# ABLE Rule Language
# User's Guide and Reference

Version 2.0

Last Update: 6/30/2003 2:34 PM

# Table of Contents

# Part 1. User's Guide

This manual is divided into three parts. Part 1, the User's Guide, presents an overview of the ABLE Rule Language and practical issues related to rule application development. Part 2, the Language Reference, presents details of the ABLE rule language syntax and technical specifications for ruleset authors. Part 3, the Programmer's Reference, deals with the Java programming APIs and integration issues of concern to Java programmers.

The User's Guide deals with practical usage aspects of the ABLE Rule Language (ARL) environment for developing applications. This includes a general introduction to the concepts of ARL, how it is similar and different from Java programming, and how to use the various capabilities of the ARL environment for specific types of applications.

## Chapter 1. Getting Started with ARL

The ABLE Rule Language (ARL) is a rule-based programming language that provides tight integration with Java objects, and the ability to externalize business logic using simple business rules or more complex inferencing rules. ABLE provides a set of rule engines ranging from light-weight procedural scripting to medium weight forward and backward chaining, up to the power and speed of advanced AI inferencing techniques. ARL also supports templates to allow customization of rulesets by non-technical users.

### Features of ARL

The ARL syntax is very Java-like, following the syntax and semantics of Java Object declarations and logical expressions. In Java you define classes with data members and methods. In ARL you define rulesets with variables, ruleblocks, and rules. In Java, methods contain one or more logical statements. In ARL ruleblocks contain one or more rules. ABLE Rule Language rulesets in text or XML documents are compiled into AbleRuleSet objects and are processed by the specified ABLE inference engines.

ABLE rulesets can be created using any text editor (or XML editor), but ABLE provides two rule authoring environments to assist with this task: the Swing-based RuleSet editor, and the Eclipse-based ARLEditor plugin. These editors allow you to compile your rulesets, and save them as serialized AbleRuleSet objects or as XML documents. The editors also provides test and debug facilities, so it is recommended you use one of the rule editors to develop your ARL rulesets.

While the ABLE rule language syntax is free-form, ABLE source rulesets have a definite structure. You first write ARL statements to specify Java classes you want to access during

inferencing, load domain-specific function libraries, define variables, and one or more rule blocks, which are collections of rules.

Depending on the inference engine used by the rule block, the rules may be processed sequentially or may be selected to fire based on priority, specificity, or some other criterion. You can write rules that invoke other rule blocks, allowing you to partition rules within a single ruleset and to employ multiple inferencing strategies. You can also invoke other rulesets from a rule, thereby allowing you to build complex rule-based applications using hierarchies of rulesets with separate rulesets as the building blocks. Your rules can also call out to arbitrary Java methods to receive values and invoke actions.

## Rules

A rule is a declarative statement or knowledge expression. The most common rule format is an if-then rule. However, in ARL all statements are referred to as rules, which can be roughly divided into type types, scripting and inference rules. Scripting rules include assertion (assignment) rules, if-then-else rules, for-loop rules, while-do, do-while, and do-until iteration rules. Inference rules include if-then rules, when-do pattern match rules, and predicate logic rules.

## Rule Blocks

Multiple rules can be grouped together into a rule block. Each rule block has an associated inference engine that interprets the rules in that block. This allows you to mix multiple inference techniques (for example, forward and backward chaining) with procedural scripts. Think of the rules as data. The inference engines implement the control strategies that affect how the rules are interpreted.

## RuleSets

An ABLE Rule Language ruleset contains data, ruleblocks and rule specifications. They can take the form of text files with Java-like syntax, of XML documents conforming to the ABLE ruleset XML schema definition, or of serialized AbleRuleSet JavaBeans.

## Inference Engines

An inference engine is a control algorithm that processes a group of rules (in a ruleblock). Each inference engine implements an algorithm for deriving a result given a set of input data and the corresponding rules in the rule block. ABLE provides a set of rule engines that can be used to process ARL rulesets and also can be extended via pluggable user-defined rule engines. The ABLE rule engines include:

**Script** (procedural), sequential evaluation of rules
**Backward** chaining, goal-chaining through if-then rules
**Predicate** backward chaining, goal-chaining with backtracking using predicate rules
**Fuzzy** forward chaining, multi-step chaining over fuzzy if-then rules with linguistic variables and hedges
**Forward** chaining, data-driven inferencing over if-then rules

**PatternMatch**, forward chaining with pattern matching - data-driven inferencing over when/do rules
**PatternMatchRete**, forward chaining with pattern matching using a Rete' network - data-driven inferencing over when/do rules

## Advantages of ARL

The ABLE Rule Language was designed with these attributes:

| | |
|---|---|
| Compiled | ABLE Rule Language rulesets are compiled into Java objects and then evaluated by the specified inference engine. While ARL can be used as a procedural scripting language, it is primarily intended to provide inferencing or evaluation of declarative rules. |
| Portable | ABLE rule language files can be parsed on any system with a Java virtual machine. The ruleset can be saved in either text or XML documents, or as a serialized AbleRuleSet bean. |
| Dynamic | You can use rules to manipulate the state of the ruleset and other rules. For example, each rule has an enabled flag that can be reasoned about and set by other rules. This can be used for enabling or disabling rules based on date and time or some other context. Each rule has a priority value that can be set by other rules. |
| Simple | The ARL syntax is similar to Java language syntax and should be familiar to C, C++, and Java programmers. While the syntax is largely the same, the semantics can be different due to the declarative nature of rules and inference engines. |
| Threading | An AbleRuleSet bean can be used as a standard Java object using the process() method to run on the caller's thread or it can be configured to run on it's own thread of control. |
| National Language Support (NLS) Enabled | An ABLE ruleset can have an associated external resource or message bundle. This allows user interface programs to use NLS-enabled text for template and variable prompts. |

## A Simple Example

The following ruleset shows an example of ARL syntax. The ruleset contains a global variable declaration block, the inputs and outputs specification, and two ruleblocks. The required process() ruleblock contains a set of if-then inference rules used with the backward chaining engine. The optional preProcess() ruleblock is used to set values on several variables prior to invocation of the process() ruleblock.

```
 ruleset FigureOutTheVehicle {
   variables {
      Categorical vehicleType = new Categorical(new String[]{ "automobile",
"cycle"});
      Discrete num_wheels = new Discrete(new Double[] { 2, 3, 4 });
      Discrete num_doors = new Discrete(new Double[] { 2, 3, 4 });
```

```
      Categorical motor = new Categorical(new String[] { "no" "yes" });
      Categorical vehicle = new Categorical(new String[] { "Bicycle", "Tricycle",
"Motorcycle", "MiniVan", "Sedan", "Sports Car", "Sports Utility Vehicle" });
      Categorical size = new Categorical(new String[]{"small","medium","large" });
   }

   inputs { }
   outputs { vehicle }


   void preProcess() using Script {
      a1: size = "medium";
      a2: num_wheels = 4;
      a3: num_doors = 3;
      a4: motor = "yes";
   }

   void process() using Backward {

    goal:  setControlOperator(ARL.Goal, "vehicle") ;

    Bicycle: if ( vehicleType == "cycle" and num_wheels == 2 and motor == "no")
             then vehicle = "Bicycle";
    Tricycle: if ( vehicleType == "cycle" and num_wheels == 3
                    and motor == "no")
              then vehicle = "Tricycle";
    Motorcycle: if ( vehicleType == "cycle" and num_wheels == 2
                    and motor == "yes" )
                then vehicle = "Motorcycle";
    SportsCar: if ( vehicleType == "automobile" and size == "small"
                    and num_doors == 2 )
               then vehicle = "Sports Car";
    Sedan: if ( vehicleType == "automobile" and size == "medium"
                    and num_doors == 4 )
           then vehicle = "Sedan";
    MiniVan: if ( vehicleType == "automobile" and size == "medium"
                    and num_doors == 3 )
             then vehicle = "MiniVan";
    SUV: if ( vehicleType == "automobile" and size == "large"
                    and num_doors == 4 )
         then vehicle = "Sports Utility Vehicle";
    Cycle: if ( num_wheels < 4 ) then vehicleType = "cycle";
    Automobile: if ( num_wheels == 4 and motor == "yes" )
                then vehicleType = "automobile";
   }
}
```

# How ARL Differs from Java

The ABLE Rule Language syntax is very much like standard Java syntax, but there are some conventions you must follow when writing ARL statements.

## RuleSet structure and environment

A ruleset is a set of ARL statements that are compiled into an AbleRuleSet bean and subsidiary com.ibm.able.rules objects. The ruleset contains specifications for importing external Java classes, loading external Java function libraries, declaring global variables, input and output variable lists, and rule blocks that contain one or more rules.

The rule blocks correspond to the major AbleBean methods, namely init(), process(), processAbleEvent(), processTimerEvent(), and quit(). Optional preProcess() and postProcess() ruleblocks can be used in conjunction with the process() block. An optional catch() block can be used to handle any exceptions thrown during rule evaluation. In addition, users can define and invoke their own rule blocks from other ruleblocks.

## NameSpaces

The ARL ruleset source is compiled in the context of an AbleRuleSet bean instance. There is no concept of a package associated with a RuleSet. However, the import and library statements use Java package definitions and class names. Data members and methods on imported classes can be referenced using standard Java dot "." notation. ABLE uses Java introspection to find data members and methods and match them to references in the ruleset. All of the introspection occurs during parsing and compilation of the ruleset. At run-time the data members and methods are invoked using the already resolved Method objects.

There are several disjoint namespaces used in an ARL ruleset. These include:

**Variables** – global variables have unique names, cannot be overridden
**Predicates** – list of predicates
**Functions** - list of functions (with arity) declared in functions{} block or via library statements.
**Data types** – list of known classes include built-in data types and imported types.

## Comments

ARL supports the standard Java syntax for single and multi-line comments. In addition, Javadoc style comments can be used before major components of a ruleset, including the ruleset, variables, ruleblocks, and rules.

## Built-in Data Types

ARL supports a set of built-in data types that map to the standard Java Object types. These include Boolean, Byte, Character, Double, Float, Integer, Long, Object, Short, String, and

TimeStamp (Calendar). Additional data types include Categorical (discrete set of String values), Discrete (discrete set of double values), Continuous (double with a min/max range), Fuzzy (a Fuzzy linguistic variable), and TimePeriod.

## User Data Types

Any valid Java class (accessible via the CLASSPATH) can be imported into an ABLE ruleset and referenced by rules. Variables of an imported type can be defined in the global variables section and instances of the class can be instantiated in rules. Data members and methods can be invoked on those Objects just as in a Java program.

## Objects

ARL supports Java objects in a first-class way. Any Java class can be imported, instantiated using available constructors and manipulated in an ARL ruleset. Data members and public methods can be accessed or invoked directly in antecedent or consequents of rules.

## Arrays

ARL provides support for Arrays of any of the built-in and imported data types. ARL does not directly support arrays of primitive data types, but they can be passed through ABLE rulesets in Object variables.

## Strings

ARL String support is essentially identical to Java. You can invoke java.String methods on ARL String variables and use the StringBuffer class to build up Strings.

## Operators

ARL supports almost all of the Java arithmetic, bitwise, and logical operators. Combination operators such as ++, --, +=, -=, /=, and *= are not supported.

## Statements vs Rules

In Java you have statements; in ARL you have rules. They are NOT the same. An ARL rule has a label, an optional priority, and an optional list of preCondition objects. ARL rules can contain multiple consequent or action clauses that may perform assignments, function calls, and method calls. However, ARL rules cannot contain embedded rules.

