Chapter 5. The Rule Language

5.1. Overview

Drools has a "native" rule language. This format is very light in terms of punctuation, and supports natural and domain specific languages via "expanders" that allow the language to morph to your problem domain. This chapter is mostly concerted with this native rule format. The diagrams used to present the syntax are known as "railroad" diagrams, and they are basically flow charts for the language terms. The technically very keen may also refer to **`DRL.g`** which is the Antlr3 grammar for the rule language. If you use the Rule Workbench, a lot of the rule structure is done for you with content assistance, for example, type "ru" and press ctrl+space, and it will build the rule structure for you.

5.1.1. A rule file

A rule file is typically a file with a .drl extension. In a DRL file you can have multiple rules, queries and functions, as well as some resource declarations like imports, globals and attributes that are assigned and used by your rules and queries. However, you are also able to spread your rules across multiple rule files (in that case, the extension .rule is suggested, but not required) - spreading rules across files can help with managing large numbers of rules. A DRL file is simply a text file.

The overall structure of a rule file is:

Example 5.1. Rules file

```
package package-name

imports

globals

functions

queries

rules
```

The order in which the elements are declared is not important, except for the package name that, if declared, must be the first element in the rules file. All elements are optional, so you will use only those you need. We will discuss each of them in the following sections.

5.1.2. What makes a rule

For the inpatients, just as an early view, a rule has the following rough structure:

```
rule "name"
    attributes
    when
        LHS
    then
        RHS
end
```

It's really that simple. Mostly punctuation is not needed, even the double quotes for "name" are optional, as are newlines. Attributes are simple (always optional) hints to how the rule should behave. LHS is the conditional parts of the rule, which follows a certain syntax which is covered below. RHS is basically a block that allows dialect specific semantic code to be executed.

It is important to note that white space is not important, *except* in the case of domain specific languages, where lines are processed one by one and spaces may be significant to the domain language.

5.2. Keywords

Drools 5 introduces the concept of *hard* and *soft* keywords.

Hard keywords are reserved, you cannot use any hard keyword when naming your domain objects, properties, methods, functions and other elements that are used in the rule text.

Here is the list of hard keywords that must be avoided as identifiers when writing rules:

- `true`
- `false`
- `null`

Soft keywords are just recognized in their context, enabling you to use these words in any other place if you wish, although, it is still recommended to avoid them, to avoid confusions, if possible. Here is a list of the soft keywords:

- `lock-on-active`
- `date-effective`
- `date-expires`
- `no-loop`
- `auto-focus`
- `activation-group`
- `agenda-group`
- `ruleflow-group`
- `entry-point`
- `duration`

- package
- import
- dialect
- salience
- enabled
- attributes
- rule
- extend
- when
- then
- template
- query
- declare
- function
- global
- eval
- not
- in
- or
- and
- exists
- forall
- accumulate
- collect
- from
- action
- reverse
- result
- end
- over
- init

Of course, you can have these (hard and soft) words as part of a method name in camel case, like notSomething() or accumulateSomething() - there are no issues with that scenario.

Although the 3 hard keywords above are unlikely to be used in your existing domain models, if you absolutely need to use them as identifiers instead of keywords, the DRL language provides the ability to escape hard keywords on rule text. To escape a word, simply enclose it in grave accents, like this:

```
Holiday( `true` == "yes" ) // please note that Drools will resolve that reference to t
```

5.3. Comments

Comments are sections of text that are ignored by the rule engine. They are stripped out when they are encountered, except inside semantic code blocks, like the RHS of a rule.

5.3.1. Single line comment

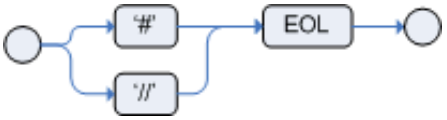

Figure 5.1. Single line comment

To create single line comments, you can use either '#' or '//'. The parser will ignore anything in the line after the comment symbol. Example:

```
rule "Testing Comments"
when
    # this is a single line comment
    // this is also a single line comment
    eval( true ) # this is a comment in the same line of a pattern
then
    // this is a comment inside a semantic code block
    # this is another comment in a semantic code block
end
```

5.3.2. Multi-line comment

Figure 5.2. Multi-line comment

Multi-line comments are used to comment blocks of text, both in and outside semantic code blocks. Example:

```
rule "Test Multi-line Comments"
when
    /* this is a multi-line comment
       in the left hand side of a rule */
    eval( true )
then
    /* and this is a multi-line comment
       in the right hand side of a rule */
end
```

5.4. Error Messages

Drools 5 introduces standardized error messages. This standardization aims to help users to find and resolve problems in a easier and faster way. In this section you will learn how to identify and interpret those error messages, and you will also receive some tips on how to solve the problems associated with them.

5.4.1. Message format

The standardization includes the error message format and to better explain this format, let's use the following example:

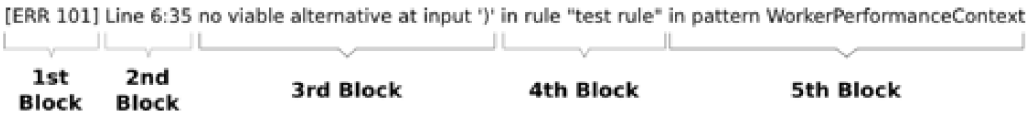

Figure 5.3. Error Message Format

1st Block: This area identifies the error code.

2nd Block: Line and column information.

3rd Block: Some text describing the problem.

4th Block: This is the first context. Usually indicates the rule, function, template or query where the error occurred. This block is not mandatory.

5th Block: Identifies the pattern where the error occurred. This block is not mandatory.

5.4.2. Error Messages Description

5.4.2.1. 101: No viable alternative

Indicates the most common errors, where the parser came to a decision point but couldn't identify an alternative. Here are some examples:

Example 5.2.

```
1: rule one
2:   when
3:     exists Foo()
4:     exits Bar()
5:   then
6: end
```

The above example generates this message:

- [ERR 101] Line 4:4 no viable alternative at input 'exits' in rule one

At first glance this seems to be valid syntax, but it is not (exits != exists). Let's take a look at next example:

Example 5.3.

```
1: package org.drools;
2: rule
3:   when
4:     Object()
5:   then
6:     System.out.println("A RHS");
7: end
```

Now the above code generates this message:

- [ERR 101] Line 3:2 no viable alternative at input 'WHEN'

This message means that the parser encountered the token **WHEN**, actually a hard keyword, but it's in the wrong place since the the rule name is missing.

The error "no viable alternative" also occurs when you make a simple lexical mistake. Here is a sample of a lexical problem:

Example 5.4.

```
1: rule simple_rule
2:   when
3:     Student( name == "Andy )
4:   then
5: end
```

The above code misses to close the quotes and because of this the parser generates this error message:

- [ERR 101] Line 0:-1 no viable alternative at input '<eof>' in rule simple_rule in pattern Student

> **Note**
>
> Usually the Line and Column information are accurate, but in some cases (like unclosed quotes), the parser generates a 0:-1 position. In this case you should check whether you didn't forget to close quotes, apostrophes or parentheses.

5.4.2.2. 102: Mismatched input

This error indicates that the parser was looking for a particular symbol that it didn't find at the current

input position. Here are some samples:

Example 5.5.

```
1: rule simple_rule
2:   when
3:     foo3 : Bar(
```

The above example generates this message:

- [ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule simple_rule in pattern Bar

To fix this problem, it is necessary to complete the rule statement.

> **Note**
>
> Usually when you get a 0:-1 position, it means that parser reached the end of source.

The following code generates more than one error message:

Example 5.6.

```
1: package org.drools;
2:
3: rule "Avoid NPE on wrong syntax"
4:   when
5:     not( Cheese( ( type == "stilton", price == 10 ) || ( type == "brie", price ==
6:   then
7:     System.out.println("OK");
8: end
```

These are the errors associated with this source:

- [ERR 102] Line 5:36 mismatched input ',' expecting ')' in rule "Avoid NPE on wrong syntax" in pattern Cheese
- [ERR 101] Line 5:57 no viable alternative at input 'type' in rule "Avoid NPE on wrong syntax"
- [ERR 102] Line 5:106 mismatched input ')' expecting 'then' in rule "Avoid NPE on wrong syntax"

Note that the second problem is related to the first. To fix it, just replace the commas (',') by AND operator ('&&').

> **Note**
>
> In some situations you can get more than one error message. Try to fix one by one, starting at the first one. Some error messages are generated merely as consequences of other errors.

5.4.2.3. 103: Failed predicate

A validating semantic predicate evaluated to false. Usually these semantic predicates are used to identify soft keywords. This sample shows exactly this situation:

Example 5.7.

```
 1: package nesting;
 2: dialect "mvel"
 3:
 4: import org.drools.Person
 5: import org.drools.Address
 6:
 7: fdsfdsfds
 8:
 9: rule "test something"
10:   when
11:     p: Person( name=="Michael" )
12:   then
13:     p.name = "other";
14:     System.out.println(p.name);
15: end
```

With this sample, we get this error message:

- [ERR 103] Line 7:0 rule 'rule_key' failed predicate: {(validateIdentifierKey(DroolsSoftKeywords.RULE))}? in rule

The **fdsfdsfds** text is invalid and the parser couldn't identify it as the soft keyword `rule`.

> **Note**
>
> This error is very similar to 102: Mismatched input, but usually involves soft keywords.

5.4.2.4. 104: Trailing semi-colon not allowed

This error is associated with the `eval` clause, where its expression may not be terminated with a semicolon. Check this example:

Example 5.8.

```
1: rule simple_rule
2:   when
3:     eval(abc();)
4:   then
```

```
5: end
```

Due to the trailing semicolon within eval, we get this error message:

- [ERR 104] Line 3:4 trailing semi-colon not allowed in rule simple_rule

This problem is simple to fix: just remove the semi-colon.

5.4.2.5. 105: Early Exit

The recognizer came to a subrule in the grammar that must match an alternative at least once, but the subrule did not match anything. Simply put: the parser has entered a branch from where there is no way out. This example illustrates it:

Example 5.9.

```
1: template test_error
2:   aa s  11;
3: end
```

This is the message associated to the above sample:

- [ERR 105] Line 2:2 required (...)+ loop did not match anything at input 'aa' in template test_error

To fix this problem it is necessary to remove the numeric value as it is neither a valid data type which might begin a new template slot nor a possible start for any other rule file construct.

5.4.3. Other Messages

Any other message means that something bad has happened, so please contact the development team.

5.5. Package

A package is a collection of rules and other related constructs, such as imports and globals. The package members are typically related to each other - perhaps HR rules, for instance. A package represents a namespace, which ideally is kept unique for a given grouping of rules. The package name itself is the namespace, and is not related to files or folders in any way.

