context defined by the Context section, prior to evaluating the conditions and actions.
Unbalanced Decision Tables allow for a more straight forward expression of the business logic without tangling different actions and conditions together. Execute First allows the business logic to define a prioritized list of conditions, and the actions that should be performed in each case.
Execute All allows the business logic to define specific sets of conditions which should result in a particular action to be taken. These are evaluated in parallel, and all the specified actions are taken in the order described.
The Policy Statements section provides for the documentation of business logic on a column by column basis. With FIRST and ALL unbalanced decision tables, the columns become significant and mostly independent implementations of specific policy directives. The Policy Statement number refers to a column in the Action and Condition tables.

Context

Adding a context to the decision tables would allow a decision table to be evaluated more clearly, and would remove very "programmer" kinds of syntax from actions. For example, in a control table we often have something like:

NAME: Determine Individual Rates

COMMENTS: This is the control table for evaluating the Individual Rates for each case within a job

TYPE: balanced

CONTEXTS:

for all cases in the job

	CONDITIONS:	**CONDITIONS**	1	2	3	4
1	Evaluate all the cases in the job	perform when called	*			
	ACTIONS:	**ACTIONS**	1	2	3	4
1	Evaluate the risk by the zipcode of the individual	perform Evaluate_zipcode	X			
2	Evaluate the risk by the health of the individual	perform Evaluate_Health_Risks	X			
3	Evaluate the risk by the life style of the individual	perform Evaluate_Life_Style	X			
4	Compute the rate for the individual	perform Compute_Individual_Rate	X			

And the Evaluate Zipcode table would look like:

NAME: Evaluate_Zipcode					
COMMENTS:					
Table Type: balanced					
CONTEXTS:					
for all individuals					
for all zipcode_info whose zipcode_info.value == individual.zipcode					
CONDITIONS:		**CONDITIONS**	1	2	3
1	Update the individual's zipcode	zipcode_info.covered == true	y	n	
ACTIONS:		**ACTIONS**	1	2	3
1	Associate the zipcode info found to this individual	individual.zipcode_info = zipcode_info.zipcode_info	X		
2	Set the zipcode info to null, as the individual does not live in a covered zipcode	Set the individual.zipcode_info to null		X	

Initial Actions

The addition of an optional section for Initial Actions. Initial Actions are always performed prior to evaluating the conditions, and within the context of the table. The Initial Actions can be used to do further evaluations which will be tested by the conditions in the table. Or the Initial Actions may add entities to the context prior to the evaluation of the conditions and actions in a table. Any entity added to the context within the initial actions of a table will remain in the context while evaluating the conditions and actions of the table. Then an entities added to the context are removed.

For example:

NAME: Evaluate the Income Levels for each EDG_Individual					
COMMENTS: For each Each Eligibility Determination Group (EDG), the individuals in that edge (represented by the EDG_Individual) is considered and their income added to the group.					
Table Type: balanced					
CONTEXTS:					
for each Edge_Group					
for all EDG_Individuals					
INITIAL_ACTIONS:					
1	Add the Individual represented by the EDG_Individual to the context of this table	Add the EDG_Individual.individual to the context of this table			
CONDITIONS:		**CONDITIONS**	1	2	3
1	Is this individuals income excluded?	The EDG_Individual.income_excluded == true	y	n	
2					
ACTIONS:		**ACTIONS**	1	2	3
1	Add the individual's income to the Edge Group's income	Add the individual.income to the edge_group.income		X	
2					
3					

As you can see, the context is rotated through each Edge_Group, and then for each Edg_Group the context is rotated through each EDG_Individual. Then the initial Action section places the individual associated with the EDG_Individual into the context for the table. Once the table is evaluated, the context is reset, and the whole process repeats.

Execute all that Match

Unbalanced Decision Tables allow logic to be expressed by describing the precise logic paths of interest in policy. But if a Decision Table doesn't explicitly define the relationships of all conditions to all execution paths, then some rule must be in force to define those execution paths. The first Rule is to execute all Columns whose conditions are matched. In this case, logically, all conditions are evaluated, and all actions are executed in those columns.