## Methods vs RuleBlocks

In Java you have methods on objects, in ARL you have rule blocks. They are NOT the same. A method defines a set of statements in a class that can access class (static) and instance variables and that can return an optional result value. An ARL ruleblock defines a set of rules in a ruleset that can access global variables (defined in the variables section) and that can return an optional result value. The ruleblock can specify which ABLE inference engine is to be used to interpret or process the rules. The engine most like a Java interpreter is the ABLE Script engine, in that the rules are evaluated in strict sequential order without regard to priority or rule type. Use of the other ABLE inference engines will give behavior that is very different from Java. That, of course, is the whole point of the ABLE rule language!

## Libraries

A library is a Java class with a single no-argument constructor (a JavaBean) and a set of public methods to be used in rules as user-defined functions. These methods could be static or not. There is a namespace limitation associated with user-defined functions. Only a single instance of a name and arity (number of arguments) can appear at one time in a ruleset.

## Predicates

ARL supports predicates and predicate rules. Predicate names must be declared in the predicates statement at the top of the ruleset. Predicates and predicate rules can then be used with the predicate inference engine to perform Prolog-like backchaining.

## Exceptions and Exception Handling

ARL provides limited support for exception handling. You can define a rule block to catch any run-time exceptions, you can instantiate an Exception object and you can throw those exception objects. Standard Java try/catch blocks are NOT supported.

## Miscellaneous Differences

ARL does not support arbitrary nesting of methods, fields, or rules. Some examples of illegal statements:

```
: System.out.println("?");  //Nested fields are not supported
: myObject.getName().toUpperCase(); // Nested methods are not supported
: if (true) {
      if (a > 0) println(a);
      else println(b);
  }  // Nested rules are not supported
```

Java supports many different data conversions automatically. ARL does not, usually in an attempt to avoid degrading performance. For example:

```
: x = Math.sin(1.0); // Math.sin(double x) is signature
: x = Math.sin(1d); // still passing a double
: x = Math.sin(1); // int not allowed in ARL
: x = Math.sin(1f); // float not allowed in ARL
```

# Chapter 2. Application Development

This chapter provides an overview of the application development process using the ABLE rule language and the AbleRuleSet bean.

## Design overview

Developing a rule-based application is similar to any modern programming application. The common elements include specification of external data, object design including specification of data members, data types and behavior based on methods. The novel aspect is the use of declarative rules and the various inferencing algorithms to supplement any procedural code required for the application.

The ABLE rule language was designed to naturally supplement the object-oriented programming paradigm of the Java language by introducing declarative rules processing using syntax and semantics that are similar to Java programs. Whether the goal is to simply externalize business or application logic using simple business rules for easy maintenance or to solve complex problems that require sophisticated inferencing, the ABLE rule language and rule environment can be used to provide a solution.

An AbleRuleSet bean and its associated ARL ruleset can be used for several different purposes. As an AbleBean, the ruleset can be used to define the behavior of the bean. This includes timer event processing logic when used as an autonomous agent, asynchronous event processing logic when used as a daemon or server process, and as rule-based transaction processing when used as externalized business logic. The basic lifecycle of an AbleRuleSet bean follows:

1. Create the AbleRuleSet bean instance (new AbleRuleSet()).
2. Parse an ARL text or XML source ruleset, or programmatically create a ruleset.
3. Initialize the bean by calling the init() method. If defined, process the init() ruleblock to perform one-time startup operations (enable/disable timer and event processing, set static variables, load working memory, etc.)

4a. If AbleRuleSet bean timer events are enabled and a processTimerEvent() ruleblock is defined, invoke that ruleblock every N milliseconds.

4b. If AbleRuleSet event posting and processing is enabled and a processAbleEvent() ruleblock is defined, invoke that ruleblock whenever incoming AbleEvents are received.

4c. Process transactions when the AbleRuleSet bean process() method is called, passing in an Object[] of input data and receiving an Object[] of output data. The process() methods support both AbleBean dataflow semantics and regular call/return semantics. Note that global variables are reset back to their initial values prior to each process() cycle. Only working memory and static variables retain their values across process() calls. The bean process() method can be called any number of times to process transactions.

5. Quit the AbleRuleSet bean processing by calling the quitAll() method. If defined, process the quitAll() ruleblock to do any end-of-life cleanup operations.

In the following sections, we present several common application scenarios using the ABLE rule language, including externalization of business logic, simple business rules, a dynamic grouping application, complex discount processing using rule templates, a staged direct marketing campaign, and a hierarchical policy-based application.

## Externalized Business Logic

Perhaps the most basic benefit of using rules in applications is that it forces you to separate the changeable business logic from the rest of the application logic. When used for this purpose, the rules are usually fairly simple if-then statements with the data passed in from the main application logic.

In the example shown below, the application needs to determine if the current Customer is a senior citizen or not. This is used to determine what discounts, if any, the customer qualifies for. The application program logic calls the AbleRuleSet bean's process() method, passing the Customer object as an input and receiving a Java Boolean result. This example illustrates the seemless use of imported application objects in rules. Notice that the Script engine is used, which is not an inference engine at all, but a basic rule-processing engine that processes the rules in strict sequential order, much like a Java program.

```
/**  An ABLE RuleSet showing simple business rules  */
ruleset SimpleBusinessRuleExample1 {
  import com.ibm.able.examples.rules.Customer ;

  variables {
      Customer customer ;
      Boolean  seniorCitizen = false ;
  }

  inputs  { customer };
  outputs { seniorCitizen };

  void process() using Script {
    SeniorTest: if (customer.age > 55) then seniorCitizen = true ;
  }
}
```

## Simple Business Rules

In the example shown below, the application needs to classify the current Customer as a gold, silver, or platinum. A built-in Categorical data type is used to represent these levels. This classification is then used to determine what services, promotions, or discounts the customer qualifies for. The application program logic calls the AbleRuleSet bean's process() method, passing the Customer object as an input and receiving a String value as the result. Once again, the Script engine is used to process the rules.

```
/**  An ABLE RuleSet showing simple business rules  */
ruleset SimpleBusinessRuleExample2 {
  import com.ibm.able.examples.rules.Customer ;
```

```
  variables {
      Customer customer ;
      Categorical  level = new Categorical(new String[] { "gold", "silver",
"platinum" }) ;
  }

  inputs{ customer };
  outputs{ level };

  void process() using Script {
    Gold:     if (customer.income > 10000) then level = "gold" ;
    Silver:   if (customer.income > 50000) then level = "silver";
    Platinum: if (customer.income > 100000) then level = "platinum";
  }
}
```

## Processing Collections

In the example shown below, the application needs to send an automated e-mail promotion to a segment of the customer base. We use a pattern match rule with a Selector to return a collection of customers whose age is greater than 18 and whose income is above $25,000 a year. When processed by the Script engine, each and every customer that meets this criteria is bound to the customer selector variable and processed by the actions in the do part of the when/do rule. The actions include setting a couple of Boolean flags and passing the customer object to a sendEmail() function for processing. Notice the use of the java.util.Vector class to hold the list of Customer objects for input and output. Also, an optional postProcess() ruleblock is used to clean up working memory after the process() completes.

```
/**  An ABLE RuleSet showing processing of collections  */
ruleset SimpleBusinessRuleExample4 {
  import com.ibm.able.examples.rules.Customer ;
  import java.util.Vector ;

  variables {
      Vector customers ;
   }

  inputs{ customers };
  outputs{ customers };

  void process() using Script {
     loadWm   : wm.assertAll(customers) ;
           // iterate over all customers who meet certain constraints
     Select : when (Customer customer ( customer.age > 18 and
                                        customer.income > 25000))
           do  {
                customer.couponXYZ = true ;
                customer.targetCampaignXYZ = true ;
                sendEmail(customer) ;  // send email content based on flag
               }
  }

  void postProcess() using Script {
           : wm.clear() ;  // remove all customers from working memory
  }
```

```
}
```

## Product Advisor

In the example show below, we use a set of if-then inference rules and the Backward chaining inference engine to implement a product advisor application. The basic scenario is to prompt a user for a list of product attributes and to recommend an appropriate product. In this simple example, we ask about the primary usage of a computer system, where it will be used, and how many users there are. The backward chaining process starts with the goal, basicTypes, and works backward through the if-then inference rules. When the engine comes across a variable whose value is not known, it prompts the user to provide a value.

In this example, we show the use of an external resource bundle for the prompt messages as well as variable prompts specified directly in the init() ruleblock. The AbleGUILib function library contains the dialog used to prompt the user. If no solution is found, a default answer is provided in the postProcess() ruleblock. This simple example illustrates how a whole class of expert system applications can be developed using ABLE rulesets.

```
/**
* This ruleset shows how to use ARL to do classic expert systems
* with user prompting for values during inferencing
*/
ruleset ComputerAdvisor {

library com.ibm.able.rules.AbleGUILib ;

variables {
    Categorical primaryUsage = new Categorical(new String[] { "games", "office",
"internet", "programming" }) ;
    Categorical primaryLocation = new Categorical(new String[] { "home", "work",
"travel" }) ;
    Discrete    numberOfUsers = new Discrete(new Double[] { 1, 5, 10, 50, 100 }) ;
    Categorical basicTypes = new Categorical(new String[] { "desktop", "notebook",
"server", "no recommendation" }) ;
    Categorical level = new Categorical(new String[] { "cheapo", "moderate", "top
of the line" }) ;
    String      recommendedType ;
    Boolean     isSingleUser  ;
}


  outputs { basicTypes } ;

 void init() {
     :
this.setResourceBundleName("com.ibm.able.examples.rules.SampleMessageBundle");
     : this.setVariablePrompt("numberOfUsers", "Please, sir, tell me the number of
users?");
     : this.setVariablePrompt("primaryUsage", "What is the primary use of this
computer?");
     : this.setVariablePrompt("primaryLocation", "Where will this computer be
used?");
  }

  void process() using Backward {
    : setControlParameter(ARL.Goal, "basicTypes") ;
```

```
    : if (numberOfUsers == 1) isSingleUser = true ;
    : if (numberOfUsers > 1) isSingleUser = false ;

    : if (isSingleUser == true and primaryLocation == "home")
      then basicTypes = "desktop" ;
    : if (isSingleUser == true and primaryLocation == "work")
      then basicTypes = "desktop" ;
    : if (isSingleUser == true and primaryLocation == "travel")
      then basicTypes = "notebook" ;


    : if (isSingleUser == false and numberOfUsers <= 5 and
         primaryLocation == "home")
      then basicTypes = "desktop" ;
    : if (numberOfUsers > 5 and primaryLocation == "home")
      then basicTypes = "server" ;
    : if (isSingleUser == false and primaryLocation == "work")
      then basicTypes = "server" ;
    : if (isSingleUser == false and primaryLocation == "travel")
      then basicTypes = "notebook" ;

  }

  void postProcess() {
     : if (basicTypes == null) then basicTypes = "no recommendation";
  }

}
```