It is possible to assemble rules from multiple rule sources, and have one top level package configuration that all the rules are kept under (when the rules are assembled). Although, it is not possible to merge into the same package resources declared under different names. A single Rulebase may, however, contain multiple packages built on it. A common structure is to have all the rules for a package in the same file as the package declaration (so that is it entirely self-contained).

The following railroad diagram shows all the components that may make up a package. Note that a package *must* have a namespace and be declared using standard Java conventions for package names; i.e., no spaces, unlike rule names which allow spaces. In terms of the order of elements, they can appear in

any order in the rule file, with the exception of the `package` statement, which must be at the top of the file. In all cases, the semicolons are optional.

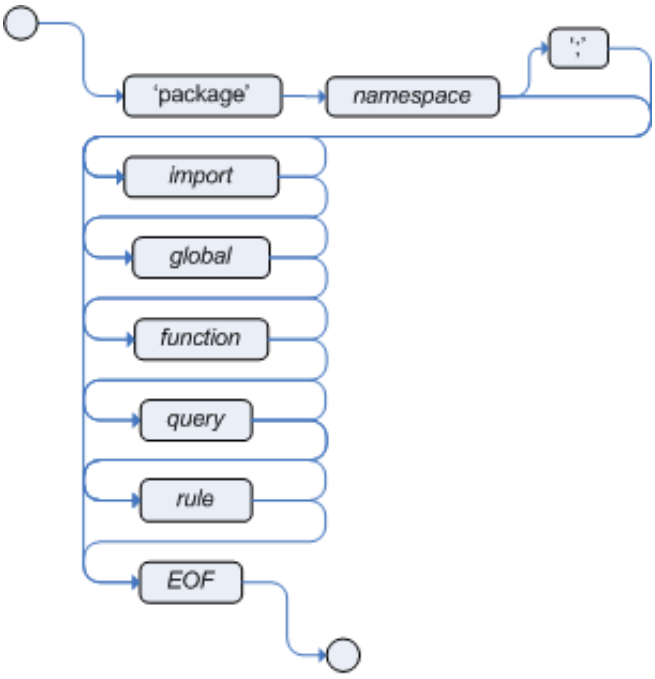

Figure 5.4. package

Notice that any rule atttribute (as described the section Rule Attributes) may also be written at package level, superseding the attribute's default value. The modified default may still be replaced by an attribute setting within a rule.

5.5.1. import

Figure 5.5. import

Import statements work like import statements in Java. You need to specify the fully qualified paths and type names for any objects you want to use in the rules. Drools automatically imports classes from the Java package of the same name, and also from the package `java.lang`.

5.5.2. global

Figure 5.6. global

With `global` you define global variables. They are used to make application objects available to the rules. Typically, they are used to provide data or services that the rules use, especially application services used in rule consequences, and to return data from the rules, like logs or values added in rule consequences, or for the rules to interact with the application, doing callbacks. Globals are not inserted into the Working Memory, and therefore a global should never be used to establish conditions in rules except when it has a constant immutable value. The engine cannot be notified about value changes of globals and does not track their changes. Incorrect use of globals in constraints may yield surprising results - surprising in a bad way.

If multiple packages declare globals with the same identifier they must be of the same type and all of them will reference the same global value.

In order to use globals you must:

1. Declare your global variable in your rules file and use it in rules. Example:

```
global java.util.List myGlobalList;

rule "Using a global"
when
    eval( true )
then
    myGlobalList.add( "Hello World" );
end
```

2. Set the global value on your working memory. It is a best practice to set all global values before asserting any fact to the working memory. Example:

```
List list = new ArrayList();
WorkingMemory wm = rulebase.newStatefulSession();
wm.setGlobal( "myGlobalList", list );
```

Note that these are just named instances of objects that you pass in from your application to the working memory. This means you can pass in any object you want: you could pass in a service locator, or perhaps a service itself. With the new `from` element it is now common to pass a Hibernate session as a global, to allow `from` to pull data from a named Hibernate query.

One example may be an instance of a Email service. In your integration code that is calling the rule engine, you obtain your emailService object, and then set it in the working memory. In the DRL, you declare that you have a global of type EmailService, and give it the name "email". Then in your rule consequences, you can use things like email.sendSMS(number, message).

Globals are not designed to share data between rules and they should never be used for that purpose. Rules always reason and react to the working memory state, so if you want to pass data from rule to rule, assert the data as facts into the working memory.

It is strongly discouraged to set or change a global value from inside your rules. We recommend to you always set the value from your application using the working memory interface.

5.6. Function

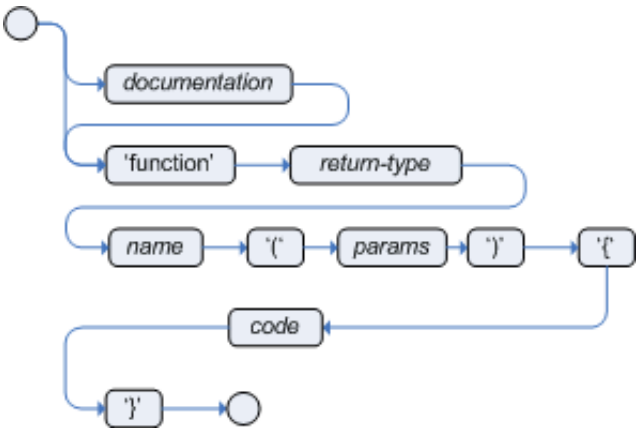

Figure 5.7. function

Functions are a way to put semantic code in your rule source file, as opposed to in normal Java classes. They can't do anything more than what you can do with helper classes. (In fact, the compiler generates the helper class for you behind the scenes.) The main advantage of using functions in a rule is that you can keep the logic all in one place, and you can change the functions as needed (which can be a good or a bad thing). Functions are most useful for invoking actions on the consequence (`then`) part of a rule, especially if that particular action is used over and over again, perhaps with only differing parameters for each rule.

A typical function declaration looks like:

```
function String hello(String name) {
    return "Hello "+name+"!";
}
```

Note that the `function` keyword is used, even though its not really part of Java. Parameters to the function are defined as for a method, and you don't have to have parameters if they are not needed. The return type is defined just like in a regular method.

Alternatively, you could use a static method in a helper class, e.g., `Foo.hello()`. Drools supports the use of function imports, so all you would need to do is:

```
import function my.package.Foo.hello
```

Irrespective of the way the function is defined or imported, you use a function by calling it by its name, in the consequence or inside a semantic code block. Example:

```
rule "using a static function"
when
    eval( true )
then
    System.out.println( hello( "Bob" ) );
end
```

5.7. Type Declaration

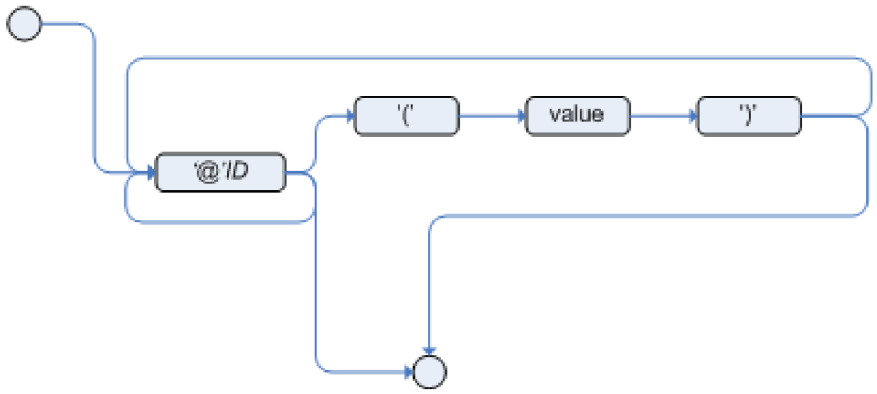

Figure 5.8. meta_data

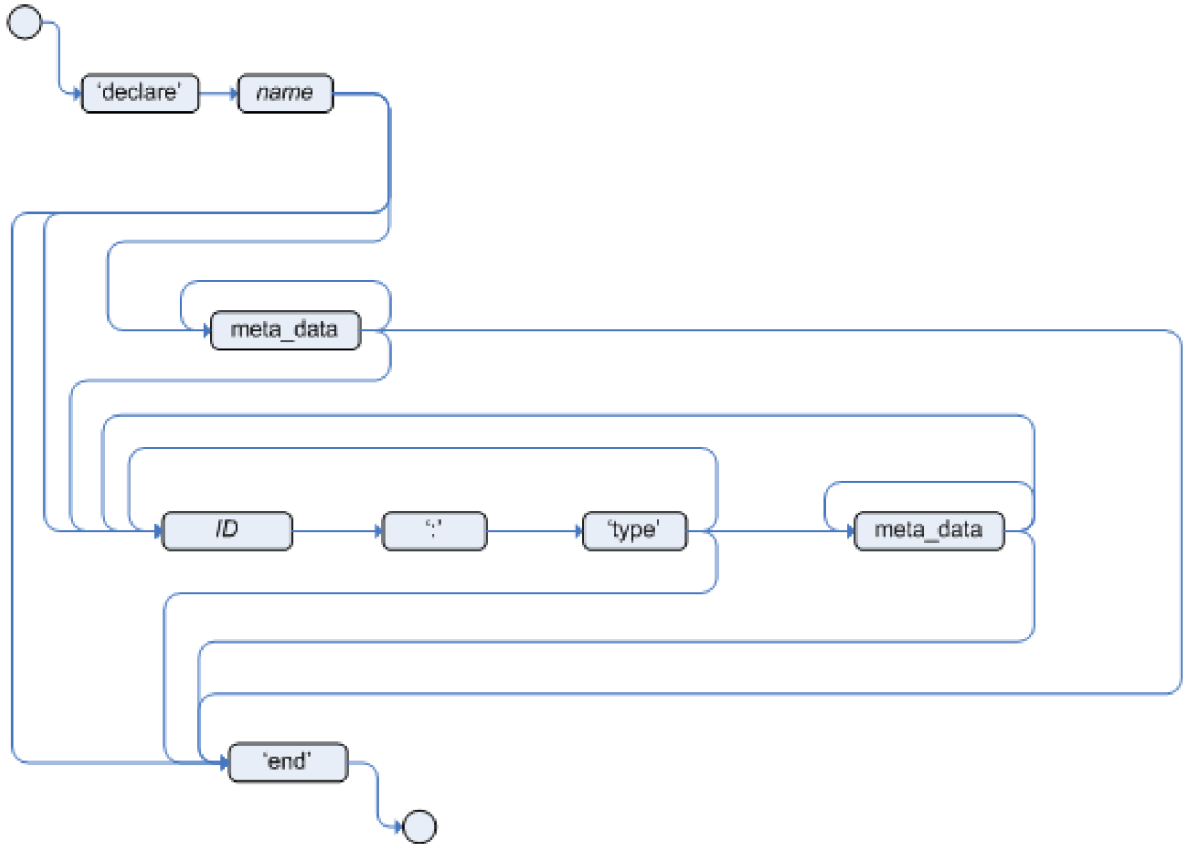

Figure 5.9. type_declaration

Type declarations have two main goals in the rules engine: to allow the declaration of new types, and to allow the declaration of metadata for types.