Even so, the actions are executed from the first action specified to the last action specified, and each action is execute once and only once, even if its execution is indicated by two or more columns to be executed.

The implementation of these decision tables actually build a balanced decision table at the time the decision table is loaded into the rules engine.

NAME: Evaluate Life Style										
COMMENTS: Add up the risks associated with each Life Style issue										
Table Type: all										
CONTEXTS:										
for all individuals										
CONDITIONS:		**CONDITIONS**	1	2	3	4	5	6	7	8
1	Individual Smokes	Smokes	y							
2	Rapid Heart Rate	resting_heart_rate > 80		y						
3	High blood pressure	systolic > 140 or diastolic > 90			y					
4	diagnosed with diabeates	Diabeates				y				
5	diagnosed with cancer	Cancer					y			
6	diagnosed with heart disease	heart disease						y		
7	Executed if nothing matches	Otherwise							*	
ACTIONS:		**ACTIONS**	1	2	3	4	5	6	7	8
1	Add the risk factor for smoking	add smoking_risk to total_risk_points	X							
2	Add the risk factor for high heart rate	add high_heart_rate_risk to total_risk_points		X						
3	add the risk factor for high blood pressure	add high_blood_presure_risks to total_risk_points			X					
4	add the risk factor for diabeates	add diabeates_risks to total_risk_points				X				
5	add the risk factor for cancer	add cancer_risks to total_risk_points					X			
6	add the risk factor for heart disease	add heart_disease to total_risk+points						X		

Execute the first that matches

The other way to balance a set of decision tables is to execute the first column that matches. Even if subsequent columns also match, once a column has matched that column is executed. In the case of a First table, the default condition is executed if and only no column matches.

NAME: Compute Individual Rate					
COMMENTS:					
Table Type: first					
CONTEXTS:					
CONTEXT: for all individuals					
CONTEXT: using individual.context_info					
CONDITIONS:		**CONDITIONS**	1	2	3
1	We can only deny people if the zipcode allows people to be denyed.	if denyable and total_riskpoints > max_risk_points	y		
2	Otherwise calculate a rate	Default		*	
ACTIONS:		**ACTIONS**	1	2	3
1	Mark the Individual as denied	set denied = true	X		
2	Set the rate at the base rate	rate = base_policy_rate		X	
3	Add the risk dollars to base rate	add risk_points * risk_to_rate_ratio to rate		X	
4					
5					
6					
7					
8					
9					

Data Mapping

Data mapping uses two objects, a Mapping Object which maps XML data into the EDD for the Rules Engine, and a DataMapping Object that maps programmatic data into the EDD. Using the DataMapping Object also allows the writing of an XML stream that in turn can be used to load the EDD through the Mapping Object later.

In addition to the explicit XML definitions for mapping data into an EDD, we should also use a Java interface to map Java objects to define the EDD for a Rule Set. We can then use an ObjectMapping Object to map such Java objects into the EDD.

Once this is done, an interface can be defined that provides a set of Java Objects which are automatically mapped into the EDD, and have a decision table execute, then have particular attributes mapped back into the same set of Java Objects. The only issue with this is defining how to restrict what attributes should be mapped back out to these Java Objects.

A Summary of the Statements Implemented in the Compiler:

The Domain Specific Language (DSL) is case insensitive. Being case insensitive allows the use of capitalization rules which are more in line with common English usage. Unlike other implementations of Decision Table based rules engines (such as the one used in Texas on the TIERS project) case insensitivity is rather uniformly implemented. Usage of case in the Entity Description Dictionary and the Decision Tables is ignored.

Articles in English are generally necessary to have the properly flow and feel of the language, but do not in and of themselves carry much meaning. So the articles (an, a, the) can be used anywhere in the language. They are ignored. The following are all valid and equivalent:

```
set the income_limit = the minimum_income_amount
set income_limit     =     minimum_income_amount
```

Entities are a rather unique structure in DTRules. They are collections of specific typed attributes. Entities also have a name. Thus an "Individual Entity" is an entity with the name "Individual."