## Complex Discount Rules

This ruleset shows the use of PatternMatch rules to compute discounts on multiple products and subject to multiple possible discounts. The ruleset takes an Order object as an input, which contains multiple OrderEntry objects. The OrderEntry objects are placed into working memory along with the Discounts. The PatternMatch inference engine matches the OrderEntry objects and Discounts and generates DiscountGroup objects which are also added to working memory. Inferencing proceeds in 3 phases, first computing all applicable discounts, then avoiding duplicates, until all rules that can fire have been processed. Finally, the postProcess() ruleblock selectes the best applicable discount, along with the discounted price.

```
/**  An ABLE RuleSet to compute discounts on a multi-item order ...  */
ruleset WCSDiscountRules {

import com.ibm.able.examples.rules.Order;
import com.ibm.able.examples.rules.OrderEntry;
import com.ibm.able.examples.rules.SKU;
import com.ibm.able.examples.rules.Discount;
import com.ibm.able.examples.rules.DiscountGroup;

  variables {
        SKU productA =  new SKU("A",100.0);
        SKU productB =  new SKU("B",100.0);
        SKU productC =  new SKU("C",100.0);
        Order order =  new Order();
        OrderEntry orderEntry1 =  new OrderEntry();
        OrderEntry orderEntry2 =  new OrderEntry();
        OrderEntry orderEntry3 =  new OrderEntry();
        Discount newDiscount =  new Discount();
        DiscountGroup newDiscountGroup =  new DiscountGroup();
        DiscountGroup bestGroup =  new DiscountGroup();
        String phase =  new String("0");
```

```
        Boolean falseValue =  new Boolean(false);
        Boolean trueValue =  new Boolean(true);
        Double totalPrice = 0.0;
        Double bestPrice = 0.0;
        Double bestDiscount = 0.0;
        OrderEntry item =  new OrderEntry();
        OrderEntry item2 =  new OrderEntry();
        Discount discount =  new Discount();
        DiscountGroup discountGroup =  new DiscountGroup();
  }

  inputs{order};
  outputs{totalPrice, bestDiscount, bestPrice};


  void preProcess() using Script {
    : wm.assert(productA);
    : wm.assert(productB);
    : wm.assert(productC);
    : wm.assert(phase);
    : wm.assert(order);
  }


  void process() using PatternMatch {
    : order.assertOrderEntries(wm);
    : phase = "0";
    : wm.assert(phase);
    : wm.assert(productA);
    : wm.assert(productB);
    : wm.assert(productC);
    : wm.assert(order);
    Rule_Discount_1[4]:
      when( String phase   ( phase == "0" )  &
            OrderEntry item   ( (item.orderId == order.orderId) &&
(item.productName == "A") )  &
            Discount discount ! ( discount.name == "A" )
          )
      do  { newDiscount =  new Discount();
            newDiscount.name = "A";
            newDiscount.type = "free-merchandise";
            newDiscount.factor1 = 1;
            newDiscount.applyDiscount(item);
            wm.assert(newDiscount);
            newDiscountGroup =  new DiscountGroup();
            newDiscountGroup.addDiscount(newDiscount);
            wm.assert(newDiscountGroup);
          }

    Rule_Discount_3[4]:
      when( String phase   ( phase == "0" )  &
            OrderEntry item   ( (item.orderId == order.orderId) &&
(item.productName == "C") )  &
            Discount discount ! ( discount.name == "C" )
          )
      do  { newDiscount =  new Discount();
            newDiscount.name = "C";
            newDiscount.type = "percentage-off";
            newDiscount.factor1 = 0.1;
            newDiscount.applyDiscount(item);
            wm.assert(newDiscount);
            newDiscountGroup =  new DiscountGroup();
            newDiscountGroup.addDiscount(newDiscount);
            wm.assert(newDiscountGroup);
          }

    Rule_Discount_2[4]:
      when( String phase   ( phase == "0" )  &
            OrderEntry item   ( (item.orderId == order.orderId) &&
(item.productName == "A") )  &
            OrderEntry item2   ( (item2.orderId == order.orderId) &&
(item2.productName == "B") )  &
```

```
            Discount discount ! ( discount.name == "A+B" )
          )
      do  { newDiscount =  new Discount();
            newDiscount.name = "A+B";
            newDiscount.type = "percentage-off";
            newDiscount.factor1 = 0.05;
            newDiscount.applyDiscount(item,item2);
            wm.assert(newDiscount);
            newDiscountGroup =  new DiscountGroup();
            newDiscountGroup.addDiscount(newDiscount);
            wm.assert(newDiscountGroup);
          }

    Rule_Discount_4[4]:
      when( String phase   ( phase == "0" )  &
            OrderEntry item   ( (item.orderId == order.orderId) &&
(item.productName == "B") )  &
            OrderEntry item2   ( (item2.orderId == order.orderId) &&
(item2.productName == "C") )  &
            Discount discount ! ( discount.name == "B+C" )
          )
      do  { newDiscount =  new Discount();
            newDiscount.name = "B+C";
            newDiscount.type = "fixed-amount";
            newDiscount.factor1 = 10;
            newDiscount.applyDiscount(item,item2);
            wm.assert(newDiscount);
            newDiscountGroup =  new DiscountGroup();
            newDiscountGroup.addDiscount(newDiscount);
            wm.assert(newDiscountGroup);
          }

    Rule_StartPhase1[1.0]:
      when( String phase   ( phase == "0" )
          )
      do  { phase = "1";
            wm.modify(phase);
          }

    Rule_6[4]:
      when( String phase   ( phase == "1" )  &
            Discount discount   ( discount.discountValue > 0 )  &
            DiscountGroup discountGroup   ( (falseValue ==
discountGroup.containsDiscount(discount)) && (falseValue ==
discountGroup.containsItems(discount)) )
          )
      do  { discountGroup.addDiscount(discount);
          }

    Rule_StartPhase2[1.0]:
      when( String phase   ( phase == "1" )
          )
      do  { phase = "2";
            wm.modify(phase);
          }

  }


  void postProcess() using Script {
    R1: bestDiscount = order.bestDiscount(wm);
    R1a: order.bestDiscount = bestDiscount;
    R2: order.computeTotalPrice();
    R3: totalPrice = order.totalPrice;
    R4: bestPrice = order.bestPrice;
  }

}
```

## Policy-based Management

This ruleset shows the use of the TimePeriod datatype and rule preconditions to enable and disable rules based on the day of the week and the time of day. The ruleset expects a single input, the current date and time represented by a TimeStamp (Java Calendar) object. In the preprocess() rule block, the checkTimePeriodPreConditions(TimeStamp) function is called to enable or disable the rules based on the input date and time. The three rules in the process() ruleblock, are enabled based on the specified TimePeriod preconditions.

```
ruleset PolicyExample {

library com.ibm.able.beans.rules.AbleCalendarLib;

  variables {
    TimeStamp simulatedTime ;
    String simulatedTimeString ;

    Integer weekendPolicyActive = 0;
    Integer weekdayDayTimePolicyActive = 0;
    Integer weekdayEveningPolicyActive = 0;

    static TimeStamp start =  new TimeStamp("01/01/2002 12:00 AM");
    static TimeStamp end =  new TimeStamp("12/31/2002 12:00 AM");

    Integer weekendMask = ARL.SATURDAY + ARL.SUNDAY;
    Integer weekdayMask = ARL.MONDAY + ARL.TUESDAY + ARL.WEDNESDAY + ARL.THURSDAY +
ARL.FRIDAY;

    TimePeriod weekend =  new TimePeriod(start,end, 0xFFF0,weekendMask);
    TimePeriod weekdayDayTime =  new TimePeriod(start,end,
0xFFF0,weekdayMask,"06:00:00 AM/06:00:00 PM");
    TimePeriod weekdayEvening =  new TimePeriod(start,end,
0xFFF0,weekdayMask,"06:00:01 PM/11:59:00 PM");
  }

  inputs{simulatedTime};
  outputs{simulatedTimeString, weekendPolicyActive, weekdayDayTimePolicyActive,
weekdayEveningPolicyActive};

  void preProcess()  using Script  {
    : checkTimePeriodPreConditions(simulatedTime);
    : simulatedTimeString = formatDateAndTime(simulatedTime) ;
    : println(simulatedTimeString) ;
  }

  void process()  using Script  {
    WeekendRule {weekend} : invokeRuleBlock("weekendPolicy");
    WeekDayDayRule {weekdayDayTime} : invokeRuleBlock("weekdayDayTimePolicy");
    WeekDayNightRule {weekdayEvening} : invokeRuleBlock("weekdayEveningPolicy");
  }

  void weekendPolicy()  using Script  {
    : weekendPolicyActive = 1;
    : println("It is weekend!" ) ;
  }

  void weekdayDayTimePolicy()  using Script  {
    : weekdayDayTimePolicyActive = 1;
    : println("It is weekday  - day time ");
  }

  void weekdayEveningPolicy()  using Script  {
    : weekdayEveningPolicyActive = 1;
    : println("It is weekday - evening " ) ;
  }
```

```
}
```

## Rule Templates

This example shows the use of rule templates to allow generation of discount rules by a business manager. This ruleset shows the use of several template variables including one for rule priority, one for the product, one for customer age and nationality, and the discount percentage. The template rule, with label “discountTemplate” shows the basic structure of the rule, with the template replacements variables. This ruleset can be used behind a web-based user interface to allow generation of new discount rules.

```
/**  An ABLE RuleSet that shows use of rule templates
*    for discount rules
*/
ruleset DiscountTemplateExample {
  import com.ibm.able.data.AbleStringLiteral ;
  import com.ibm.able.examples.rules.Customer ;
  import com.ibm.able.examples.rules.ShoppingCart;

  variables {
    Customer customer = new Customer("joe", 23, "American") ;
    String[] items = new String[] { "Fruit", "Cigars", "Wine", "Beer", "Cheese"};
    ShoppingCart shoppingCart = new ShoppingCart("joe", items) ;
    Double   discount ;

    // template variables are not part of run-time ruleset !!!!
    template Double priority ;
    template Categorical Item = new Categorical(new String[] { "Fruit", "Cigars",
"Wine", "Beer", "Cheese" });
    template Integer Age = 21 ;
    template Categorical Nationality = new Categorical(new String[] { "American",
"Italian", "French", "German", "Japanese"}) ;
    template Discrete Percentage = new Discrete(new Double[] { 0, 5, 10, 15, 20,
25, 30, 35, 40, 45, 50 });
  }

  inputs{};
  outputs{ discount};

 void preProcess() {
    : customer.age = 21 ;
    : customer.nationality = "French" ;
 }

 void process() using Forward {

   /** example discount rule per Arvind's template doc */
  discountExample :
    if (shoppingCart.contains("Cigars") and
       customer.age > 18 and
       customer.nationality == "French")
    then  discount = 10 ;

 // example data = Wine, 21, Italian, 5%
 /** A sample rule template, with custom priority also allowed */
 template discountTemplate [ priority ]  :
    if (shoppingCart.contains(Item) and
       customer.age > Age and
       customer.nationality == Nationality)
    then discount = Percentage ;
 }
```

```
}
```