- **Declaring new types:** Drools works out of the box with plain Java objects as facts. Sometimes, however, users may want to define the model directly to the rules engine, without worrying about creating models in a lower level language like Java. At other times, there is a domain model already built, but eventually the user wants or needs to complement this model with additional entities that are used mainly during the reasoning process.
- **Declaring metadata:** facts may have meta information associated to them. Examples of meta information include any kind of data that is not represented by the fact attributes and is consistent among all instances of that fact type. This meta information may be queried at runtime by the engine and used in the reasoning process.

5.7.1. Declaring New Types

To declare a new type, all you need to do is use the keyword `declare`, followed by the list of fields, and the keyword `end`.

Example 5.10. Declaring a new fact type: Address

```
declare Address
   number : int
   streetName : String
   city : String
end
```

The previous example declares a new fact type called `Address`. This fact type will have three attributes: `number`, `streetName` and `city`. Each attribute has a type that can be any valid Java type, including any other class created by the user or even other fact types previously declared.

For instance, we may want to declare another fact type `Person`:

Example 5.11. declaring a new fact type: Person

```
declare Person
    name : String
    dateOfBirth : java.util.Date
    address : Address
end
```

As we can see on the previous example, `dateOfBirth` is of type `java.util.Date`, from the Java API, while `address` is of the previously defined fact type Address.

You may avoid having to write the fully qualified name of a class every time you write it by using the `import` clause, as previously discussed.

Example 5.12. Avoiding the need to use fully qualified class names by using import

```
import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

When you declare a new fact type, Drools will, at compile time, generate bytecode that implements a Java class representing the fact type. The generated Java class will be a one-to-one Java Bean mapping of the type definition. So, for the previous example, the generated Java class would be:

Example 5.13. generated Java class for the previous Person fact type declaration

```
public class Person implements Serializable {
    private String name;
    private java.util.Date dateOfBirth;
    private Address address;

    // empty constructor
    public Person() {...}

    // constructor with all fields
    public Person( String name, Date dateOfBirth, Address address ) {...}

    // if keys are defined, constructor with keys
    public Person( ...keys... ) {...}

    // getters and setters
    // equals/hashCode
    // toString
}
```

Since the generated class is a simple Java class, it can be used transparently in the rules, like any other fact.

Example 5.14. Using the declared types in rules

```
rule "Using a declared Type"
when
    $p : Person( name == "Bob" )
then
    // Insert Mark, who is Bob's mate.
    Person mark = new Person();
    mark.setName("Mark");
    insert( mark );
end
```

5.7.2. Declaring Metadata

Metadata may be assigned to several different constructions in Drools: fact types, fact attributes and rules. Drools uses the at sign ('@') to introduce metadata, and it always uses the form:

```
@metadata_key( metadata_value )
```

The parenthesized *metadata_value* is optional.

For instance, if you want to declare a metadata attribute like `author`, whose value is *Bob*, you could simply write:

Example 5.15. Declaring a metadata attribute

```
@author( Bob )
```

Drools allows the declaration of any arbitrary metadata attribute, but some will have special meaning to the engine, while others are simply available for querying at runtime. Drools allows the declaration of metadata both for fact types and for fact attributes. Any metadata that is declared before the attributes of a fact type are assigned to the fact type, while metadata declared after an attribute are assigned to that particular attribute.

Example 5.16. Declaring metadata attributes for fact types and attributes

```
import java.util.Date

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )

    name : String @key @maxLength( 30 )
    dateOfBirth : Date
    address : Address
end
```

In the previous example, there are two metadata items declared for the fact type (`@author` and `@dateOfCreation`) and two more defined for the name attribute (`@key` and `@maxLength`). Please note that the `@key` metadata has no required value, and so the parentheses and the value were omitted.:

@position can be used to declare the position of a field, overriding the default declared order. This is used for positional constraints in patterns.

```
declare Cheese
    name : String @position(1)
    shop : String @position(2)
    price : int @position(0)
end
```

5.7.3. Declaring Metadata for Existing Types

Drools allows the declaration of metadata attributes for existing types in the same way as when declaring metadata attributes for new fact types. The only difference is that there are no fields in that declaration.

For instance, if there is a class org.drools.examples.Person, and one wants to declare metadata for it, it's possible to write the following code:

Example 5.17. Declaring metadata for an existing type

```
import org.drools.examples.Person

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

Instead of using the import, it is also possible to reference the class by its fully qualified name, but since the class will also be referenced in the rules, it is usually shorter to add the import and use the short class name everywhere.

Example 5.18. Declaring metadata using the fully qualified class name

```
declare org.drools.examples.Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

5.7.4. Parameterized constructors for declared types

Generate constructors with parameters for declared types.

Example: for a declared type like the following:

```
declare Person
    firstName : String @key
    lastName : String @key
    age : int
end
```

The compiler will implicitly generate 3 constructors: one without parameters, one with the @key fields, and one with all fields.

```
Person() // parameterless constructor
Person( String firstName, String lastName )
Person( String firstName, String lastName, int age )
```

5.7.5. Non Typesafe Classes

@typesafe(<boolean>) has been added to type declarations. By default all type declarations are compiled with type safety enabled; @typesafe(false) provides a means to override this behaviour by permitting a fall-back, to type unsafe evaluation where all constraints are generated as MVEL constraints and executed dynamically. This can be important when dealing with collections that do not have any generics or mixed type collections.

5.7.6. Accessing Declared Types from the Application Code

Declared types are usually used inside rules files, while Java models are used when sharing the model between rules and applications. Although, sometimes, the application may need to access and handle facts from the declared types, especially when the application is wrapping the rules engine and providing higher level, domain specific user interfaces for rules management.

In such cases, the generated classes can be handled as usual with the Java Reflection API, but, as we know, that usually requires a lot of work for small results. Therefore, Drools provides a simplified API for the most common fact handling the application may want to do.

The first important thing to realize is that a declared fact will belong to the package where it was declared. So, for instance, in the example below, `Person` will belong to the `org.drools.examples` package, and so the fully qualified name of the generated class will be `org.drools.examples.Person`.

Example 5.19. Declaring a type in the org.drools.examples package

```
package org.drools.examples

import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

Declared types, as discussed previously, are generated at knowledge base compilation time, i.e., the application will only have access to them at application run time. Therefore, these classes are not available for direct reference from the application.

Drools then provides an interface through which users can handle declared types from the application code: `org.drools.definition.type.FactType`. Through this interface, the user can instantiate, read and write fields in the declared fact types.

Example 5.20. Handling declared fact types through the API

```
// get a reference to a knowledge base with a declared type:
KnowledgeBase kbase = ...

// get the declared FactType
FactType personType = kbase.getFactType( "org.drools.examples",
                                         "Person" );

// handle the type as necessary:
// create instances:
Object bob = personType.newInstance();

// set attributes values
personType.set( bob,
```

```
                "name",
                "Bob" );
personType.set( bob,
                "age",
                42 );

// insert fact into a session
StatefulKnowledgeSession ksession = ...
ksession.insert( bob );
ksession.fireAllRules();

// read attributes
String name = personType.get( bob, "name" );
int age = personType.get( bob, "age" );
```

The API also includes other helpful methods, like setting all the attributes at once, reading values from a Map, or reading all attributes at once, into a Map.

Although the API is similar to Java reflection (yet much simpler to use), it does not use reflection underneath, relying on much more performant accessors implemented with generated bytecode.

5.7.7. Type Declaration 'extends'

Type declarations now support 'extends' keyword for inheritance

In order to extend a type declared in Java by a DRL declared subtype, repeat the supertype in a declare statement without any fields.

```
import org.people.Person

declare Person
end

declare Student extends Person
    school : String
end

declare LongTermStudent extends Student
    years : int
    course : String
end
```

5.8. Rule

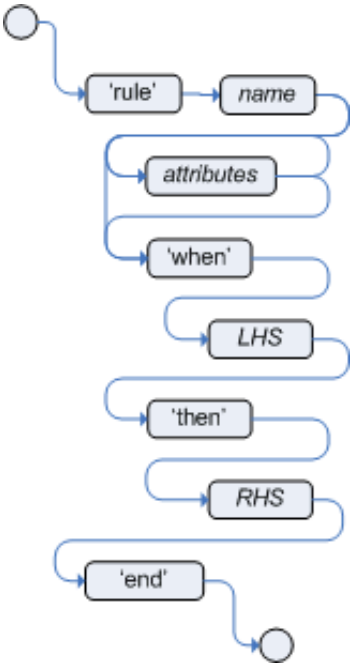

Figure 5.10. rule

A rule specifies that *when* a particular set of conditions occur, specified in the Left Hand Side (LHS), *then* do what queryis specified as a list of actions in the Right Hand Side (RHS). A common question from users is "Why use when instead of if?" "When" was chosen over "if" because "if" is normally part of a procedural execution flow, where, at a specific point in time, a condition is to be checked. In contrast, "when" indicates that the condition evaluation is not tied to a specific evaluation sequence or point in time, but that it happens continually, at any time during the life time of the engine; whenever the condition is met, the actions are executed.

A rule must have a name, unique within its rule package. If you define a rule twice in the same DRL it produces an error while loading. If you add a DRL that includes a rule name already in the package, it replaces the previous rule. If a rule name is to have spaces, then it will need to be enclosed in double quotes (it is best to always use double quotes).

Attributes - described below - are optional. They are best written one per line.

The LHS of the rule follows the `when` keyword (ideally on a new line), similarly the RHS follows the `then` keyword (again, ideally on a newline). The rule is terminated by the keyword `end`. Rules cannot be nested.

Example 5.21. Rule Syntax Overview

```
rule "<name>"
    <attribute>*
when
    <conditional element>*
then
    <action>*
end
```

Example 5.22. A simple rule

```
rule "Approve if not rejected"
  salience -100
  agenda-group "approval"
    when
        not Rejection()
        p : Policy(approved == false, policyState:status)
        exists Driver(age > 25)
        Process(status == policyState)
    then
        log("APPROVED: due to no objections.");
        p.setApproved(true);
end
```

5.8.1. Rule Attributes

Rule attributes provide a declarative way to influence the behavior of the rule. Some are quite simple, while others are part of complex subsystems such as ruleflow. To get the most from Drools you should make sure you have a proper understanding of each attribute.