Each attribute of an "Individual Entity" defines a slot with its own type and attribute name. Thus the attributes of an Individual such as "age", "birth_date", "citizen" etc. are typed attributes (age might be an integer, birth_date a time value, and citizen a Boolean).

A set of Decision Tables used to define policy assume some set of Entities. What Entities are defined, and its attributes are defined in the Entity Description Dictionary, or EDD.

At times, two entities on the entity stack may have an attribute of the same name. For example, an individual might have an attribute like "total_income". At the same time, there may be an entity such as "fiscal_group" which also has an attribute like "total_income".

A "fiscal_group" might represent a group of individuals, and its attribute "total_income" would give the total income for everyone in the fiscal group.

If one needs to write actions which combine the incomes of individuals to yield a total income for a fiscal group, then some way of distinguishing the difference in these two attributes is necessary. In this case, the "dot syntax" can be used to specify not only the attribute name, but the entity type as well. For example, if the entity stack contained an individual entity and an fiscal_group entity, we could still access an attribute defined by both using the dot syntax.

```
add the individual.total_income to the fiscal_group.total_income
```

Note that when an attribute's value is being determined, its value is defined by the context in force at that time. The context is defined by a structure within the Rules Engine referred to as the entity stack. These entities define the context for the execution of decision tables.

When we need the value of an attribute, a search is made of the entity stack. The value returned is the value defined by the top most entity that defines a value for that attribute. When an attribute uses the dot syntax, then not only the attribute must match, but the entity specifier must match as well.

When a decision table is executed "for all individuals", then what happens is that each element of the array individuals is placed in turn on the entity stack, and that decision table is executed. As

it turns out, policy is driven by lists. The list of cases to process, lists of individuals in these cases, lists of income sources for an individual, etc.

The EDD defines the attributes for an Entity. But in addition, every Entity gets an attribute generated for it which is self referential. So if you create an entity **Individual**, it will have an attribute **Individual** generated of type entity that refers to that **Individual** entity.

Attributes, Decision Tables, and operators are all defined by Names. A Name in DTRules is implemented by the RName class. (As a convention, all the classes that implement key object types in DTRules begin with an R. Part of this is to make the Rules Engine implementation of objects easy to recognized. Another reason is that the Objects implemented (Integers, Entities, Booleans, etc.) are all very common name types in programming anyway. Thus leading each name with an 'R' reduces the name collisions.

A Condition must be a single Boolean expression. An Action can be a list of one or more statements separated (but not terminated) by a semicolon. The fact that the last statement cannot have a trailing semicolon is a side effect of avoiding all ambiguities in the language.

The last statement in an action list cannot have a trailing semicolon

Blocks are lists of statements contained in curly brackets. A block is also a statement. Generally speaking, we avoid the use of blocks of statements.

By convention a list of entities is kept in an attribute with an added "s". So for example, a case would have an attribute "Individuals" to hold a list of "Individual" entities.

Perform

The **perform** statement initiates the execution of a decision table. So for example, if we wished to execute a decision table entitled "Initialize", then one would say:

```
perform initialize
```

Decision tables are also directly executable. Thus the following does the same thing as the previous example:

```
initialize
```

For all

The Context block in a decision table is helpful in defining how a decision table should be executed. In general, a decision table is applied over a list of Entities, and perhaps done so as to limit its application to those entities that meet a particular criteria.

For example, a case may have a list of individual entities which each represent the individuals in a case. In this case, a Decision table might specify its execution over the individuals in the case:

NAME: Compute Individual Rate					
COMMENTS:					
Table Type: first					
CONTEXTS:					
for all individuals					
CONDITIONS:		**CONDITIONS**	1	2	3
1	We can only deny people if the zipcode allows people to be denyed.	if denyable and total_riskpoints > max_risk_points	y		
2	Otherwise calculate a rate	Default		*	
3					
4					

You can also place a condition on this iteration

NAME: Compute Individual Rate					
COMMENTS:					
Table Type: first					
CONTEXTS:					
for all individuals whose adult_flg == true					
CONDITIONS:		**CONDITIONS**	1	2	3
1	We can only deny people if the zipcode allows people to be denyed.	if denyable and total_riskpoints > max_risk_points	y		
2	Otherwise calculate a rate	Default		*	
3					
4					

Another option for wording iterations over an array of entities is to use the **for each** syntax

for each individual in the adult_list

Another option is to add to the context an Entity referred to by an Entity in the list. For example, each individual might reference an address. The individual and address can be added to the context:

for each individual and its address in the adult_list

Note that this syntax is another way of adding an entity to the context similar to using the "Add <entity> to the context of this table" syntax described earlier.