## Fuzzy Mortgage Calculator

This example ruleset uses Fuzzy variables and the Fuzzy inference engine to compute whether a mortgage should be approved based on attributes of the applicant.

```
ruleset MortgageUnderwriter {

  variables {
    Fuzzy Income = new Fuzzy(0, 10000)  {
      Linear    HIGH   = new Linear   (4000, 10000, ARL.Up    );
      Trapezoid MEDIUM = new Trapezoid(1000,  3000, 7000, 9000);
      Linear    LOW    = new Linear   (   0,  4000, ARL.Down  );
    };

    Fuzzy YearsEmployed = new Fuzzy(0, 30)  {
      Segments MORETHAN2 = new  Segments(2, 0,  2.1, 1,  30, 1);
      Segments LESSTHAN2 = new ~Segments(MORETHAN2);
    };

    Fuzzy Savings = new Fuzzy(0, 10000)  {
      Linear    HIGH   = new Linear   (4000, 10000, ARL.Up  );
      Pi        MEDIUM = new Pi       (5000,  4000          );
      Linear    LOW    = new Linear   (   0,  4000, ARL.Down);
    };

    Fuzzy AppraisedValue = new Fuzzy(0, 1000000)  {
      Sigmoid   HIGH   = new Sigmoid  (120000, 200000, 300000, ARL.Up);
      Trapezoid MEDIUM = new Trapezoid( 40000,  85000, 120000, 300000);
      Linear    LOW    = new Linear   (     0,  85000, ARL.Down      );
    };

    Categorical Location = new Categorical(new String[] {"RURAL", "SUBURB",
"URBAN"});

    Fuzzy StructureValue = new Fuzzy(0, 1000000)  {
      Linear    Unused = new Linear   (0.0, 1.0, ARL.Up);
    };

    Fuzzy LoanAmount = new Fuzzy(0, 1000000)  {
      Linear    HIGH   = new Linear   (120000, 200000, ARL.Up        );
      Trapezoid MEDIUM = new Trapezoid( 40000,  85000, 120000, 200000);
      Linear    LOW    = new Linear   (     0,  85000, ARL.Down      );
    };

    Fuzzy MonthlyPayment = new Fuzzy(0, 3000)  {
      Linear    Unused = new Linear   (0.0, 1.0, ARL.Up);
    };

    Fuzzy LoanToValueRatio = new Fuzzy(0.0, 1.0)  {
      Linear    HIGH = new Linear   (0.85, 0.851, ARL.Up);
    };

    Fuzzy PaymentToIncomeRatio = new Fuzzy(0.0, 1.0)  {
      Linear    LOW       = new  Linear(0.25, 0.35, ARL.Down);
      Linear    HIGH      = new ~Linear(LOW);
      Linear    EXCESSIVE = new  Linear(0.35, 0.45, ARL.Up);
    };

    Fuzzy StructureToValueRatio = new Fuzzy(0.0, 1.0)  {
      Linear    LOWRURAL = new Linear (0.4 , 0.6 ,       ARL.Down);
      Sigmoid   LOW      = new Sigmoid(0.65, 0.75, 0.85, ARL.Down);
    };
```

```
    Fuzzy CreditScore = new Fuzzy(0.0, 1.0)  {
      Linear    POOR = new Linear   (0.8, 0.801, ARL.Down);
    };

    Categorical Response        = new Categorical(new String[] {"Approved",
"Rejected"});

    Categorical IncomeCheck     = new Categorical(new String[] {"OK",
"InsufficientIncome"});
    Categorical EmploymentCheck = new Categorical(new String[] {"OK",
"UnstableEmployment"});
    Categorical CreditCheck     = new Categorical(new String[] {"OK",
"PoorCredit"});
    Categorical PropertyCheck   = new Categorical(new String[] {"OK",
"LowPropertyValue"});
    Categorical StructureCheck  = new Categorical(new String[] {"OK",
"LowStructureValue"});

    //-------------------------------------------------------
    // Fields in the input buffer for supervised learning by
    // neural networks; not needed for rule processing but can
    // be used to confirm rules arrive at expected conclusion
    //-------------------------------------------------------
    Categorical HumanApproved           = new Categorical(new String[] {"YES",
"NO"});
    Categorical HumanInsufficientIncome = new Categorical(new String[] {"YES",
"NO"});
    Categorical HumanUnstableEmployment = new Categorical(new String[] {"YES",
"NO"});
    Categorical HumanPoorCredit         = new Categorical(new String[] {"YES",
"NO"});
    Categorical HumanLowPropertyValue   = new Categorical(new String[] {"YES",
"NO"});
    Categorical HumanLowStructureValue  = new Categorical(new String[] {"YES",
"NO"});

  }


  // Use the following inputs{} statement if variable values are
  // provided by assertions in the "preProcess" ruleblock below.
  inputs {};

  // Use the following inputs{} statement if variable values are
  // to come from the ruleset's inputBuffer (which must be
  // primed by some external process before the ruleset is
  // invoked).
  /* inputs {Income, YearsEmployed, Savings, AppraisedValue, Location,
            StructureValue, LoanAmount, MonthlyPayment, LoanToValueRatio,
            PaymentToIncomeRatio, StructureToValueRatio, CreditScore,
            HumanApproved, HumanInsufficientIncome, HumanUnstableEmployment,
            HumanPoorCredit, HumanLowPropertyValue, HumanLowStructureValue};
  */

  // Values to place into the output buffer when inferencing is complete.
  outputs {Response, IncomeCheck, EmploymentCheck, CreditCheck, PropertyCheck,
StructureCheck};


  void preProcess() using Fuzzy {
    //-------------------------------------------------------------------
    // Optionally initialize the InputVariables here for
    // running the ruleset within the ruleset editor.
    // Delete these values if the InputBuffer is to provide
    // values!
    //
    // For each pair of statements below, the first one shown
    // should be OK; the commented second statement should cause
    // the check to fail.
    //-------------------------------------------------------------------
      : Income = 945;
```

```
      : YearsEmployed = 5;
      : Savings = 10000;
      : AppraisedValue = 30000;
      : Location = "URBAN";

      : StructureValue = 27000;
      : LoanAmount = 20000;
      : MonthlyPayment = 315;

      : CreditScore = 0.9;
   }

 void init() {
     : setControlParameter(ARL.process, ARL.InferenceMethod,   ARL.FuzzyAdd); // {
MinMax | FuzzyAdd | ProductOr }
    : setControlParameter(ARL.process, ARL.CorrelationMethod, ARL.Product);  // {
Product | Minimum }
    : setControlParameter(ARL.process, ARL.DefuzzifyMethod,   ARL.Centroid); // {
Centroid | MaxHeight }
    : setControlParameter(ARL.process, ARL.AlphaCut, 0.10);                  // 0.0
< n < 1.0
}

 void process() using Fuzzy {


    : Response        = "Rejected";
    : IncomeCheck     = "OK";
    : EmploymentCheck = "OK";
    : CreditCheck     = "OK";
    : PropertyCheck   = "OK";
    : StructureCheck  = "OK";

    R1: LoanToValueRatio      =  LoanAmount / AppraisedValue;
    R2: PaymentToIncomeRatio  = (MonthlyPayment / Income);
    R3: StructureToValueRatio = (StructureValue / AppraisedValue);

    I1: if (PaymentToIncomeRatio is EXCESSIVE)
      then  IncomeCheck = "InsufficientIncome";

    I2: if (PaymentToIncomeRatio is HIGH and Savings is not HIGH)
      then  IncomeCheck = "InsufficientIncome";

    E1: if (YearsEmployed is LESSTHAN2)
      then  EmploymentCheck = "UnstableEmployment";

    C1: if (CreditScore is POOR)
      then  CreditCheck = "PoorCredit";

    P1: if (LoanToValueRatio is HIGH)
      then  PropertyCheck = "LowPropertyValue";

    S1: if (Location == "RURAL" and StructureToValueRatio is LOWRURAL)
      then  StructureCheck = "LowStructureValue";

    S2: if (Location == "URBAN" and StructureToValueRatio is LOW)
      then  StructureCheck = "LowStructureValue";

    S3: if (Location == "SUBURB" and StructureToValueRatio is LOW)
      then  StructureCheck = "LowStructureValue";

    Result: if (IncomeCheck     == "OK"  and
                EmploymentCheck == "OK"  and
                CreditCheck     == "OK"  and
                PropertyCheck   == "OK"  and
                StructureCheck  == "OK")
      then  Response = "Approved";
  }

}
```

## Predicate Eight Queens

In addition to boolean inferencing using if-then rules, the ABLE Rule Language supports backchaining with backtracking using Predicates. In this example, we show a solution to the classic Eight Queens problem using an ABLE ruleset.

```
/**
*  Compute the solution(s) to the classic Eight Queens chess problem
*/
ruleset EightQueens {
  // Bratko, Ivan.  Prolog Programming for Artificial Intelligence,
  //  Addison-Wesley, 1986, p 108.

  library com.ibm.able.rules.AblePredicateLib;

  predicates {solution, permutation, del, safe, noattack, define, add};

  variables{
     Object result;
  }
  outputs{result};

  void process()  using Predicate {

    :  setControlParameter(ARL.Goal, solution(Pos)) ;  // find all solutions

      // entire 8 queens problem
 // : solution (Queens) :- permutation( [ 1,2,3,4,5,6,7,8], Queens), safe(Queens).

      // abreviated 5 queens problem
    : solution(Queens) :- permutation([1,2,3,4,5],Queens), safe(Queens).

    : permutation([],[]).
    : permutation([Head|Tail],PermList) :-
          permutation(Tail,PermTail),
          del(Head,PermList,PermTail).

    : del(A,[A|List1],List1).
    : del(A,[B|List1],[B|List2]) :-
          del(A,List1,List2).

    : safe([]).
    : safe([Queen|Others]) :-
          safe(Others),
          noattack(Queen,Others,1).

    : noattack(_,[],_).
    : noattack(Y,[Y1|Ylist],Xdist) :-
          (Xdist != minus(Y1,Y)),
          (Xdist != minus(Y,Y1)),
          add(Xdist,1,Dist1),
          noattack(Y,Ylist,Dist1).

    : add(X,Y,Z) :-
          Z = plus(X,Y).

  }

  void postProcess() {
    : result = getSolutionList(this, "process");
  }

}
```

## Hierarchical Rulesets

You can create a hierarchy of rulesets by invoking sub-rulesets from rules in an ARL ruleset. There are several ways to do this, depending on whether the sub-ruleset is self-contained, or whether you have to pass input data to it, or even if you want to share the ruleset context (variable bindings, user-defined functions, and working memory) with the sub-ruleset.

```
/**    An ABLE RuleSet...    */
ruleset HighLevelPolicy {

import com.ibm.able.rules.AbleRuleSet;

  variables {
        AbleRuleSet RuleSet3;
        AbleRuleSet RuleSet2;
        AbleRuleSet RuleSet1;
        String Action =  new String("do_nothing");
        Categorical Status = new Categorical(new String[] {"Green", "Yellow",
"Red"});
        Categorical SystemLoad = new Categorical(new String[] {"Low", "Medium",
"High"});
    Object[] results =  new Object();
  }

  inputs{Status, SystemLoad};
  outputs{Action};


  void process() using Script {
      : println("HighLevel status = " + Status);
      : println("HighLevel systemLoad = " + SystemLoad);
    A1: RuleSet1 = parent.getBean("MidLevel1");
    A2: RuleSet2 = parent.getBean("MidLevel2");
    A3: RuleSet3 = parent.getBean("MidLevel3");
      : RuleSet1.setDataFlowEnabled(false);
      : RuleSet2.setDataFlowEnabled(false);
      : RuleSet3.setDataFlowEnabled(false);
    R1:
      if (Status == "Green")
      then {
      // process sub-ruleset, without context, returns first outputbuffer element
          Action = ARL.processRuleSet(RuleSet1, new Object[0]);
      }

    R2:
      if (Status == "Yellow")
      then {
      // process sub-ruleset and share this ruleset context, returns outputBuffer
          results = RuleSet2.processWithContext(new Object[0], variableList,
functionList, wm);
          Action = results[0];
      }

    R3:
      if (Status == "Red")
      then {
          // process sub-ruleset, without context, returns outputBuffer
          results = RuleSet3.process(inputBuffer);
          Action = results[0];
      }
  }

}
```

## Event Processing

This example shows a ruleset sending as well as processing AbleEvents. The process() ruleblock instantiates a new AbleEvent object and sends it to any registered listeners. If any AbleEvents are received by this ruleset bean, they get forwarded to the processAbleEvent() ruleblock and handled by the rule logic there. Note that no data is passed in the input or output buffer. The events are sent directly to other AbleBeans and processed directly by the processAbleEvent() ruleblock via the built-in **event** variable.

```
/** Send an AbleEvent with process() action and an Object[] to a filter bean... */
ruleset SendProcessEventDemo {

  import com.ibm.able.AbleEvent ;

  variables {
    String[] outBufData = new String[] { "apples", "5", "2.3" };
    Object[] argObjData = new Object[3] ;
    AbleEvent processEvent ;
  }

  inputs{};
  outputs{};

  void process() using Script {
         // create a "process" AbleEvent with returnObject=this,
returnAction="filterOutput", trans Id = 001
       : processEvent = new AbleEvent(this, outBufData, "process", true, this,
"filterOutput", "001") ;
       : notifyAbleEventListeners(processEvent) ;
  }


  void processAbleEvent() using Script {
      : println("received an Able event ") ;
      : println(event.getAction()) ;
      : argObjData = event.getArgObject() ;
      : println(argObjData[0]) ;

  }

}
```

## Tasks

This section describes several common tasks associated with the use of ABLE rules for application development.