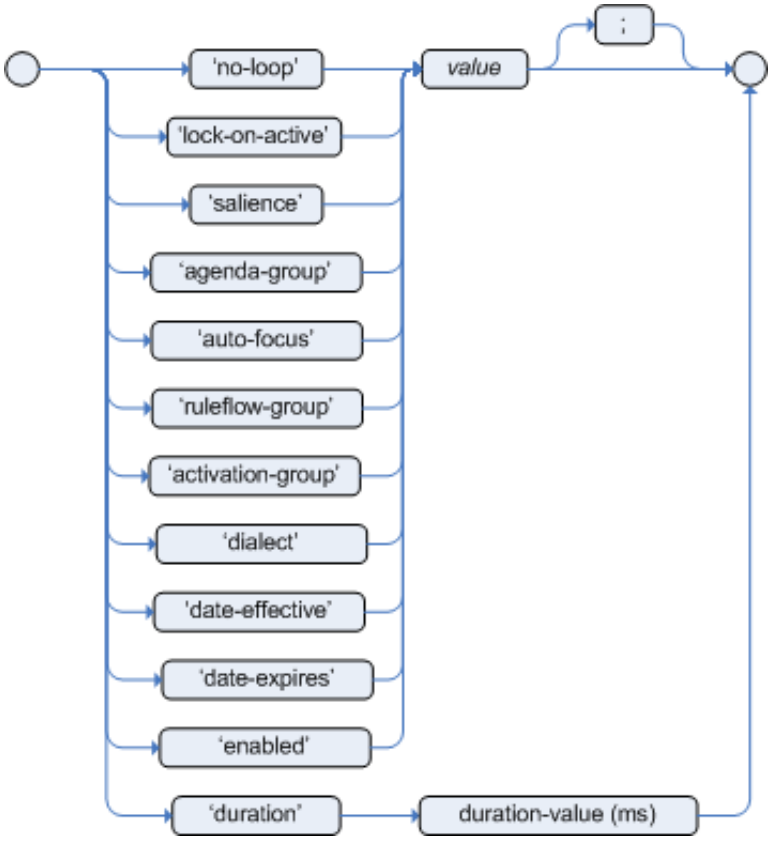

Figure 5.11. rule attributes

`no-loop`

default value: `false`

type: Boolean

When a rule's consequence modifies a fact it may cause the rule to activate again, causing an infinite loop. Setting no-loop to true will skip the creation of another Activation for the rule with the current set of facts.

`ruleflow-group`

default value: N/A

type: String

Ruleflow is a Drools feature that lets you exercise control over the firing of rules. Rules that are assembled by the same ruleflow-group identifier fire only when their group is active.

`lock-on-active`

default value: `false`

type: Boolean

Whenever a ruleflow-group becomes active or an agenda-group receives the focus, any rule within that group that has lock-on-active set to true will not be activated any more; irrespective of the origin of the update, the activation of a matching rule is discarded. This is a stronger version of no-loop, because the change could now be caused not only by the rule itself. It's ideal for calculation rules where you have a number of rules that modify a fact and you don't want any rule re-matching and firing again. Only when the ruleflow-group is no longer active or the agenda-group loses the focus those rules with lock-on-active set to true become eligible again for their activations to be placed onto the agenda.

`salience`

default value: `0`

type: integer

Each rule has an integer salience attribute which defaults to zero and can be negative or positive. Salience is a form of priority where rules with higher salience values are given higher priority when ordered in the Activation queue.

Drools also supports dynamic salience where you can use an expression involving bound variables.

Example 5.23. Dynamic Salience

```
rule "Fire in rank order 1,2,.."
        salience( -$rank )
    when
        Element( $rank : rank,... )
    then
        ...
end
```

`agenda-group`

default value: MAIN

type: String

Agenda groups allow the user to partition the Agenda providing more execution control. Only rules in the agenda group that has acquired the focus are allowed to fire.

`auto-focus`

default value: `false`

type: Boolean

When a rule is activated where the `auto-focus` value is true and the rule's agenda group does not have focus yet, then it is given focus, allowing the rule to potentially fire.

`activation-group`

default value: N/A

type: String

Rules that belong to the same activation-group, identified by this attribute's string value, will only fire exclusively. In other words, the first rule in an activation-group to fire will cancel the other rules' activations, i.e., stop them from firing.

Note: This used to be called Xor group, but technically it's not quite an Xor. You may still hear people mention Xor group; just swap that term in your mind with activation-group.

`dialect`

default value: as specified by the package

type: String

possible values: "java" or "mvel"

The dialect species the language to be used for any code expressions in the LHS or the RHS code block. Currently two dialects are available, Java and MVEL. While the dialect can be specified at the package level, this attribute allows the package definition to be overridden for a rule.

`date-effective`

default value: N/A

type: String, containing a date and time definition

A rule can only activate if the current date and time is after date-effective attribute.

`date-expires`

default value: N/A

type: String, containing a date and time definition

A rule cannot activate if the current date and time is after the date-expires attribute.

`duration`

default value: no default value

type: long

The duration dictates that the rule will fire after a specified duration, if it is still true.

Example 5.24. Some attribute examples

```
rule "my rule"
  salience 42
  agenda-group "number 1"
    when ...
```

5.8.2. Timers and Calendars

Rules now suport both interval and cron based timers, which replace the now deprecated duration attribute.

Example 5.25. Sample timer attribute uses

```
timer ( int: <initial delay> <repeat interval>? )
timer ( int: 30s )
timer ( int: 30s 5m )

timer ( cron: <cron expression> )
timer ( cron:* 0/15 * * * ? )
```

Interval (indicated by "int:") timers follow the semantics of java.util.Timer objects, with an initial delay and an optional repeat interval. Cron (indicated by "cron:") timers follow standard Unix cron expressions:

Example 5.26. A Cron Example

```
rule "Send SMS every 15 minutes"
    timer (cron:* 0/15 * * * ?)
when
    $a : Alarm( on == true )
then
    channels[ "sms" ].insert( new Sms( $a.mobileNumber, "The alarm is still on" );
end
```

Calendars are used to control when rules can fire. The Calendar API is modelled on Quartz:

Example 5.27. Adapting a Quartz Calendar

```
Calendar weekDayCal = QuartzHelper.quartzCalendarAdapter(org.quartz.Calendar quartzC
```

Calendars are registered with the StatefulKnowledgeSession:

Example 5.28. Registering a Calendar

```
ksession.getCalendars().set( "weekday", weekDayCal );
```

They can be used in conjunction with normal rules and rules including timers. The rule attribute "calendars" may contain one or more comma-separated calendar names written as string literals.

Example 5.29. Using Calendars and Timers together

```
rule "weekdays are high priority"
   calendars "weekday"
   timer (int:0 1h)
when
    Alarm()
then
    send( "priority high - we have an alarm );
end

rule "weekend are low priority"
   calendars "weekend"
   timer (int:0 4h)
when
    Alarm()
then
    send( "priority low - we have an alarm );
end
```

5.8.3. Left Hand Side (when) syntax

5.8.3.1. What is the Left Hand Side?

The Left Hand Side (LHS) is a common name for the conditional part of the rule. It consists of zero or more Conditional Elements. If the LHS is empty, it will be considered as a condition element that is always true and it will be activated once, when a new WorkingMemory session is created.

Figure 5.12. Left Hand Side

Example 5.30. Rule without a Conditional Element

```
rule "no CEs"
when
    // empty
then
    ... // actions (executed once)
end
```

```
# The above rule is internally rewritten as:

rule "eval(true)"
when
    eval( true )
then
    ... // actions (executed once)
end
```

Conditional elements work on one or more *patterns* (which are described below). The most common conditional element is "and". Therefore it is implicit when you have multiple patterns in the LHS of a rule that are not connected in any way:

Example 5.31. Implicit and

```
rule "2 unconnected patterns"
when
    Pattern1()
    Pattern2()
then
    ... // actions
end

# The above rule is internally rewritten as:

rule "2 and connected patterns"
when
    Pattern1()
    and Pattern2()
then
    ... // actions
end
```

Note

An "and" cannot have a leading declaration binding (unlike for example or). This is obvious, since a declaration can only reference a single fact at a time, and when the "and" is satisfied it matches both facts - so which fact would the declaration bind to?

```
// Compile error
$person : (Person( name == "Romeo" ) and Person( name == "Juliet"))
```

5.8.3.2. Pattern (conditional element)

5.8.3.2.1. What is a pattern?

A pattern element is the most important Conditional Element. It can potentially match on each fact that is inserted in the working memory.

A pattern contains of zero or more constraints and has an optional pattern binding. The railroad diagram below shows the syntax for this.

Figure 5.13. Pattern

In its simplest form, with no constraints, a pattern matches against a fact of the given type. In the following case the type is `Cheese`, which means that the pattern will match against all `Person` objects in the Working Memory:

```
Person()
```

The type need not be the actual class of some fact object. Patterns may refer to superclasses or even interfaces, thereby potentially matching facts from many different classes.

```
Object() // matches all objects in the working memory
```

Inside of the pattern parenthesis is where all the action happens: it defines the constraints for that pattern. For example, with a age related constraint:

```
Person( age == 100 )
```

> **Note**
>
> For backwards compatibility reasons it's allowed to suffix patterns with the ; character. But it is not recommended to do that.

5.8.3.2.2. Pattern binding

For referring to the matched object, use a pattern binding variable such as `$p`.

Example 5.32. Pattern with a binding variable

```
rule ...
when
    $p : Person()
then
    System.out.println( "Person " + $p );
end
```

The prefixed dollar symbol (`$`) is just a convention; it can be useful in complex rules where it helps to easily differentiate between variables and fields, but it is not mandatory.

5.8.3.3. Constraint (part of a pattern)

5.8.3.3.1. What is a constraint?

A constraint is an expression that returns `true` or `false`. This example has a constraint that states *5 is smaller than 6*:

```
Person( 5 < 6 )  // just an example, as constraints like this would be useless in a re
```

In essence, it's a Java expression with some enhancements (such as property access) and a few differences (such as `equals()` semantics for ==). Let's take a deeper look.

5.8.3.3.2. Property access on Java Beans (POJO's)

Any bean property can be used directly. A bean property is exposed using a standard Java bean getter: a method `getMyProperty()` (or `isMyProperty()` for a primitive boolean) which takes no arguments and return something. For example: the age property is written as `age` in DRL instead of the getter `getAge()`:

```
Person( age == 50 )

// this is the same as:
Person( getAge() == 50 )
```

Drools uses the standard JDK `Introspector` class to do this mapping, so it follows the standard Java bean specification.

> **Note**
>
> We recommend using property access (`age`) over using getters explicitly (`getAge()`) because of performance enhancements through field indexing.

> **Warning**
>
> Property accessors must not change the state of the object in a way that may effect the rules. Remember that the rule engine effectively caches the results of its matching in between invocations to make it faster.

```
public int getAge() {
    age++; // Do NOT do this
    return age;
}
```

```
public int getAge() {
    Date now = DateUtil.now(); // Do NOT do this
```

```
        return DateUtil.differenceInYears(now, birthday);
    }
```

To solve this latter case, insert a fact that wraps the current date into working memory and update that fact between `fireAllRules` as needed.