For the First

These phrases can be used in the Context section to pick out a particular entity for which some processing should be performed. They can also be used in statements.

This statement searches a list of entities for a match, then executes one or more statements. The "**and its**" phrase can also be used here.

In the context form:

CONTEXTS:
For the first of the individuals where hoh == true

And as used in an Action:

```
for the first of the individuals where hoh == true then {
        perform individual_test;
        perform case_test
}
```

In the context form:

CONTEXTS:
for the first of the individuals and its address where the coveredzipcodes includes address.zipcode

```
for the first of the individuals
      and its address where coveredzipcodes includes address.zipcode
  then {
        perform address_test;
        perform individual_test;
}
```

In the statement form, we have the option of doing one thing if an entity is found that meets the given condition, or another thing if no entity is found that meets that condition. This cannot be done within the context statement for a decision table at this time. If the "for the first" type statement is used in the context of a decision table, and no entity is found, then no other blocks are executed (no initial actions, the condition block or the action block).

I have considered adding some syntax to insure that the table could be executed once whether or not the entity is found, and some syntax to test this condition in the condition block of the table. However, this has not be implemented at the time of this writing.

```
For the first of the individuals whose age < 18 then {
        set case.has_children = true;
} else if none are found {
        set case.has_children = false;
}
```

Blocks

blocks are lists of statements enclosed with {} (curly braces). Any where in this document you see curly braces, you can replace will one of the following options:

Using statement
For all or For each statement
Find the First statement
For the first statement
if statement.

Generally speaking, the use of blocks in Actions should be avoided. It leads to Actions which are very complicated and confusing.

Statements

Statements are used to define actions either in the Initial Actions Section, or the Actions block. Several statements can be placed within a single action. This should be avoided, especially when doing so requires the same statement to appear in several actions. When this happens, then many times a element of the Policy is being entangled in the representation of the table.

The Domain Specific Language supports the following types of statements:

set statement	- Set an attribute
perform statement	- Perform a decision table
debug statement	- Print a debug statement (if trace is on)
if statement	- Test a value (should be avoided)
add to statement	- Add a value to an integer, double, or array
clear statement	- Clear the elements of an array
using statement	- Perform a statement within the context of an Entity (should be avoided)
common error statement	- Throw an error
ignore	- A null statement (does nothing)
operator statement	- A direct call to an operator implemented in the Rules Engine (Could be a custom extension to the regular set of operators)
context statement	- Puts an Entity onto the Entity Stack for the duration of the execution of the decision table.
null statement	- Extra Semicolons are ignored.

Set

Set statements are used to set attributes to particular values. The reason we use the **Set** key word is that such expressions are a bit more friendly to non-programmers. The general form is:

Set <attribute name> = <value>

Examples would include:

Set age = 1

Set percentage = 5 / 100

Set flag = true

Set note = “This is a test”

Set startdate = currentdate

Set current_individual = individual

Set adults[0] = individual

The left handed expression must be an Integer Attribute, a Double Attribute, a Boolean Attribute, an Entity Attribute, a String Attribute, a Time attribute, a Table attribute, or an indexed Array reference. A conversion of the right handed expression is done to match the destination type.

Note that arrays allow indexed access. Arrays themselves are not typed. This is a limitation of the Compiler and the Rules Engine that will be addressed at some point. For now the developer or policy expert must insure arrays are handled consistently.

Debug statement

The debug statement can print a string expression, or any other basic type in DSL

```
debug “Got here”
debug  adult_flg
debug Age * income * year_percentage
debug count
debug individual
debug start_date
debug eligible_list
```

In the case of complex types (arrays, entities), the stringValue for the type is printed. This isn’t necessarily the prettiest output you might like.