### Writing a RuleSet

An ABLE RuleSet can be written using the text or XML editor of your choice, the ABLE Swing-based RuleSet editor, or the ARL Editor provided with the ABLE WSAD plugin.

Designing a ruleset is similar to typical program design. You must define your data, variables and data types. If you will be using the ruleset in a transaction-oriented mode, you

must define your inputs and outputs. You must also decide which rule engines are required and which rule types are appropriate

## Deploying a RuleSet

An ABLE ruleset can be deployed by deserializing the RuleSet bean. A single static method is used to deserialize any AbleBean as shown below.

```
AbleRuleSet ruleSet = (AbleRuleSet)
         AbleObject.restoreFromSerializedFile("C:\able\rules\Sample.ser");
```

An ABLE ruleset can also be recreated by saving the text ARL file or the XML ARML file and parsing it during run-time. The steps include first creating an AbleRuleSet object, parsing the source file, and the initializing the AbleRuleSet bean.

```
      AbleRuleSet ruleSet = new AbleRuleSet("Sample");
      ruleSet.parseFromARL("C:\able\rules\Sample.arl");
      ruleSet.init();
```

## Maintaining a RuleSet

An ABLE ruleset can be maintained in one of three ways. As a source text file, as a source XML document, or as a serialized AbleRuleSet bean. These three formats are interchangeable and rulesets can be easily transformed between formats. For example, you could author the ruleset using the ARL text rule editor, parse and load it into an AbleRuleSet bean, and then save it as an XML document.

Note: ABLE does not provide rule file or document management functions.

## Parsing

An ARL ruleset can be parsed in the stand-alone Swing editor using the Verify option. In the WSAD ARLEditor the ruleset will be parsed when you save the file. Programmatically, you can instantiate an AbleRuleSet bean and then call the parseFromARL() or parseFromXML() methods to parse an external ARL file.

## Debugging a RuleSet

An ABLE ruleset can be debugged in a variety of ways. You can include println() statements to write messages to the Java console. You can include trace() messages to write to the ABLE logger or trace viewer. You can use the ABLE Swing-based Debug console or the WSAD ABLE Rule Console to step through rulesets by rule block, individual rules, or by clauses of rules.

To interactively debug a ruleset within the Able RuleSet Editor environment, select the Debug menu option. A debugging console will appear on your display. The debug console contains two frames; the lefthand frame shows the inference engine's current context and the next rule to be evaluated, while the righthand frame shows trace statements from the inference engine. There is a set of buttons on the bottom of the console that you can use to step through your ruleset at different levels of detail.

You can control the amount of information that appears in the righthand frame by using the Trace menu item and selecting a different trace level.

You can step through your ruleset by using either the Run menu items or the buttons on the bottom of the debug console. The menu items and buttons operate as follows:

Clause / Expression
Evaluates the current clause or expression and steps to the next clause.
Rule
Evaluates all clauses in the current rule and steps to the next rule.
RuleBlock
Evaluates all rules in the current rule block and steps to the first rule of the next rule block to be evaluated. If there are no more rule blocks, the debug console will close upon completion of the inference cycle.
Run
Evaluates all clauses, rules, and rule blocks until the next userBreakpoint(this) statement is encountered. If no userBreakpoint(this) statement is encountered, the inference cycle runs to completion and the debug console closes.
Quit
Closes the debug console and the inference cycle runs to completion.

## Tracing and Logging

There are several options for tracing a ruleset. You can use the print() or println() built-in functions to write messages out to the Java console. You can use the trace() or traceFormat() built-in functions to write messages to the ABLE inference trace. In general, it is easiests to use the ABLE rule editors to display the trace messages and to set the trace level. However you can insert this.setInferenceTraceLevel(int) statements into your ruleset to turn tracing on and off at specific parts of your ruleset.

## Data input and output

An AbleRuleSet bean has an input and output buffer which is implemented as a Java Object[]. Data passed into the bean is mapped in order to the specified variables in the

**inputs{}** section and  results are returned in the corresponding slot in the output buffer in the same order as specified in the **outputs{}** section.

# Chapter 3. Frequently Asked Questions

1. How do I use my application objects in an ARL ruleset?
   You import each class just as you would in a Java program. Imported classes can then be used to define variables of that type and to access data members and call methods on Objects of that type.

2. How do I pass data into an ARL ruleset for processing?
   You specify input data in the inputs{ } section. This contains a list of variables (defined in the variables {} section). The input buffer of the AbleRuleSet bean is an Object[] containing the set of Objects that are mapped to the corresponding variables.

3. How do I access data members of an imported object?
   You can access data members using standard Java "dot" notation, **myObject.name**. If the data member is public, it can be accessed directly by the ABLE run-time, otherwise there must be a public accessor (getXXX()) method.

4. How do I call methods on an imported object?
   You can directly call methods on an imported object using standard Java "dot" notation. You must take care that the arguments evaluate to the correct data types so that the ARL compiler can match the method signatures.

5. I need to hold collections of objects. How do I do this in ARL.
   You can import and use any of the Java collection classes. Vectors, Hashtables, HashMaps, Sets, etc. can all be imported and used just as you would in a Java program.

6. How do I decide which inference engine to use?
   In general, the different engines are applied to specific application requirements. For example, if you need to sketch out some procedural code, use the Script engine. If you have data coming in and you want to infer new information use one of the forward chaining engines. If you have a large number of objects (customers, products, etc.) and rules use one of the PatternMatch engines. If you want to find goals use either the Backward engine or the Predicate engine.

7. What is the difference between the PatternMatch and PatternMatchRete engines?
   The engines are functionally equivalent, but their implementation and run-time performance characteristics are very different. In general, the Rete engine is the engine which should be used, because it provides the highest performance. For very small problems, with a few objects and small number of rules, the PatternMatch engine, which has less overhead, may be appropriate.

8. Why would I use ARL Scripting when I can just write Java methods and import them into a ruleset?
   The main reason why you would use Scripting in ARL is to externalize your business logic and keep the procedural code visible with the declarative rules. Once an

application is completed, you may want to recode a Script block into Java for improved performance.

9. I want to have nested if- statements, but I can't nest rules in ARL. How do I work around this?

If you want to have a mutiple-stage if- statement, you need to use ruleblocks and nest their invocations. In the outer if- , if it is true you can call ruleblock A and if it is false, you can call ruleblock B in the else part of the rule. Likewise in ruleblock A or B, you could have other rules that invoke ruleblocks C and D.

10. I want to use primitive Java data types, but ARL only seems to support Objects such as Integer, Double, etc. What do I do?

The ARL built-in data types hold their data values as primitive Java data types. However, they are represented in the ARL run-time by Able data classes (i.e. Java Objects). Think of them as mutable equivalents to Integer, Double, etc. That is, you can change their values via assignment statements. The primitive values will get passed into any methods or functions you invoke.

11. How can I process AbleEvents using rules?

To process AbleEvents using rules you must declare a processAbleEvent() ruleblock. You must also insure that event posting and processing is enabled on the RuleSet bean. When an AbleEvent is received by the AbleRuleSet bean, the processAbleEvent() ruleblock will be invoked. The rules can access the event object via the built-in **event** variable.

12. How do I send asynchronous events to other AbleBeans or to external applications.

You can instantiate an AbleEvent object or any other type of event (TECEvent, etc.) using a new AbleEvent() constructor and the notifyEventListener() AbleBean method.

13. How can I do autonomous behavior using rules?

To have the AbleRuleSet bean wake up at specified intervals to do processing, you must set the sleepTime to the desired interval (in milliseconds) and declare a processTimerEvent() ruleblock. Every sleepTime milliseconds the processTimerEvent() ruleblock will be invoked.

# Chapter 4. What's New in ARL

This chapter contains a list of the significant changes from release to release of the ABLE rule language.

## What's New in ARL 2.0

1. Added getControlParameter() AbleRuleSet methods
2. Changed syntax on Categorical and Discrete variables to use standard Java object syntax with Array initializers. Also support setting initial values on Categorical, Continuous, and Discrete vars.
3. Removed AbleBeanLib, AbleCalendarLib, AbleMathLib, AbleStringLib, and AbleUtilLib from driver and moved selected methods into the ARL.java built-in class.
4. Added a number (approx 20) new ISO Prolog compatible built-in predicates for use by the Predicate inference engine.
5.Added resetVariable(String) and resetVariables(String[]) methods to AbleRuleSet to allow unbinding of variables used by the Forward, Fuzzy, and Backward rule engines.

## What's New in ARL 1.5.0

### Templates

A major functional addition to the ABLE rule language was the addition of ruleset and rule level templates. Templates allow ruleset authors to designate single rules or entire rulesets as being customizable. A set of programming APIs on the AbleRuleSet bean make this function available to Web user interfaces.

### Pluggable Inference Engines

The ABLE rule architecture was enhanced to allow third-party plugins of inference engines. Users can extend the AbleInferenceEngine class and provide their own engines.

### Priorities as variables

Rule priorities used to be limited to real values. They can now be represented by variables or expressions. This allows the ruleset or external API to dynamically change rule priorities individually or in groups (if shared variables are used).

### Performance Tuning

A significant improvement in the run-time performance of the ABLE rule engines was realized between ABLE 1.4.0 and the current release.

### Exception handling and error message reporting

Improved content and formatting of ruleset parsing exceptions.

### Inference engine tracing and logging

Tracing in the AbleRuleSet was found to be impacting performance so a new method of gating trace statements was designed and applied.

### RuleSet APIs

The AbleRuleSet bean APIs went through a comprehensive cleanup. This included changing the instantiateFrom() apis to parseFrom() , simplifying the number and types of exceptions thrown, addition of ability to parse expressions, rules, and ruleblocks.

### Built-in variables changes

The built-in variables have been changed to be typed variables. This means that you can invoke methods on these variables. The **this** variable is now an AbleRuleSet object. The **parent** variable is now an AbleAgent object. The **wm** variable is now an AbleWorkingMemory object. The **inputBuffer** and **outputBuffer** variables are now Object[] variables so you can reference input data using array notation.