Note

The following fallback applies: if the getter of a property cannot be found, the compiler will resort to using the property name as a method name and without arguments:

```
Person( age == 50 )

// If Person.getAge() does not exists, this falls back to:
Person( age() == 50 )
```

Nested property access is also supported:

```
Person( address.houseNumber == 50 )

// this is the same as:
Person( getAddress().getHouseNumber() == 50 )
```

Nested properties are also indexed.

Warning

In a stateful session, care should be taken when using nested accessors as the Working Memory is not aware of any of the nested values, and does not know when they change. Either consider them immutable while any of their parent references are inserted into the Working Memory. Or, instead, if you wish to modify a nested value you should mark all of the outer facts as updated. In the above example, when the `houseNumber` changes, any `Person` with that `Address` must be marked as updated.

5.8.3.3.3. Java expression

You can use any Java expression that returns a `boolean` as a constraint inside the parentheses of a pattern. Java expressions can be mixed with other expression enhancements, such as property access:

```
Person( age == 50 )
```

It is possible to change the evaluation priority by using parentheses, as in any logic or mathematical expression:

```
Person( age > 100 && ( age % 10 == 0 ) )
```

It is possible to reuse Java methods:

```
Person( Math.round( weight / ( height * height ) ) < 25.0 )
```

Warning

As for property accessors, methods must not change the state of the object in a way that may affect the rules. Any method executed on a fact in the LHS should be a *read only* method.

```
Person( incrementAndGetAge() == 10 ) // Do NOT do this
```

Warning

The state of a fact should not change between rule invocations (unless those facts are marked as updated to the working memory on every change):

```
Person( System.currentTimeMillis() % 1000 == 0 ) // Do NOT do this
```

Normal Java operator precedence applies, see the operator precedence list below.

Important

All operators have normal Java semantics except for `==` and `!=`.

The `==` operator has null-safe `equals()` semantics:

```
// Similar to: java.util.Objects.equals(person.getFirstName(), "John")
// so (because "John" is not null) similar to:
// "John".equals(person.getFirstName())
Person( firstName == "John" )
```

The `!=` operator has null-safe `!equals()` semantics:

```
// Similar to: !java.util.Objects.equals(person.getFirstName(), "John")
Person( firstName != "John" )
```

Type coercion is always attempted if the field and the value are of different types; exceptions will be thrown if a bad coercion is attempted. For instance, if "ten" is provided as a string in a numeric evaluator, an exception is thrown, whereas "10" would coerce to a numeric 10. Coercion is always in favor of the field type and not the value type:

```
Person( age == "10" ) // "10" is coerced to 10
```

5.8.3.3.4. Comma separated AND

The comma character (',') is used to separate constraint groups. It has implicit *AND* connective semantics.

```
// Person is at least 50 and weighs at least 80 kg
Person( age > 50, weight > 80 )
```

```
// Person is at least 50, weighs at least 80 kg and is taller than 2 meter.
Person( age > 50, weight > 80, height > 2 )
```

> **Note**
>
> Although the && and , operators have the same semantics, they are resolved with different priorities: The && operator precedes the || operator. Both the && and || operator precede the , operator. See the operator precedence list below.
> The comma operator should be preferred at the top level constraint, as it makes constraints easier to read and the engine will often be able to optimize them better.

The comma (,) operator cannot be embedded in a composite constraint expression, such as parentheses:

```
Person( ( age > 50, weight > 80 ) || height > 2 ) // Do NOT do this: compile error

// Use this instead
Person( ( age > 50 && weight > 80 ) || height > 2 )
```

5.8.3.3.5. Binding variables

A property can be bound to a variable:

```
// 2 persons of the same age
Person( $firstAge : age ) // binding
Person( age == $firstAge ) // constraint expression
```

The prefixed dollar symbol ($) is just a convention; it can be useful in complex rules where it helps to easily differentiate between variables and fields.

> **Note**
>
> For backwards compatibility reasons, It's allowed (but not recommended) to mix a constraint binding and constraint expressions as such:
>
> ```
> // Not recommended
> Person( $age : age * 2 < 100 )
> ```
>
> ```
> // Recommended (seperates bindings and constraint expressions)
> Person( age * 2 < 100, $age : age )
> ```

Bound variable restrictions using the operator == provide for very fast execution as it use hash indexing to improve performance.

5.8.3.3.6. Unification

Drools does not allow bindings to the same declaration. However this is an important aspect to derivation query unification. While positional arguments are always processed with unification a special unification symbol, ':=', was introduced for named arguments named arguments. The following "unifies" the age argument across two people.

```
Person( $age := age )
Person( $age := age)
```

In essence unification will declare a binding for the first occurance, and constrain to the same value of the bound field for sequence occurances.

5.8.3.3.7. Special literal support

Besides normal Java literals (including Java 5 enums), this literal is also supported:

5.8.3.3.7.1. Date literal

The date format `dd-mmm-yyyy` is supported by default. You can customize this by providing an alternative date format mask as the System property named `drools.dateformat`. If more control is required, use a restriction.

Example 5.33. Date Literal Restriction

```
Cheese( bestBefore < "27-Oct-2009" )
```

5.8.3.3.8. List and Map access

It's possible to directly access a `List` value by index:

```
// Same as childList(0).getAge() == 18
Person( childList[0].age == 18 )
```

It's also possible to directly access a `Map` value by key:

```
// Same as credentialMap.get("jsmith").isValid()
Person( credentialMap["jsmith"].valid )
```

5.8.3.3.9. Abbreviated combined relation condition

This allows you to place more than one restriction on a field using the restriction connectives `&&` or `||`. Grouping via parentheses is permitted, resulting in a recursive syntax pattern.

Figure 5.14. Abbreviated combined relation condition

Figure 5.15. Abbreviated combined relation condition with parentheses

```
// Simple abbreviated combined relation condition using a single &&
Person( age > 30 && < 40 )
```

```
// Complex abbreviated combined relation using groupings
Person( age ( (> 30 && < 40) ||
              (> 20 && < 25) ) )
```

```
// Mixing abbreviated combined relation with constraint connectives
Person( age > 30 && < 40 || location == "london" )
```

5.8.3.3.10. Special DRL operators

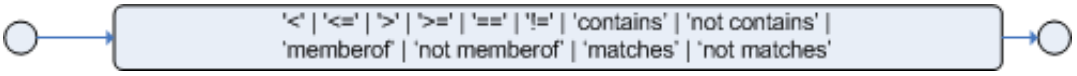

Figure 5.16. Operators

Coercion to the correct value for the evaluator and the field will be attempted.

5.8.3.3.10.1. The operators < <= > >=

These operators can be used on properties with natural ordering. For example, for Date fields, < means *before*, for `String` fields, it means alphabetically lower.

```
Person( firstName < $otherFirstName )
```

```
Person( birthDate < $otherBirthDate )
```

Only applies on `Comperable` properties.

5.8.3.3.10.2. The operator `matches`

Matches a field against any valid Java Regular Expression. Typically that regexp is a string literal, but variables that resolve to a valid regexp are also allowed.

Example 5.34. Regular Expression Constraint

```
Cheese( type matches "(Buffalo)?\\S*Mozarella" )
```

> **Note**
>
> Like in Java, regular expressions written as string literals *need to escape* '\'.
> Up to Drools 5.1 there was a configuration to support non-escaped regexps from Drools 4.0 for backwards compatilibity. From Drools 5.2 this is no longer supported.

Only applies on `String` properties.

5.8.3.3.10.3. The operator `not matches`

The operator returns true if the String does not match the regular expression. The same rules apply as for the `matches` operator. Example:

Example 5.35. Regular Expression Constraint

```
Cheese( type not matches "(Buffulo)?\\S*Mozarella" )
```

Only applies on `String` properties.

5.8.3.3.10.4. The operator `contains`

The operator `contains` is used to check whether a field that is a Collection or array contains the specified value.

Example 5.36. Contains with Collections

```
CheeseCounter( cheeses contains "stilton" ) // contains with a String literal
CheeseCounter( cheeses contains $var ) // contains with a variable
```

Only applies on `Collection` properties.

5.8.3.3.10.5. The operator `not contains`

The operator `not contains` is used to check whether a field that is a Collection or array does *not* contain the specified value.

Example 5.37. Literal Constraint with Collections

```
CheeseCounter( cheeses not contains "cheddar" ) // not contains with a String litera
CheeseCounter( cheeses not contains $var ) // not contains with a variable
```

Only applies on `Collection` properties.

> **Note**
>
> For backward compatibility, the `excludes` operator is supported as a synonym for `not contains`.

5.8.3.3.10.6. The operator `memberOf`

The operator `memberOf` is used to check whether a field is a member of a collection or array.

Example 5.38. Literal Constraint with Collections

```
CheeseCounter( cheese memberOf $matureCheeses )
```

```
Person( father memberOf parents )
```

5.8.3.3.10.7. The operator `not memberOf`

The operator `not memberOf` is used to check whether a field is not a member of a collection or array.

Example 5.39. Literal Constraint with Collections

```
CheeseCounter( cheese not memberOf $matureCheeses )
```

```
Person( father not memberOf childern )
```

5.8.3.3.10.8. The operator `soundslike`

This operator is similar to `matches`, but it checks whether a word has almost the same sound (using English pronunciation) as the given value. This is based on the Soundex algorithm (see `http://en.wikipedia.org/wiki/Soundex`).

Example 5.40. Test with soundslike

```
// match cheese "fubar" or "foobar"
Cheese( name soundslike 'foobar' )
```

5.8.3.3.10.9. The operator `str`

This operator `str` is used to check whether a field that is a `String` starts with or ends with a certain value. It can also be used to check the length of the String.

```
Message( routingValue str[startsWith] "R1" )
```

```
Message( routingValue str[endsWith] "R2" )
```

```
Message( routingValue str[length] 17 )
```

5.8.3.3.10.10. The operators `in` and `not in` (compound value restriction)

The compound value restriction is used where there is more than one possible value to match. Currently only the `in` and `not in` evaluators support this. The second operand of this operator must be a comma-separated list of values, enclosed in parentheses. Values may be given as variables, literals, return values or qualified identifiers. Both evaluators are actually *syntactic sugar*, internally rewritten as a list of multiple restrictions using the operators `!=` and `==`.