If Then Else

Your basic If statement. ***This should be avoided***, since most conditions ought to be managed in the decision tables themselves. However, there are times that a simple If Statement implements the logic easier than other options.

If adult then perform adult_test

If child then perform child_test else perform adult_test

Add to Statement

Add to statement makes addition much more “readable”, and also serves to add elements to arrays.

add 1 to the count_of_adults

add individual to the adults

add individual to the adults and the eligible_individuals

In this later case the one individual is added to two different lists. Sometimes you only want to add the individual if it isn't already a member. This prevents duplicate references to the same entity in the list. This syntax is:

add individual if not a member to the eligibles

add individual if not a member to the adults and the eligibles

Context Statement

If you need an entity to be part of the context, then the Context Statement can be used. The use of this statement can eliminate most Using Statements.

Add the individual to the context of this table.
Add the individual to the context for this table.

The entity added to the context remains there until the Decision Table completes execution of the Condition and Action block. Thus for a table:

NAME: Evaluate the Income Levels for each EDG_Individual					
COMMENTS: For each Each Eligibility Determination Group (EDG), the individuals in that edge (represented by the EDG_Individual) is considered and their income added to the group.					
Table Type: balanced					
CONTEXTS:					
for each Edge_Group					
for all EDG_Individuals					
INITIAL_ACTIONS:					
1	Add the Individual represented by the EDG_Individual to the context of this table	Add the EDG_Individual.individual to the context of this table			
CONDITIONS:		**CONDITIONS**	1	2	3
1	Is this individuals income excluded?	The EDG_Individual.income_excluded == true	y	n	
2					
ACTIONS:		**ACTIONS**	1	2	3
1	Add the individual's income to the Edge Group's income	Add the individual.income to the edge_group.income		X	
2					
3					

The EDG_Individual.individual added to the context in the Initial Actions will be removed for each iteration over the Edg_Groups and the EDG_Individuals.

Subtract Statement

Only for numeric attributes, you can use this statement to remove a value

subtract 1 from the count

subtract .5 from the percentage

Remove Statement

Allows elements to be removed from arrays

Remove the 10 element from the adults array

Remove individual from the adults array

The first example removes the 10 th element from the adults array, while the second removes the reference to the particular individual entity from the adults array.

Randomize

Randomize just mixes up the order of the elements in an array

Randomize adults

Clear

The clear statement clears all the elements from an array, leaving it empty

clear adults

Sort

The sort statement sorts an array of Entities by some given attribute. Because we use a sort that preserves the order of the entities should the attribute match, then you can sort by several attributes. The result will be ordered first by the last attribute sorted, then within matches by the second to the last, and then the third and so forth. So to sort a list of addresses by state, then by county, then by zip, you would say:

```
sort addresss in ascending order by $zip_code;
sort addresss in ascending order by $county;
sort addresss in ascending order by $state
```

We could also sort individuals in descending order by some score:

```
sort individuals in descending order by $score
```

Note that the $ sign is used to denote a name object. The use of the name object limits the error checking we can do, but allows for a much more general syntax. Take special care when using the sort statement to insure that the attribute you are sorting by matches the attribute in the entity you are sorting.

Operator Statements

Operator Statements allow Operators defined outside of the Rules Engine to be called from decision tables. This allows Decision Tables to easily access functions supplied by the application. The Syntax for Operator Statements looks much like a procedure or function call in other languages:

Set individualPlan = The String Value of ManagedCareProviders(Zipcode, CountyCode);

Set DiscountPercentage = The Double Value of the DiscountCalculation(Individual);

Set the StockIndex = The Long Value of the HistoricStockIndex(PlanStartDate);

Set the EligibleOnAllDates = the Boolean Value of the EligibleDateChecks(planDates);

Array Expressions

Anywhere you might have an array, you might use any of the following Array Expressions.

individuals – an attribute that refers to an array
get a copy of the individuals – Makes a copy of an array and returns it.
copy of the individuals - Also makes a copy of an array
new entity array – Makes an array of entities
new string array – Makes an array of strings
[10, 20, 30] – Array literal

Entity Expressions

Anywhere you might need an entity in the DSL syntax, you can use one of these expressions:

individual – an attribute that refers to an entity
individuals[i] – an indexed expression into an array of entities
(individual) – parenthesis around any entity expression
new individual – creates a new instance of an individual entity
clone of individual – creates a clone of a particular entity
:other_individual:individual – uses a reference to push on the stack
prior to evaluating the individual attribute.