### Syntax Changes

While there were no ARL grammar changes, there were changes which may cause parser errors for old rulesets. One of the biggest changes, was the reduction in the number of built-in functions, reduced from approx 30 down to 8. The old functions are still there, but must be accessed as methods using the **this.method()** syntax. The AbleBeanLib and AbleWorkingMemoryLib built-in function libraries were removed. There is a large overhead using the libraries. Most of the functions in AbleBeanLib were deprecated anyway and equivalent methods are accessible using the **this.method()** syntax. Working memory built-in variable (**wm**) is now a typed variable so you can invoke all working memory functions as methods on the **wm** object.

## What's New in ARL 1.4.0

### Major data type changes

Added new Byte, Character, Integer, Long, Short, Float, Double, TimePeriod, and TimeStamp built-in data types. Added dot method notation on user-defined types. So, for example, you can call arbitrary methods with arbitrary parameter lists on any typed variable. Added Array support for all built-in and user-defined data types.

### Major enhancement for Policy rules

The addition of preCondition lists to rules and of the TimePeriod data type add significant new functionality to the ABLE rule language. You can now have rules turn on and off based on date/time periods, month of the year, day of the month, and day of the week. This support provides all of the power and flexibility as defined in the IETF Policy Framework specification for TimePeriodConditions.

## Change summary

1. Java-like syntax -- ARL subset uses equivalent Java syntax, with ARL-specific additions.
2. Arbitrary nesting of AND/OR/NOT expressions.
3. Function nesting (e.g. Math.sin ( Math.tan (1.4) )).
4. Arbitrary math expressions x = x + 1; and y = ( z * 4 ) / k.
5. Data Types - add TimeStamp, TimePeriod, Integer, Double, Float.
6. Inference method names (Forward2 is now PatternMatch, Forward3 is now PatternMatchRete).
7. Add return parameter from ruleblock ... use returnFromRuleBlock() built-in function.
8. Define simple inner classes in ARL: use JikesBT, front-end completed.
9. RuleBlock supports inference method on each rule block.
10. Dot notation on method invocation on object variables.
11. Inference engine controls set using built-in function setControlParameter; are variables rather than constants.
12. Added match() function to do queries against WM (need Selector data type); use find() findAll() in wm.
13. Added named patterns (Expressions) and Selector (added as built-in type).
14. Exception rule block handles exceptions from AbleRuleSet process().
15. Added enabled/disabled flag to rules to allow rules to be turned on/off as necessary (like by date/time).
16. Added start/stop Date/Time stamps to each rule (implemented IETF TimePeriods).
17. Added array support for built-in and user-defined data types, including array initializer support in ruleblock.
18. Remove AbleListVariable and AbleListLib - use java.util.Vector instead.
19. Slash '/' as legal char in identifiers ... changed functions{ } to look for (ident '/' int ), remove '/' from identifier list.
20. Built-in data types are implicit imports rather than keywords (treated similar to imported types).
21. Allow variable definition with null value such as MyType var1 = null ; or MyType var1 ;
22. Added comment field to AbleRuleSet, AbleRuleBlock, Rules, and Variable using JavaDoc style.
23. Implemented do/while , and add for-loop rule.
24. Added ARL.constants to com.ibm.able.rules.ARL class.
25. Added support for bitwise operators &, |, ^, ~ and modulo %.

# Part 2. Language Reference

The Language Reference presents the technical specification of the ABLE rule language in detail. It is meant to be a resource to ruleset authors on issues of the syntax and semantics of ARL.

## Chapter 5. Basic Syntax

While the ABLE Rule Language syntax is similar to standard Java syntax, it is not identical. In this document we highlight areas where differences exist and an unsuspecting Java programmer could be led astray.

### Comments

A comment may appear in only two places: on a line by itself or at the end of a line containing a rule language statement. Comments may be delimited by a beginning // (slash-slash) or inside a multi-line comment block delimited by /* (slash-star) and ended by */ (star-slash). Note that comments are part of the source ARL text document, but comments do not flow through the compilation process (they will be lost during serialization/deserialization. The following lines all demonstrate valid comments:

```
 /*
  * This is a multi-line comment.
  *
 */
  rule1: a = b; // Comment at end of line
```

JavaDoc style comments beginning with a /** (slash-star-star) and ending with a */ (star-slash) are treated as part of the associated ARL object: the ruleset, ruleblocks, individual rules, and variables in the variable declaration section. JavaDoc comments are added to the generated ABLE objects using the setComment() API, so they stay with the ruleset through serialization/deserialization and round-tripping between text ARL and XML formats.

```
/**
*  This comment becomes part of the AbleRuleSet object
*/
ruleset test {

}
```

### WhiteSpace

White space (blanks, tabs, newlines, etc.) may be sprinkled freely throughout a source file. They are all ignored. The following two rule statements are parsed identically:

```
rule_001:
 if (Temperature is slightly elevated) then Pressure is positively moderate ;

rule_002 :
 if (Temperature is slightly elevated)
 then Pressure is positively moderate  ;
```

## Identifiers

Identifiers are the names of things that you define; for example, variables names, fuzzy set names, and rule labels that you create are all identifiers. ARL follows the same rules for identifiers as the Java programming language.

Identifiers are case sensitive. For example, if you define a continuous variable with the name of myVar you cannot later refer to that variable as MYvar as the latter name is taken to refer to a completely different variable that might be of a different type, if it exists at all.

Identifiers are composed of alphabetics (a-zA-Z), numerics (0-9), and the characters _ (underscore) and $ (dollar sign). Numeric characters cannot be used to begin an identifier.

The following are all valid identifiers:

```
thisIsVar1
_this_Is_var_2
foo$
$foo
```

The following are examples of illegal identifiers:

```
 IsVar.1 // Contains a dot
9foo // Begins with a number
```

## Literals

ARL supports the following literal or constant types:

**Boolean** - "true"or "false"
**Character** – character values in single quotes including control '\t' values
**Integer** - standard integer values with leading plus or minus sign, 'l' or 'L' for long
**Floating point** - standard single precision and double precision IEEE floating point values with leading plus or minus sign, decimal point or scientific (e) notation. Also supports 'f' and 'F' for float values, and 'd' and 'D' for double values.

**String** - standard Java string literals enclosed in double quotes and string literal constructions using the '+' operator

## Reserved Words

Rule language keywords are case sensitive. For example, the keyword true must be entered as true. True, tRue, TRUE, and so on are invalid. You may not use any keyword as an identifier.

The following table identifies ARL keywords:

Keywords and their meaning

| | |
|---|---|
| ruleset | Denotes the start of a ruleset |
| import | Loads a Java class for use by rules in the ruleset |
| library | Loads a user defined function library (a Java class) |
| predicates | Defines a list of predicates |
| variables | Denotes the global variables section |
| functions | Declares a list of user defined functions |
| inputs | Denotes a list of input variables expected by the ruleset |
| outputs | Denotes a list of output values returned by the ruleset |
| static | Variable modifier |
| template | Ruleset, Rule and variable modifier |
| void | Return value on a RuleBlock |
| true | Boolean true value |
| false | Boolean false value |

In addition to the above reserved keywords, the following tables identify other, reserved, keywords (all case sensitive):

Fuzzy Hedges

| | |
|---|---|
| about | |
| above | |
| below | |
| closeTo | |
| extremely | |
| generally | |
| inVicinityOf | |
| not | |
| positively | |
| slightly | |
| somewhat | |
| very | |

Rule Related

| | |
|---|---|
| and | Logical and |

| | |
|---|---|
| or | Logical or |
| xor | Logical exclusive or |
| is | Fuzzy set assignment or set membership test |
| if | |
| then | |
| else | |
| when | |
| do | |
| while | |
| until | |
| for | For loop |

# Expressions

Expressions are one of the basic elements used by all ARL rules. In this section we describe the types of expressions supported in the ABLE rule language. Expressions can contain the following items:

Literals -- constant Boolean, String or numeric values such as true, “apples” or 4.5
Variables – any global or local variable
Variable.field – a reference to a data member of an object contained in a variable
Variable.method(args*) – a method call on an object contained in a variable
Function(args*) -- a function call to a built-in or user-defined function.
Fuzzy Hedges -- a linguistic modifier applied to a Fuzzy Set value

## Math Expressions

ARL supports standard Java math expressions using all of the operators and precedence as defined in the Operators section. Not all rule types support expressions in all places. Full expression support is available in assertion rules (ex. X = <expr>; ) and as consequent clauses in if-then, if-then-else, when-do, and while-do rules.

```
Ex1 :  x  =   ( 5 * numVar1) / (-10.2 + offset) ;
Ex2:   println( Math.sin(1.3) * 10.0) ;
Ex3 :
```

## Assignment Expressions

Assignment expressions are used to assign a value to some variable; therefore, only a variable may appear on the lefthand side of an assignment expression. The righthand side of an

assignment expression can be a constant, variable, or expression that evaluates to a result that is compatible with the lefthand side variable. For example, a numeric variable cannot be assigned from a Boolean literal.

There are two assignment operators supported, the standard "=" assignment is used for all variables, and the "is" assignment, which is used to assign fuzzy set values to Fuzzy variables within fuzzy ruleblocks.

```
variable    = literal;
variable    = variable;
variable    = variable.field;
variable    = variable.method(args*);
variable    = class.staticMethodName(args*);
variable    = functionName(args*);
variable    = class.staticMethodName(args*);
```

Fuzzy variables can be assigned fuzzy set values using the "is" operator in a fuzzy ruleblock. Hedges such as about or generally can be included as desired. Fuzzy variables can also be assigned crisp values using the "=" in ruleblocks regardless of the inference engine in use.

```
fuzzyVariable  is hedge* fuzzySet;
fuzzyTemp  = 32;
```

## Action Expressions

Action expressions are used as consequent clauses of if-then rules and are used in the body of most ARL rules. They can be an assignment expression, a function call, or method call. Multiple action expressions can be specified in the body of most rules.

```
            variable.method(args*) ;
            class.staticMethod(args*) ;
            functionName(args*)  ;
variable = variable.method(args*) ;
variable = class.staticMethod(args*) ;
variable =  functionName(args*)  ;
variable =  <expression>
```

## Boolean Expressions

Boolean expressions are used to test the current value of one object against the current value of another object. All of the usual comparison operators are available and have their standard meanings. Note that ARL uses a double equals "==" for equality comparison operations.

Some types of complex Boolean expressions are not supported in the simple if-then inference rules used by the Forward and Backward engines.

The fuzzy inference engine supports optional weights as suffixes to Boolean expressions.

## Pattern Match Selectors

Selectors are pattern match expressions that look for the existence (or non-existence) of objects of the specified data type in working memory, such that the constraints represented in the Boolean expression are true. Selectors are equivalent to the notion of patterns in CLIPS rules.

Selectors can be used to query working memory and return lists of objects using the find(Selector) and findAll(Selector) AbleWorkingMemory methods. Selectors can be used singly or in combinations as part of when/do pattern match rules to perform complex joins of objects for rule processing. See When/Do pattern match rules for details.

Syntax

<dataType> <variableName> [!] ( <Boolean expression>+ )

Parameters

<dataType>

Is either a built-in or user-defined data type that was imported. Note: Arrays are not supported in Pattern Match expressions.

<variableName>
Is an identifier that names a new local variable. This variable is referenced in subsequent constraint and action expressions.

!
Is optional and means that no such object in working memory matches the pattern represented in the comparison clause(s).

<Boolean expression>
Is zero or one Boolean constraint expression. . If no constraint expression is specified, all instances of the specified data type are selected for binding.