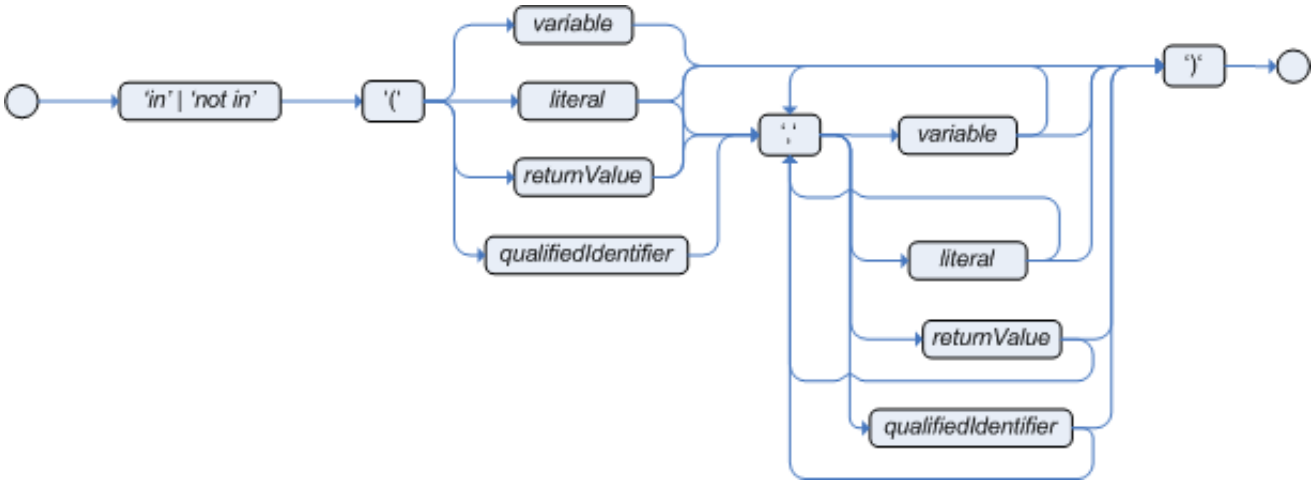

Figure 5.17. compoundValueRestriction

Example 5.41. Compound Restriction using "in"

```
Person( $cheese : favouriteCheese )
Cheese( type in ( "stilton", "cheddar", $cheese ) )
```

5.8.3.3.11. Inline eval operator (deprecated)

Figure 5.18. Inline Eval Expression

An inline eval constraint can use any valid dialect expression as long as it results to a primitive boolean. The expression must be constant over time. Any previously bound variable, from the current or previous pattern, can be used.

This example will find all male-female pairs where the male is 2 years older than the female; the variable `age` is auto-created in the second pattern by the autovivification process.

Example 5.42. Return Value operator

```
Person( girlAge : age, sex = "F" )
Person( eval( age == girlAge + 2 ), sex = 'M' ) // eval() is actually obselete in th
```

> **Note**
>
> Inline eval's are effectively obsolete as their inner syntax is now directly supported. It's recommended not to use them. Simply write the expression without wrapping eval() around it.

5.8.3.3.12. Operator precedence

The operators are evaluated in this precedence:

Table 5.1. Operator precedence

Operator type	Operators	Notes
(nested) property access	.	Not normal Java semantics
List/Map access	[ ]	Not normal Java semantics
constraint binding	:	Not normal Java semantics
multiplicative	* / %	
additive	+ -	
shift	<< >> >>>	

relational	`< > <= >=` `instanceof`	
equality	`== !=`	Does not use normal Java (*not*) *same* semantics: uses (*not*) *equals* semantics instead.
non-short circuiting AND	`&`	
non-short circuiting exclusive OR	`^`	
non-short circuiting inclusive OR	`|`	
logical AND	`&&`	
logical OR	`||`	
ternary	`? :`	
Comma separated AND	`,`	Not normal Java semantics

5.8.3.4. Positional Arguments

Patterns now support positional arguments on type declarations.

Positional arguments are ones where you don't need to specify the field name, as the position maps to a known named field. i.e. Person(name == "mark") can be rewritten as Person("mark";). The semicolon ';' is important so that the engine knows that everything before it is a positional argument. Otherwise we might assume it was a boolean expression, which is how it could be interpretted after the semicolon. You can mix positional and named arguments on a pattern by using the semicolon ';' to separate them. Any variables used in a positional that have not yet been bound will be bound to the field that maps to that position.

```
declare Cheese
    name : String
    shop : String
    price : int
end
```

Example patterns, with two constraints and a binding. Remember semicolon ';' is used to differentiate the positional section from the named argument section. Variables and literals and expressions using just literals are supported in posional arguments, but not variables. Positional arguments are always resolved using unification.

```
Cheese( "stilton", "Cheese Shop", p; )
Cheese( "stilton", "Cheese Shop"; p : price )
Cheese( "stilton"; shop == "Cheese Shop", p : price )
Cheese( name == "stilton"; shop == "Cheese Shop", p : price )
```

Positional arguments that are given a previously declared binding will consrain against that using unification; these are referred to as input arguments. If the binding does not yet exist, it will create the declaration binding it to the field represented by the position argument; these are referred to as output arguments.

5.8.3.5. Basic conditional elements

5.8.3.5.1. Conditional Element `and`

The Conditional Element `"and"` is used to group other Conditional Elements into a logical conjunction. Drools supports both prefix `and` and infix `and`.

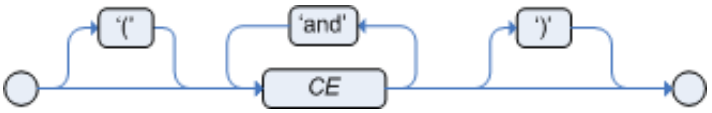

Figure 5.19. infixAnd

Traditional infix `and` is supported:

```
//infixAnd
Cheese( cheeseType : type ) and Person( favouriteCheese == cheeseType )
```

Explicit grouping with parentheses is also supported:

```
//infixAnd with grouping
( Cheese( cheeseType : type ) and
  ( Person( favouriteCheese == cheeseType ) or
    Person( favouriteCheese == cheeseType ) )
```

> **Note**
>
> The symbol `&&` (as an alternative to `and`) is deprecated. But it is still supported in the syntax for backwards compatibility.

Figure 5.20. prefixAnd

Prefix `and` is also supported:

```
(and Cheese( cheeseType : type )
     Person( favouriteCheese == cheeseType ) )
```

The root element of the LHS is an implicit prefix `and` and doesn't need to be specified:

Example 5.43. implicit root prefixAnd

```
when
    Cheese( cheeseType : type )
    Person( favouriteCheese == cheeseType )
then
```

```
    ...
```

5.8.3.5.2. Conditional Element `or`

The Conditional Element `or` is used to group other Conditional Elements into a logical disjunction. Drools supports both prefix `or` and infix `or`.

Figure 5.21. infixOr

Traditional infix `or` is supported:

```
//infixOr
Cheese( cheeseType : type ) or Person( favouriteCheese == cheeseType )
```

Explicit grouping with parentheses is also supported:

```
//infixOr with grouping
( Cheese( cheeseType : type ) or
  ( Person( favouriteCheese == cheeseType ) and
    Person( favouriteCheese == cheeseType ) )
```

> **Note**
>
> The symbol `||` (as an alternative to `or`) is deprecated. But it is still supported in the syntax for backwards compatibility.

Figure 5.22. prefixOr

Prefix `or` is also supported:

```
(or Person( sex == "f", age > 60 )
    Person( sex == "m", age > 65 )
```

> **Note**
>
> The behavior of the Conditional Element `or` is different from the connective `||` for constraints and restrictions in field constraints. The engine actually has no understanding of the

Conditional Element `or`. Instead, via a number of different logic transformations, a rule with `or` is rewritten as a number of subrules. This process ultimately results in a rule that has a single `or` as the root node and one subrule for each of its CEs. Each subrule can activate and fire like any normal rule; there is no special behavior or interaction between these subrules. - This can be most confusing to new rule authors.

The Conditional Element `or` also allows for optional pattern binding. This means that each resulting subrule will bind its pattern to the pattern binding. Each pattern must be bound separately, using eponymous variables:

```
pensioner : ( Person( sex == "f", age > 60 ) or Person( sex == "m", age > 65 ) )
```

```
(or pensioner : Person( sex == "f", age > 60 )
    pensioner : Person( sex == "m", age > 65 ) )
```

Since the conditional element `or` results in multiple subrule generation, one for each possible logically outcome, the example above would result in the internal generation of two rules. These two rules work independently within the Working Memory, which means both can match, activate and fire - there is no shortcutting.

The best way to think of the conditional element `or` is as a shortcut for generating two or more similar rules. When you think of it that way, it's clear that for a single rule there could be multiple activations if two or more terms of the disjunction are true.

5.8.3.5.3. Conditional Element `not`

Figure 5.23. not

The CE `not` is first order logic's non-existential quantifier and checks for the non-existence of something in the Working Memory. Think of "not" as meaning "there must be none of...".

The keyword `not` may be followed by parentheses around the CEs that it applies to. In the simplest case of a single pattern (like below) you may optionally omit the parentheses.

Example 5.44. No Busses

```
not Bus()
```

Example 5.45. No red Busses

```
// Brackets are optional:
not Bus(color == "red")
```

```
// Brackets are optional:
not ( Bus(color == "red", number == 42) )
// "not" with nested infix and - two patterns,
// brackets are requires:
not ( Bus(color == "red") and
      Bus(color == "blue") )
```

5.8.3.5.4. Conditional Element `exists`

Figure 5.24. exists

The CE `exists` is first order logic's existential quantifier and checks for the existence of something in the Working Memory. Think of "exists" as meaning "there is at least one..". It is different from just having the pattern on its own, which is more like saying "for each one of...". If you use `exists` with a pattern, the rule will only activate at most once, regardless of how much data there is in working memory that matches the condition inside of the `exists` pattern. Since only the existence matters, no bindings will be established.

The keyword `exists` must be followed by parentheses around the CEs that it applies to. In the simplest case of a single pattern (like below) you may omit the parentheses.

Example 5.46. At least one Bus

```
exists Bus()
```

Example 5.47. At least one red Bus

```
exists Bus(color == "red")
// brackets are optional:
exists ( Bus(color == "red", number == 42) )
// "exists" with nested infix and,
// brackets are required:
exists ( Bus(color == "red") and
         Bus(color == "blue") )
```

5.8.3.6. Advanced conditional elements

5.8.3.6.1. Conditional Element `forall`

Figure 5.25. forall

The Conditional Element `forall` completes the First Order Logic support in Drools. The Conditional Element `forall` evaluates to true when all facts that match the first pattern match all the remaining patterns. Example:

```
rule "All English buses are red"
when
    forall( $bus : Bus( type == 'english')
                   Bus( this == $bus, color = 'red' ) )
then
    # all english buses are red
end
```

In the above rule, we "select" all Bus objects whose type is "english". Then, for each fact that matches this pattern we evaluate the following patterns and if they match, the forall CE will evaluate to true.