Date Expressions

Policy often involves the manipulation of dates. Tests between dates will be covered in the discussion about generating Boolean expressions.

A String can be converted to date by something that looks like a cast expression:

(Date) “10/3/2007”

A long time value can also be cast in such a fashion:

(Date) 1191387600000

A number of days can be turned into a date as well

(10 days)

We currently don't provide in the Rules Engine a function to identify business days. But if such a function is provided, the following can also be used to return a date:

get the next business date for the current_date
add 10 business days to the current date
subtract 5 business days to the current date

Often you just need the next date:

get the next date for the current_date

Also provided in the past implementations but not currently in this implementation are:

get the last day of the current month for the current_date
get the first day of the current month for the current_date
get the first day of the next month for the current_date
get the first day of the previous month for the current_date
get the first day of the current year for the current_date
get the last day of the current year for the current_date
add 4 months to the current_date
subtract 4 months from the current_date

Math is also possible

(10 days) + the current_date

Names

Names are used exclusively to identify decision tables, entities, and attributes. Generally speaking there just isn't too much reason to use names except for sorting arrays of entities and possibly a few other rare and tricky purposes. However, some syntax is provided, none the less:

attribute_name	-- If some attribute is of a name type, it will return a name.
the name of Individual	-- You can always get the name of an entity
The Name namestr	– You can convert any string or string expression into a name.
$income	– A dollar in front of any identifier will force the name of the

identifier to be returned rather than interpreting it.

Strings

String expressions are heavily used in Rules. The following expressions return a string value:

```
address                                 -- A String attribute on an Entity
“1200 anywhere st.”                     -- Double quoted literal
‘1200 anywhere st.’                     -- Single quoted literal
street + “,” + state + “ “ + zipcode    -- ‘+’ provides concatenation
(string) <number>                       -- Casting a number to a string
```

Any other value added to a string has its string representation added to the string.

```
“count “ + count
```

String expressions can have the whitespace removed from the beginning and ending of the string, have their case modified, or get a current time stamp.

```
trim address

change name_str to lower case
change name_str to upper case

get a current timestamp
```

Doubles

Documented in the cup file as floats, all doubles are represented in DTRules as doubles. At some point in the future I might change this, but for now that is the way it is.

```
1.00
(double) “1.00”
1.00 + 2
```

The math is all pretty basic. Any integer combined with a double using + - * / becomes a double. Multiplication and division have a higher precedence than addition and subtraction. There are some English like statements that are also supported:

```
add to dcount 1.5
subtract from dcount 2.6
multiply dcount by 7
the absolute value of dcount
dcount rounded                          -- Truncate dcount towards 0
dcount rounded to 5 decimal places      -- Truncate towards 0
```

This rounding expression deserves a bit more discussion:

```
dcount rounded to 2 decimal places with boundary -1
```

If the number following boundary is negative, then we round towards negative infinity. If zero, then we round towards zero. If positive, we round towards positive infinity.

Integers

Integer expressions are fundamental to all policy implementations. We support the basic operators as described above, with the same precedence as one might expect. The following expressions also return integer values:

```
get days in the year of the current_date
(long) "12345"
the number of individuals          – Count the entities in an array
length of individuals              – The same thing, only more geeky
length of "this string"
add to count 1
subtract from count 1
multiply count by 5
divide count by 6
the absolute value of count
the days from start_date to end_date
get the year of the start_date
```

Boolean expressions

Boolean expressions are exclusively used for expressing conditions in decision tables. They can also be used anywhere a Boolean test is called for within syntax for defining Context statements or Actions.

```
case->allowed          Same as :case:allowed.  Pushes the case on the
                              entity stack then evaluates the attribute
                              allowed.

excluded_zips do not include the string "78717"

code_list does not include the value 12

eligible_list does not include the individual

holiday_list does not include the date current_date
```

What you search for is one of the following:

Value <integer expression>
Value <double expression>
Boolean expression
Date <date expression>

You can use either “do not include” or “does not include” to fit the implied grammer of the array name, i.e. a list is single, but a plural name is, well, plural.