Examples

```
Puzzle p (   p.level == "hard" and  p.start == postN  )
```

This positive selector would return all Puzzle objects (bound to variable 'p') such that p's level is equal to 'hard' and p's start is equal to the value of variable 'postN'

```
Puzzle p ! ( p.level == "hard" and  p.start == postN  )
```

This negative selector specifies that there must not exist some Puzzle 'p' such that p's level is equal to 'hard' and p's start is equal to the value of variable 'postN'

```
Puzzle p ( )
```

This positive selector would return all Puzzle objects in working memory.

## Predicates

Predicates are used as a knowledge representation of facts that can be stored in working memory and reasoned about using the Predicate inference engine. Predicates are treated differently from other built-in data types in that they must be declared in the **predicates { }** section at the top of the ruleset. Predicates of any arity and argument data types can be used once the predicate name or functor is declared.

If a symbol with a leading uppercase character is used as an argument in a Predicate, it is taken to represent a local variable. Predicates can be used as stand-alone facts to be asserted and retracted from a working memory, or as clauses in a Predicate rule.

Syntax

```
<predicateName> ( <arg>* )
```

Parameters

<predicateName>
Is the name of a pre-declared predicate. All predicate names must be pre-declared in the Predicates section of a ruleset.
<arg>
Is a list of zero or more arguments to the predicate.

Predicates can take any number of arguments. Arguments may be any of:

A symbol beginning with an uppercase character; these are variables completely local to the current rule.
A symbol beginning with a lowercase character, treated as equivalent to a String literal.
A Boolean literal (true or false).
A numeric literal represented by a Double.

A Character literal in single quotes

A String literal in double quotes.

A list enclosed in square brackets; for example, [a , b, c]. A list may contain any of the elements enumerated here, including other lists and predicates. The special character bar (|) denotes the "tail" of the list.

Other predicates.

The "don't care" symbol, which is the underscore character (_).

Examples

```
foo().              // No args
foo(X).             // Local variable 'X'
foo(bar, baz).      // Either global vars or simple symbols
foo( [a, b | c] ).  // A list of 2 elements followed by any number of additional elements
foo(X, _, bar(Y), "3", 4.4, true, z, [a,b] ).  // Something for everyone!
```

## Built-in Predicates

The following predicates are built-in to the Predicate engine and can be used without being declared in the predicates {} block:

**assert(Fact)** asserts a predicate fact or object to working memory

**asserta(Fact)** asserts a predicate fact or object to front to working memory list

**assertz(Fact)** asserts a predicate fact or object to back of working memory list

**atom_chars(Atom, List)** succeeds if and only if List is a list whose elements are the one character atoms that in order make up Atom.

**atom_concat(Start, End, Whole)** is true if and only if Whole is the atom obtained by concatenating the characters of End to those of First. If Whole is instantiated then all decompositions of Whole can be obtained by back-tracking.

**atom_length(Atom, Length)** is true if and only if the integer Length equals the number of characters in the name of the atom Atom.

**atom_number(Atom, Number)** is true if the Atom is a string whose value is equivalent to the value of Number.

**call(Goal)** allows Goal predicates to be dynamically added to the goal stack

**consult(FileName)** reads an external file containing an optional predicates {} block and a syntactically correct ARL predicate ruleblock. All of the predicate facts and rules are added to the current working memory.

**cut()** cut and backtrack

**fail()** fail the rule

**functor(Term, Name, Arity)** is true if and only if Term is a compound term with functor name Name and arity Arity or Term is an atomic term equal to Name and Arity is 0.

**isList(X)** succeeds if and only if X is a predicate list

**member(X, List)** succeeds if and only if X is a member of List

**nonvar(X)** succeeds if and only if X is not a variable

**not()** name is reserved, not() predicate definition must be part of the ruleblock.

**retract(Fact)** removes a single matching Fact from working memory
**retractAll(Fact)** removes all matching Facts from working memory
**sub_atom(Atom, Before, Length, After, Sub_atom)** is true if and only if Sub_atom is the sub atomof Atom of length Length that appeats with Before characters preceding it and After characters following it.
**unify(X, Y)** is true if and only if X and Y are unifiable.
**univ(Term, [ functor | args ])** takesa predicate list and turns it into a Predicate
**var(X)** is true if and only if X is a variable

## Fuzzy Expressions

A fuzzy expression's left hand side can contain one or more comparisons. Only 'and' or '&&' can be used to connect comparisons. Either side of a comparison can be a variable (Fuzzy or not fuzzy), a fuzzy set, or a constant. The comparison operator can include 'is' for fuzzy variables and fuzzy sets, or a mathematical operator such as '<'. Fuzzy variables can be modified by zero or more comma-separated hedges. Fuzzy expressions can be used in the antecedent of a rule, where they are used as test expressions, or on the consequent of a rule, where they are used as assignment expressions. For example, assume temp is a fuzzy variable with fuzzy sets hot and warm, and heatIndex is a fuzzy variable with fuzzy set high:

```
if  ( temp is very above hot && humidity > 50)  then heatIndex is somewhat high;
```

An example of an illegal expression is:

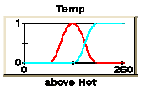
```
if  ( temp is very, very hot or (temp is warm && humidity > 90)) then heatIndex is
high; // 'or' is not allowed
```

**Hedges for Fuzzy Expressions**

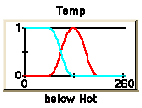
The following hedges are available for qualifying a fuzzy set in a fuzzy clause. Multiple hedges may be used in any combination, but the rule author must be fully aware of what each hedge can do to the shape of a fuzzy set.

Approximation Hedges -- useful for all bell-shaped sets
**closeTo** (1.2)
Narrow approximation.
**about** (2.0)
General approximation; slightly broadens the fuzzy region
**inVicinityOf** (4.0)
Broad approximation; produces a very wide fuzzy space.

Restriction Hedges

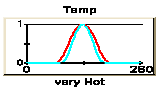
**Above** 

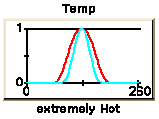

Useful for decreasing linear and sigmoidal sets and for all bell-shaped sets; do not use with increasing linear and sigmoidal sets.

**below** 

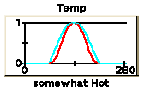

Useful for increasing linear and sigmoidal sets and for all bell-shaped sets; do not use with decreasing linear and sigmoidal sets.

Concentration Hedges -- these hedges depress the surface of linear sets and concentrate or intensify bell-shaped sets; the candidate space is reduced; a domain element must occur farther to the right for an equivalent truth value.

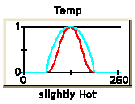
**very** (2.0) 


**extremely** (3.0) 

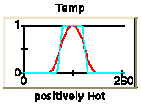

Dilution Hedges -- these hedges raise the surface of linear sets and dilute bell-shaped sets; the candidate space is increased; a domain element must occur farther to the left for an equivalent truth value.

**somewhat** (0.5) 


Complement of very.

**slightly** (0.3) 


Contrast Intensification Hedge

**positively** (n2) 

Truth values less than .5 are decreased, truth values greater than .5 are increased; complement of generally.

Contrast Diffusion Hedge
**generally** (n0.5)

Truth values less than .5 are increased, truth values greater than .5 are decreased; complement of positively.

Negation Hedge
**not**

The fuzzy region is inversed.

## Operators

ARL supports a subset of the Java operators. These include:

| Precedence | Operator | Operand Type(s) | Assoc. | Operation Performed |
|---|---|---|---|---|
| 1 | +, - | Arithmetic | R | Unary plus, unary minus |
| 1 | ~ | Integral | R | Bitwise complement (unary) |
| 1 | ! | Boolean | R | Logical complement (unary) |
| 2 | *, /, % | Arithmetic | L | Multiplication, division, modulo |
| 3 | +,- | Arithmetic | L | Addition, subtraction |
| | + | String | L | String concatenation |
| 4 | << | Integral | | Left shirt |
| 4 | >> | Integral | | Right shift (keep sign) |
| 4 | >>> | Integral | | Right shift (zero fill) |
| 5 | < | Arithmetic | L | Less than |
| 5 | > | Arithmetic | L | Greater than |
| 5 | <= | Arithmetic | L | Less than or equal |
| 5 | >= | Arithmetic | L | Greater than or equal |
| 6 | == | Primitive | L | Equal (have identical values) |
| 6 | != | Primitive | L | Not equal (have different values) |
| 6 | is | Fuzzy | L | Fuzzy comparison or assignment |
| 7 | & | Integral | | Bitwise AND |
| 7 | & | Logical | | Logical AND |
| 8 | ^ | Integral | | Bitwise XOR |
| 8 | ^ | Logical | | LogicalXOR |
| 9 | \| | Integral | | Bitwise OR |
| 9 | \| | Logical | | Logical OR |
| 10 | && | Boolean | L | Conditional AND |
| 11 | \|\| | Boolean | L | Conditional OR |

| | | | | |
|---|---|---|---|---|
| 13 | = | Variable, any | R | Assignment |

The following operators are NOT supported in ARL:

| Precedence | Operator | Operand Type(s) | Assoc. | Operation Performed |
|---|---|---|---|---|
| 1 | ++, -- | Arithmetic | R | Pre- or post increment/decrement |
| 1 | (type) | Any | R | Cast |
| 5 | instanceof | Object, type | L | Type comparison |
| 12 | ?: | Boolean,any, amy | R | Conditional(ternary) |
| 13 | *=, /=, %=, +=, -=, <<=, >>=, …=, &=, ^=, \|= | Variable, any | R | Assignment with operation |

## ARL Constants

Every AbleRuleSet bean has the ARL.java class implicitly loaded. The ARL.java class contains a set of commonly used constant values and several static utility methods. These constants and methods can be accessed using standard dot notation for fields and methods. For example:

: setControlParameter(ARL.Goal, "discount") ;

uses the ARL.Goal constant String (which equates to "Goal"). Other constants are defined for inference engine control parameters. See the ARL.java JavaDoc for details on the available constants and methods.

# Chapter 6. RuleSet Structure

An ABLE source ruleset is a collection of ABLE Rule Language (ARL) statements.

Depending on the number of variables, ruleblocks, and rules that you define, a ruleset can become rather lengthy, but the structure of an ARL source file is quite simple. All of the sections shown are explained in detail elsewhere in this document.

```
ruleset <nameOfRuleSet> {

 <import package.class;>*          // Zero or more statements

 <library package.class;>*         // Zero or more statements

 predicates { <predName>* } ;      // optional


 variables {                       // optional
        <Variable Declaration>+    // One or more statements
 }

 inputs { <variableName>* }    // optional, zero or more names
 outputs{ <variableName>* }    // optional, zero or more names

 functions { <name/arity>* }* // optional, zero or more names



 void init() { <rule>+ };       // optional ruleblock invoked on bean init()


 void preProcess() { <rule>+  }  // optional ruleblock invoked before process()

 void process() using <engine> {  // required ruleblock
    <rule>+                       // One or more rules
 }

 void postProcess() { <rule>+ }  // optional ruleblock invoked after process()



 void processTimerEvent() { <rule>+ }  // optional, invoked on timer pop

 void processAbleEvent() { <rule>+ }  // optional,  invoked when AbleEvent arrives

 void catch() { <rule>+ }   // optional ruleblock invoked when an exception occurs

 void quitAll() { <rule>+ } // optional ruleblock invoked when quitAll is invoked
}
```

## Name

The ruleset statement is the outer-most statement in a source rule file. Comments may appear before the ruleset statement. All other rule language statements must appear within the ruleset statement.