To state that all facts of a given type in the working memory must match a set of constraints, `forall` can be written with a single pattern for simplicity. Example:

Example 5.48. Single Pattern Forall

```
rule "All Buses are Red"
when
    forall( Bus( color == 'red' ) )
then
    # all asserted Bus facts are red
end
```

Another example shows multiple patterns inside the `forall`:

Example 5.49. Multi-Pattern Forall

```
rule "all employees have health and dental care programs"
when
    forall( $emp : Employee()
            HealthCare( employee == $emp )
            DentalCare( employee == $emp )
          )
then
    # all employees have health and dental care
end
```

Forall can be nested inside other CEs. For instance, `forall` can be used inside a `not` CE. Note that only single patterns have optional parentheses, so that with a nested `forall` parentheses must be used:

Example 5.50. Combining Forall with Not CE

```
rule "not all employees have health and dental care"
when
    not ( forall( $emp : Employee()
                  HealthCare( employee == $emp )
                  DentalCare( employee == $emp ) )
        )
then
    # not all employees have health and dental care
end
```

As a side note, `forall( p1 p2 p3...)` is equivalent to writing:

```
not(p1 and not(and p2 p3...))
```

Also, it is important to note that `forall` is a *scope delimiter*. Therefore, it can use any previously bound variable, but no variable bound inside it will be available for use outside of it.

5.8.3.6.2. Conditional Element `from`

Figure 5.26. from

The Conditional Element `from` enables users to specify an arbitrary source for data to be matched by LHS patterns. This allows the engine to reason over data not in the Working Memory. The data source could be a sub-field on a bound variable or the results of a method call. It is a powerful construction that allows out of the box integration with other application components and frameworks. One common example is the integration with data retrieved on-demand from databases using hibernate named queries.

The expression used to define the object source is any expression that follows regular MVEL syntax. Therefore, it allows you to easily use object property navigation, execute method calls and access maps and collections elements.

Here is a simple example of reasoning and binding on another pattern sub-field:

```
rule "validate zipcode"
when
    Person( $personAddress : address )
    Address( zipcode == "23920W") from $personAddress
then
    # zip code is ok
end
```

With all the flexibility from the new expressiveness in the Drools engine you can slice and dice this problem many ways. This is the same but shows how you can use a graph notation with the 'from':

```
rule "validate zipcode"
when
    $p : Person( )
    $a : Address( zipcode == "23920W") from $p.address
then
    # zip code is ok
end
```

Previous examples were evaluations using a single pattern. The CE `from` also support object sources that return a collection of objects. In that case, `from` will iterate over all objects in the collection and try to match each of them individually. For instance, if we want a rule that applies 10% discount to each item in an order, we could do:

```
rule "apply 10% discount to all items over US$ 100,00 in an order"
when
    $order : Order()
    $item  : OrderItem( value > 100 ) from $order.items
then
    # apply discount to $item
end
```

The above example will cause the rule to fire once for each item whose value is greater than 100 for each given order.

You must take caution, however, when using `from`, especially in conjunction with the `lock-on-active` rule attribute as it may produce unexpected results. Consider the example provided earlier, but now slightly modified as follows:

```
rule "Assign people in North Carolina (NC) to sales region 1"
ruleflow-group "test"
lock-on-active true
when
    $p : Person( )
    $a : Address( state == "NC") from $p.address
then
    modify ($p) {} #Assign person to sales region 1 in a modify block
end

rule "Apply a discount to people in the city of Raleigh"
ruleflow-group "test"
lock-on-active true
when
    $p : Person( )
    $a : Address( city == "Raleigh") from $p.address
then
    modify ($p) {} #Apply discount to person in a modify block
end
```

In the above example, persons in Raleigh, NC should be assigned to sales region 1 and receive a discount; i.e., you would expect both rules to activate and fire. Instead you will find that only the second rule fires.

If you were to turn on the audit log, you would also see that when the second rule fires, it deactivates the first rule. Since the rule attribute `lock-on-active` prevents a rule from creating new activations when a set of facts change, the first rule fails to reactivate. Though the set of facts have not changed, the use of `from` returns a new fact for all intents and purposes each time it is evaluated.

First, it's important to review why you would use the above pattern. You may have many rules across different rule-flow groups. When rules modify working memory and other rules downstream of your RuleFlow (in different rule-flow groups) need to be reevaluated, the use of `modify` is critical. You don't, however, want other rules in the same rule-flow group to place activations on one another recursively. In this case, the `no-loop` attribute is ineffective, as it would only prevent a rule from activating itself recursively. Hence, you resort to `lock-on-active`.

There are several ways to address this issue:

- Avoid the use of `from` when you can assert all facts into working memory or use nested object references in your constraint expressions (shown below).
- Place the variable assigned used in the modify block as the last sentence in your condition (LHS).
- Avoid the use of `lock-on-active` when you can explicitly manage how rules within the same rule-flow group place activations on one another (explained below).

The preferred solution is to minimize use of `from` when you can assert all your facts into working memory directly. In the example above, both the Person and Address instance can be asserted into working memory. In this case, because the graph is fairly simple, an even easier solution is to modify your rules as follows:

```
rule "Assign people in North Carolina (NC) to sales region 1"
ruleflow-group "test"
lock-on-active true
when
    $p : Person(address.state == "NC" )
then
    modify ($p) {} #Assign person to sales region 1 in a modify block
end

rule "Apply a discount to people in the city of Raleigh"
ruleflow-group "test"
lock-on-active true
when
    $p : Person(address.city == "Raleigh" )
then
    modify ($p) {} #Apply discount to person in a modify block
end
```

Now, you will find that both rules fire as expected. However, it is not always possible to access nested facts as above. Consider an example where a Person holds one or more Addresses and you wish to use an existential quantifier to match people with at least one address that meets certain conditions. In this case, you would have to resort to the use of `from` to reason over the collection.

There are several ways to use `from` to achieve this and not all of them exhibit an issue with the use of `lock-on-active`. For example, the following use of `from` causes both rules to fire as expected:

```
rule "Assign people in North Carolina (NC) to sales region 1"
ruleflow-group "test"
```

```
lock-on-active true
when
    $p : Person($addresses : addresses)
    exists (Address(state == "NC") from $addresses)
then
    modify ($p) {} #Assign person to sales region 1 in a modify block
end

rule "Apply a discount to people in the city of Raleigh"
ruleflow-group "test"
lock-on-active true
when
    $p : Person($addresses : addresses)
    exists (Address(city == "Raleigh") from $addresses)
then
    modify ($p) {} #Apply discount to person in a modify block
end
```

However, the following slightly different approach does exhibit the problem:

```
rule "Assign people in North Carolina (NC) to sales region 1"
ruleflow-group "test"
lock-on-active true
when
    $assessment : Assessment()
    $p : Person()
    $addresses : List() from $p.addresses
    exists (Address( state == "NC") from $addresses)
then
    modify ($assessment) {} #Modify assessment in a modify block
end

rule "Apply a discount to people in the city of Raleigh"
ruleflow-group "test"
lock-on-active true
when
    $assessment : Assessment()
    $p : Person()
    $addresses : List() from $p.addresses
    exists (Address( city == "Raleigh") from $addresses)
then
    modify ($assessment) {} #Modify assessment in a modify block
end
```

In the above example, the $addresses variable is returned from the use of `from`. The example also introduces a new object, assessment, to highlight one possible solution in this case. If the $assessment variable assigned in the condition (LHS) is moved to the last condition in each rule, both rules fire as expected.

Though the above examples demonstrate how to combine the use of `from` with `lock-on-active` where no loss of rule activations occurs, they carry the drawback of placing a dependency on the order of conditions on the LHS. In addition, the solutions present greater complexity for the rule author in terms of keeping track of which conditions may create issues.

A better alternative is to assert more facts into working memory. In this case, a person's addresses may be asserted into working memory and the use of `from` would not be necessary.

There are cases, however, where asserting all data into working memory is not practical and we need to find other solutions. Another option is to reevaluate the need for `lock-on-active`. An alternative to `lock-on-active` is to directly manage how rules within the same rule-flow group activate one another by including conditions in each rule that prevent rules from activating each other recursively when working memory is modified. For example, in the case above where a discount is applied to citizens of Raleigh, a condition may be added to the rule that checks whether the discount has already been applied. If so, the rule does not activate.

5.8.3.6.3. Conditional Element `collect`

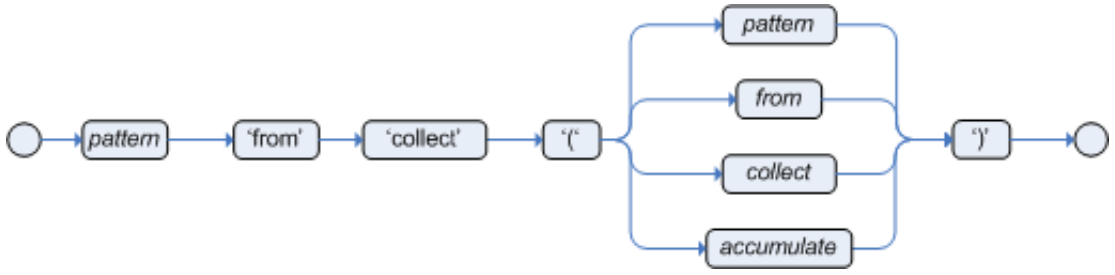

Figure 5.27. collect

The Conditional Element `collect` allows rules to reason over a collection of objects obtained from the given source or from the working memory. In First Oder Logic terms this is the cardinality quantifier. A simple example:

```
import java.util.ArrayList

rule "Raise priority if system has more than 3 pending alarms"
when
    $system : System()
    $alarms : ArrayList( size >= 3 )
              from collect( Alarm( system == $system, status == 'pending' ) )
then
    # Raise priority, because system $system has
    # 3 or more alarms pending. The pending alarms
    # are $alarms.
end
```

In the above example, the rule will look for all pending alarms in the working memory for each given system and group them in ArrayLists. If 3 or more alarms are found for a given system, the rule will fire.

The result pattern of `collect` can be any concrete class that implements the `java.util.Collection` interface and provides a default no-arg public constructor. This means that you can use Java collections like ArrayList, LinkedList, HashSet, etc., or your own class, as long as it implements the `java.util.Collection` interface and provide a default no-arg public constructor.

Both source and result patterns can be constrained as any other pattern.

Variables bound before the `collect` CE are in the scope of both source and result patterns and therefore you can use them to constrain both your source and result patterns. But note that `collect` is a scope

delimiter for bindings, so that any binding made inside of it is not available for use outside of it.