You can search a list of entities using the “is there” syntax. This only works in situations where there is an array of such entities whose name is the entity name plus an ‘s’. So if you are searching for an individual, there needs to be in the context an array named individuals for this syntax to work:

```
is there an individual whose age > 18
is there a case where the number of individuals is greater than 5
```

The test can also push some entity onto the entity stack as part of the expression:

```
is there an individual in the current_case whose age < 3
```

In all these cases, you can also test for the negative, the lack of any such individual.

```
is there no individual whose age > 18
```

The tests allowed are == != > < >= <=, and in all cases the English versions work as well:

```
equal to
not equal to
greater than
less than
greater than or equal to
less than or equal to
```

Boolean attributes are also supported.

Equality is supported between objects such as names, strings, entities. Strings are compared as strings, and names as names. Entities are compared by object. Strings are compared in a case sensitive fashion. But strings and names are compared in a case insensitive fashion.

Any Boolean expression can be wrapped with:

```
[do, does, is, was] <Boolean expression> ?
```

For example:

```
is the age > 18?
was the_rejected_flag == false ?
```

Boolean expressions can be combined with == != AND and OR, and negated with a NOT. NOT has the highest precedence, == and != have the next highest precedence, AND the next, and OR the last.

All attributes can be compared to null. The Null object can be assigned to all attributes.

```
current_date is null
current_individual is null
start_date is not null
```

String tests:

```
string1 == string2
string1 at 3 starts with string2
string1 starts with string2
```

Date tests:

```
date1 is before date2
date1 is greater than date2
date1 is after date2
date1 is between date2 and date3
```

Relationships

A number of relationship tests have been added to the DSL because of the need for Relationships in many policy applications.

This syntax assumes the following structure for relationships:

Entity	Attribute	Type	Default Value	input	access
relationship	source	entity		main	Rw
relationship	target	entity		main	Rw
relationship	type	String		main	Rw
relationship	inverse	entity		main	Rw

The following statements are supported:

```
<relationship> of <entity>                    -- An Entity value

set spouse = the spouse of the individual
```

The Relationship between <entity> and <entity> -- A String value

is the Relationship between the individual.individual
and the edg_individual.individual == parent?

<entity> does not have <string> -- A Boolean value

the individual does not have a spouse

<entity> has a <string> -- A Boolean value

the individual has a parent

<entity> has a <string> [whose|which|where] <Boolean expression> -- A Boolean value

individual has a child whose age < 18

<entity> is the <string> of <entity> -- A Boolean value

individual is the child of the edg_individual.individual

Using

The Entity Stack defines the context used to interpret statements in the conditions and actions of a decision table. Entities are much like dictionaries in PostScript. A "Using Block" places an Entity on the Entity stack for the execution of a particular statement. Assume there is an attribute **individual_1** which holds a reference to an b **individual** entity. We could set the eligible flag on that individual:

```
using individual_1 { set adult = true }
```

Or if we need the context of a particular case as well as the individual, we could say:

```
using case_1, individual_1 {
set individual.adult = true;
        set case.has_adults  = true;
}
```

The Using syntax is different from the dot syntax in some rather important ways. In the example above, both a **case** entity and an **individual** entity were pushed to the top of the entity stack by the

using clause. Then the dot syntax was used to clearly indicate that the individual entity's adult attribute was set to true, and the case entity's has_adults attribute was set to true.

The using syntax has been mostly replaced with the possessive syntax. For example, you can write:

```
Set the individual_1's adult = true;
Set the case_1's has_adults = true;
```

This does the same as the previous Using syntax example. However, it is much easier to read and maintain.