Collect accepts nested `from` CEs. The following example is a valid use of "collect":

```
import java.util.LinkedList;

rule "Send a message to all mothers"
when
    $town : Town( name == 'Paris' )
    $mothers : LinkedList()
               from collect( Person( gender == 'F', children > 0 )
                             from $town.getPeople()
                           )
then
    # send a message to all mothers
end
```

5.8.3.6.4. Conditional Element `accumulate`

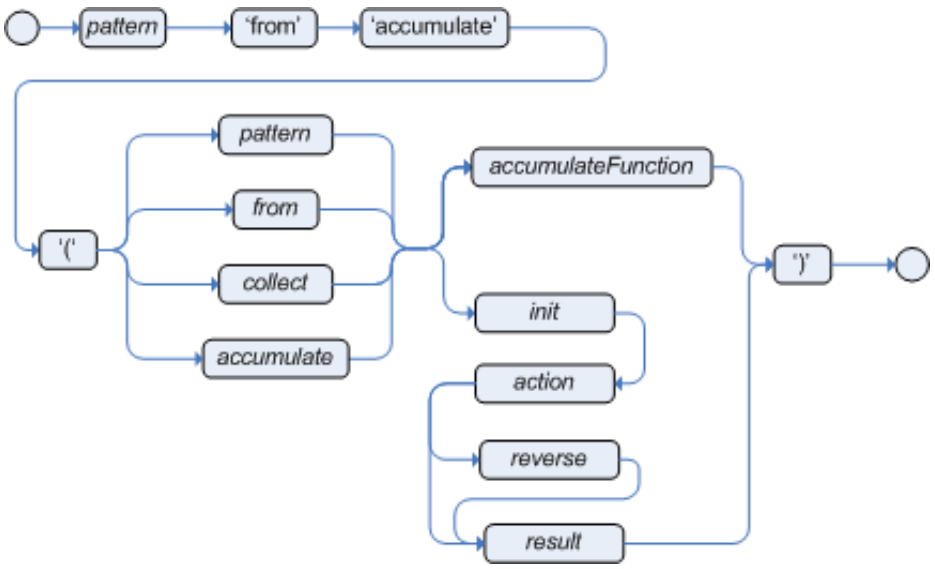

Figure 5.28. accumulate

The Conditional Element `accumulate` is a more flexible and powerful form of `collect`, in the sense that it can be used to do what `collect` does and also achieve results that the CE `collect` is not capable of doing. Basically, what it does is that it allows a rule to iterate over a collection of objects, executing custom actions for each of the elements, and at the end it returns a result object.

Accumulate supports both the use of pre-defined accumulate functions, or the use of inline custom code. Inline custom code should be avoided though, as it is extremely hard to maintain, and frequently leads to code duplication. Accumulate functions are easier to test and reuse.

5.8.3.6.4.1. Accumulate Functions

The accumulate CE is a very powerful CE, but it gets real declarative and easy to use when using predefined functions that are known as Accumulate Functions. They work exactly like accumulate, but

instead of explicitly writing custom code in every accumulate CE, the user can use predefined code for common operations.

For instance, a rule to apply a 10% discount on orders over $100 could be written in the following way, using Accumulate Functions:

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
             from accumulate( OrderItem( order == $order, $value : value ),
                              sum( $value ) )
then
    # apply discount to $order
end
```

In the above example, sum is an Accumulate Function and will sum the $value of all OrderItems and return the result.

Drools ships with the following built-in accumulate functions:

- average
- min
- max
- count
- sum
- collectList
- collectSet

These common functions accept any expression as input. For instance, if someone wants to calculate the average profit on all items of an order, a rule could be written using the average function:

```
rule "Average profit"
when
    $order : Order()
    $profit : Number()
              from accumulate( OrderItem( order == $order, $cost : cost, $price : pric
                               average( 1 - $cost / $price ) )
then
    # average profit for $order is $profit
end
```

Accumulate Functions are all pluggable. That means that if needed, custom, domain specific functions can easily be added to the engine and rules can start to use them without any restrictions. To implement a new Accumulate Functions all one needs to do is to create a Java class that implements the `org.drools.runtime.rule.TypedAccumulateFunction` interface and add a line to the configuration file or set a system property to let the engine know about the new function. As an example of an Accumulate Function implementation, the following is the implementation of the `average` function:

```
/**
 * An implementation of an accumulator capable of calculating average values
 */
```

```
public class AverageAccumulateFunction implements AccumulateFunction {

    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundExceptio

    }

    public void writeExternal(ObjectOutput out) throws IOException {

    }

    public static class AverageData implements Externalizable {
        public int    count = 0;
        public double total = 0;


        public AverageData() {}

        public void readExternal(ObjectInput in) throws IOException, ClassNotFoundExce
            count   = in.readInt();
            total   = in.readDouble();
        }

        public void writeExternal(ObjectOutput out) throws IOException {
            out.writeInt(count);
            out.writeDouble(total);
        }

    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#createContext()
     */
    public Serializable createContext() {
        return new AverageData();
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#init(java.lang.Object)
     */
    public void init(Serializable context) throws Exception {
        AverageData data = (AverageData) context;
        data.count = 0;
        data.total = 0;
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#accumulate(java.lang.Objec
     */
    public void accumulate(Serializable context,
                           Object value) {
        AverageData data = (AverageData) context;
        data.count++;
        data.total += ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
```

```
     * @see org.drools.base.accumulators.AccumulateFunction#reverse(java.lang.Object,
     */
    public void reverse(Serializable context,
                        Object value) throws Exception {
        AverageData data = (AverageData) context;
        data.count--;
        data.total -= ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#getResult(java.lang.Object
     */
    public Object getResult(Serializable context) throws Exception {
        AverageData data = (AverageData) context;
        return new Double( data.count == 0 ? 0 : data.total / data.count );
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#supportsReverse()
     */
    public boolean supportsReverse() {
        return true;
    }

    /**
     * {@inheritDoc}
     */
    public Class< ? > getResultType() {
        return Number.class;
    }

}
```

The code for the function is very simple, as we could expect, as all the "dirty" integration work is done by the engine. Finally, to plug the function into the engine, we added it to the configuration file:

```
drools.accumulate.function.average =
   org.drools.base.accumulators.AverageAccumulateFunction
```

Here, "drools.accumulate.function." is a prefix that must always be used, "average" is how the function will be used in the rule file, and "org.drools.base.accumulators.AverageAccumulateFunction" is the fully qualified name of the class that implements the function behavior.

5.8.3.6.4.2. Accumulate with inline custom code

> **Warning**
>
> The use of accumulate with inline custom code is not a good practice for several reasons, including dificulties on maintaining and testing rules that use them, as well as the inability of reusing that code. Implementing your own accumulate functions is very simple and

straightforward, they are easy to unit test and to use. This form of accumulate is supported for backward compatibility only.

The general syntax of the `accumulate` CE with inline custom code is:

```
<result pattern> from accumulate( <source pattern>,
                                  init( <init code> ),
                                  action( <action code> ),
                                  reverse( <reverse code> ),
                                  result( <result expression> ) )
```

The meaning of each of the elements is the following:

- *<source pattern>*: the source pattern is a regular pattern that the engine will try to match against each of the source objects.
- *<init code>*: this is a semantic block of code in the selected dialect that will be executed once for each tuple, before iterating over the source objects.
- *<action code>*: this is a semantic block of code in the selected dialect that will be executed for each of the source objects.
- *<reverse code>*: this is an optional semantic block of code in the selected dialect that if present will be executed for each source object that no longer matches the source pattern. The objective of this code block is to undo any calculation done in the *<action code>* block, so that the engine can do decremental calculation when a source object is modified or retracted, hugely improving performance of these operations.
- *<result expression>*: this is a semantic expression in the selected dialect that is executed after all source objects are iterated.
- *<result pattern>*: this is a regular pattern that the engine tries to match against the object returned from the *<result expression>*. If it matches, the `accumulate` conditional element evaluates to *true* and the engine proceeds with the evaluation of the next CE in the rule. If it does not matches, the `accumulate` CE evaluates to *false* and the engine stops evaluating CEs for that rule.

It is easier to understand if we look at an example:

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
             from accumulate( OrderItem( order == $order, $value : value ),
                              init( double total = 0; ),
                              action( total += $value; ),
                              reverse( total -= $value; ),
                              result( total ) )
then
    # apply discount to $order
end
```

In the above example, for each `Order` in the Working Memory, the engine will execute the *init code* initializing the total variable to zero. Then it will iterate over all `OrderItem` objects for that order, executing the *action* for each one (in the example, it will sum the value of all items into the total variable). After iterating over all `OrderItem` objects, it will return the value corresponding to the *result expression*

(in the above example, the value of variable `total`). Finally, the engine will try to match the result with the `Number` pattern, and if the double value is greater than 100, the rule will fire.

The example used Java as the semantic dialect, and as such, note that the usage of the semicolon as statement delimiter is mandatory in the init, action and reverse code blocks. The result is an expression and, as such, it does not admit ';'. If the user uses any other dialect, he must comply to that dialect's specific syntax.

As mentioned before, the *reverse code* is optional, but it is strongly recommended that the user writes it in order to benefit from the *improved performance on update and retract*.

The `accumulate` CE can be used to execute any action on source objects. The following example instantiates and populates a custom object:

```
rule "Accumulate using custom objects"
when
    $person   : Person( $likes : likes )
    $cheesery : Cheesery( totalAmount > 100 )
                from accumulate( $cheese : Cheese( type == $likes ),
                                 init( Cheesery cheesery = new Cheesery(); ),
                                 action( cheesery.addCheese( $cheese ); ),
                                 reverse( cheesery.removeCheese( $cheese ); ),
                                 result( cheesery ) );
then
    // do something
end
```

5.8.3.6.4.3. Multi-function Accumulates

The accumulate CE now supports multiple functions. For instance, if one needs to find the minimum, maximum and average value for the same set of data, instead of having to repeat the accumulate statement 3 times, a single accumulate can be used.

```
rule "Max, min and average"
    when
        accumulate( Cheese( $price : price ),
                    $max : max( $price ),
                    $min : min( $price ),
                    $avg : average( $price ) )
    then
        // do something
end
```

5.8.3.7. Conditional Element `eval`

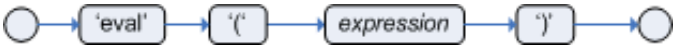

Figure 5.29. eval

The conditional element `eval` is essentially a catch-all which allows any semantic code (that returns a primitive boolean) to be executed. This code can refer to variables that were bound in the LHS of the rule, and functions in the rule package. Overuse of eval reduces the declarativeness of your rules and can result in a poorly performing engine. While `eval` can be used anywhere in the patterns, the best practice is to add it as the last conditional element in the LHS of a rule.

Evals cannot be indexed and thus are not as efficient as Field Constraints. However this makes them ideal for being used when functions return values that change over time, which is not allowed within Field Constraints.

For folks who are familiar with Drools 2.x lineage, the old Drools parameter and condition tags are equivalent to binding a variable to an appropriate type, and then using it in an eval node.

```
p1 : Parameter()
p2 : Parameter()
eval( p1.getList().containsKey( p2.getItem() ) )
```

```
p1 : Parameter()
p2 : Parameter()
// call function isValid in the LHS
eval( isValid( p1, p2 ) )
```

5.8.3.8. Railroad diagrams