Syntax

```
ruleset <name> {
  <rule language statements>+
}
```

Parameters

<name>
Is an identifier that names the ruleset.

Example

The following ruleset is preceded by a JavaDoc style comment statement and has the name myRuleSet:

```
/**
 * A ruleset to amaze and astound the world!
*/
ruleset myRuleset {
  . . .
}
```

## Import

The import statement allows you to define your own data types so that you can declare and manipulate variables of those types. Each type you declare must be equated to a public Java class. Note: One import statement can define only a single Java class. Asterisks ("*") are not allowed in the import specification.

Syntax

```
import <package.class>;

import com.yourCo.ClassName ;
import com.yourCo.PackageName.* ; // NOT ALLOWED!
```

Parameters

package.class

The fully qualified name of the Java class that you want to import into the ruleset. Instances of the named class must be fully serializable if you want to save your ruleset as a serialized

AbleRuleSet object. The named class must also have a constructor method that matches the number of parameters that you will use when declaring variables of the specified type. If the named class has public members, or if it has public "get" and "set" methods for its members, those members can also be referenced in your rules. See the example below.

The class name is an Identifier that names a new data type. Variables of this type can then be declared and used throughout your ruleset. You must not redefine the built-in data types.

Example

```
ruleSet <nameOfRuleSet> {

  import hanoi.Puzzle;
  import hanoi.Ring;

  variables {
    Puzzle easyPuzzle = new Puzzle("Puzzle1"); // Name
    Puzzle hardPuzzle = new Puzzle("Puzzle2"); // Name

    Ring smallRing = new Ring("RingA", 2); // Name, diameter
    Ring mediumRing = new Ring("RingB", 4); // Name, diameter
    Ring largeRing = new Ring("RingC", 6); // Name, diameter
  }

  . . .
}
```

The first statement imports the "hanoi.Puzzle" class and defines the Puzzle data type, while the second imports the “hanoi.Ring” class and the corresponding Ring data type.

These statements then allow the ruleset author to define variables as shown. Note that for each variable declaration an instance of the specified class is created and the constructor is passed the arguments in parentheses.

In the example above, Puzzle (the hanoi.Puzzle class) must have a constructor that takes a string as a parameter. Ring (the hanoi.Ring class) must have a constructor that takes a string and a number as parameters. Variables as declared above will create an instance of the respective class.

Furthermore, depending on whether the Ring class has public members, or at least public getter and setter methods for internal members, rules such as the following are allowed, where diameter is a publicly accessible member of class Ring.

```
void process() using Script {
 r1:
   if  (smallRing.diameter >= mediumRing.diameter)
   then result = someAction();
```

```
}
```

## Imported Data Types

### Syntax

```
[static] <userType> <variableName> ;            // initial value is null
[static] <userType> <variableName> =  null   // initial value is null

[static] <userType> <variableName> = new <userType> (<arg>*);
```

### Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<userType>
Is a user data type (i.e. a class) that was declared with an import statement in the ruleset.

<arg>
Is a list of zero or more arguments to be passed to the constructor of the Java class represented by the data type. If zero arguments are used, the class must have a null constructor. If any arguments are used, the constructor must take the same number and type of arguments. The arguments may be any of the built-in or imported data types or expressions containing those data types, as well as references to built-in and previously declared variable names. Because the constructor methods must be found using Java introspection, the compile time arguments must be set with valid values in order to find the correct constructor signature. The argument matching algorithm looks for Object types (Boolean, Float, Double, Integer, Long, etc.) first, then primitive (Boolean, float, double, int, long, etc.) types. A parser error will be issued if a matching constructor is not found or if more than one possible match is found.

### Examples

```
ruleSet <nameOfRuleSet> {

  variables {
    Puzzle foo = new Puzzle("HardPuzzle"); // CTOR takes name
    Ring bar = new Ring("RingA", 12.0);  // CTOR takes name & diameter
  }
```

```
  . . .
}
```

## Library

Allows you to import all the public methods in the specified classes (or libraries) as user-defined functions without having to declare each method explicitly. Library functions then become available to be called from rules. Note that if two or more library classes contain duplicate method names with an identical number of arguments, the last method imported is the method that is available to your rules. Also, when using library functions in rules, you must use the number and types of arguments expected by the associated method. No compile-time checking is done, and improper use will result in runtime errors during ruleset processing. Arguments must be of the correct types to match the signature of the method.

See also the functions statement.

See also the general section on interacting with functions. Among other things, the section describes some built-in functions and libraries that are available.

Note that built-in functions always take precedence over imported functions of the same name and arity.

Syntax

```
  library package.class;
```

Parameters

package.class
Specifies a single fully qualified class name, whose public methods are to be made available as user-defined functions. These methods can then be called from rule statements.

Examples

```
ruleset <nameOfRuleSet> {

  library com.binford.Hammers;
  library com.binford.Saws;
  library com.tooltime.FirstAid;

  . . .
}
```

The first two statements loads and defines all the public methods in both Hammers and Saws as user-defined functions. The third statement additionally loads all the public methods in FirstAid. If the Hammers library contains the public method driveNail(), for example, then a rule such as the following can be coded:

```
rule: if (nail_length == 6) then driveNail();
```

## Variables

Variables are declared in the **variables** {... } section of a ruleset; this section appears after any **import** or **library** or **predicates** statements:

```
ruleset <nameOfRuleSet> {

 import java.util.Vector ;
 import java.lang.Math;

  variables {
   <Variable Declaration Statement>+ // One or more statements
  }
  . . .
}
```

The variables section is a required element of any ABLE ruleset. It must contain at least one variable declaration. It must not redefine a built-in variable. All variables declared in the variables section are global variables, visible to all rule methods and rules in the ruleset. Each variable name must be unique. Variable names are case-sensitive.

The values of global variables are reset to their initial values at the beginning of each process() cycle. The only way to avoid this reset is to use the static modifier. Variables with the static modifier are initialized once, when the ruleset is initialized and are not reset prior to each process() cycle.

## Predicates

Specifies the names of predicates used in rules. All predicates referenced in rules must be pre-declared in this section.

Syntax

```
predicates { <predicateName>* };
```

Parameters

predicateName
Is a list of zero or more names of predicates used in rules.

Example

```
ruleset <nameOfRuleSet> {

  predicates { father, mother, sister, brother, aunt, uncle  } ;

  . . .
}
```

## Functions

ABLE provides a UserDefinedFunction class to allow definition of dynamically bound functions to the ABLE rule language. The library statement allows you to take all public methods in a class and turn them into UserDefinedFunctions with a single statement. Alternatively, you can explicitly declare functions and their arity (number of arguments) in the functions statement and programmatically register the actual methods to be called.
The function statement lets you define names for functions that can be referenced in rules and the actual binding to the objects and methods happens when the ruleset is processed.

For example, you could declare a function called notify that takes two arguments. However, you want to be able to have the flexibility of having the notify function resolve to a JMS provider in once case, and a Paging service provider in another. You would specify the function as follows:

```
functions {  notify/2 } ;
```

## Rule Blocks

A rule block is a logical grouping of ARL rules. Each rule block is processed by a corresponding inference engine specified by the **using** clause. A rule block can also declare a return type and pass back return values to the calling rule via the returnFromRuleBlock built-in function.

# Chapter 7. Data Types

This chapter describes the data types that are supported in ABLE rule language rulesets, including built-in types and imported or user-defined types, and arrays.

The ABLE rule language supports these kinds of data types:

- Built-in, which include:
  - Java data type representations such as Boolean, Integer, and Double
  - Able-specific such as Categorical, Continuous, TimePeriod, and Fuzzy
- User-defined, which are Java classes imported to the ruleset
- Arrays of built-in or user-defined data types

## Built-in

The ABLE rule language provides a set of built-in data types that correspond to all of the standard Java primitive (and associated Object) types. For example, the ARL Integer data type holds Java int values, and behaves much like a Java Integer instance, with one important exception. The ARL Integer is mutable, you can assign new values to it.

| | |
|---|---|
| Boolean | A Boolean variable can hold true/false values |
| Byte | A Byte variable can hold signed integers |
| Categorical (String) | A Categorical variable can hold one of a discrete set of String values |
| Continuous (Double) | A Continuous variable can hold a real value (double) in a range from min to max |
| Character | A Character variable can hold a single Unicode character |
| Discrete (Double) | A Discrete variable can hold one of a discrete set of numeric values |
| Double | A Double variable holds a double precision floating point value |
| Expression | An Expression variable can hold any legal simple or compound expression |
| Fuzzy (Double) | A Fuzzy variable can represents a domain from min to max and a collection of FuzzySets over that domain |
| Float | A Float variable holds a single precision floating point value |
| Integer | An Integer variable holds an integer (int) value |
| Long | A Long variable holds an integer (long) |
| Object | An Object variable can hold a reference to any Java object |
| Selector | A selector variable holds an AbleSelector instance used to perform queries against working memory or in when/do rules |

| | |
|---|---|
| Short | A Short variable holds a 16 bit integer (short) |
| String | A String variable holds a Java String |
| TimePeriod | An IETF TimePeriodCondition value |
| TimeStamp | A TimeStamp variable holds a Java Calendar object |

## Boolean

Boolean variables can have one of two values: either true or false. If not specified, the initial value is set to false.

Syntax

```
  [static] Boolean <variableName> ;   // initial value is false
[static] Boolean <variableName> =  true | false ;
[static] Boolean <variableName> = new Boolean ( true | false );
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

Example

```
variables {
  Boolean isMale =  true;
  static Boolean runOnce = new Boolean(false);
}
```

The initial value of 'isMale' is true. Whenever the ruleset is processed, 'isMale' is reset to its initial value of true. The initial value of 'runOnce' is false. This variable is not reset, so it retains the value it had at the last process cycle.

## Byte

Byte variables can hold signed integer values (8 bits) between –128 and 127. If not specified, the initial value is set to 0.

Syntax

```
[static] Byte <variableName>  ;                          // initial value is 0
[static] Byte <variableName>  = <initialValue>  ;        // literal or numeric expression
[static] Byte <variableName> = new Byte ( <initialValue> );
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

Example

```
variables {
  Byte bitmap =  127;
  static Byte mask = new Byte(0x80);
}
```

The initial value of 'bitmap' is 127. Whenever the ruleset is processed, 'bitmap' is reset to its initial value of 127. The initial value of 'mask' is 0x80 hex. The variable is not reset, so it retains the value it had at the last process cycle.

## Categorical

Categorical variables hold String data values. However, these variables may not be assigned any arbitrary String; they must be assigned a value from a predefined list of strings that you declare along with the variable. While you can declare numbers in the list of strings, numbers are treated as strings. This means that the numbers "1" and "1.0", for example, are two different strings and they do not compare equally. "ABC" and "abc" are also two different strings that do not compare equally.

You must define at least one string for a categorical variable. The initial value of a categorical variable is null unless you explicitly assign an initial value via the 2 argument constructor.

The list of valid values can be set in a rule or programmatically using the setValueList() method on the AbleRuleSet bean.

Syntax

```
[static] Categorical <variableName>  ;     // initial value list is empty
[static] Categorical <variableName>  = new Categorical(new String[] {  <strings>+  }) ;
```

[static] Categorical <variableName> = new Categorical(new String[] { <strings>+ }, <initialValue>) ;
[static] Categorical <variableName> = new Categorical(Vector, <initialValue>) ;

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<strings>
Is a list of one or more strings any one of which can be assigned to the variable at any given time. This list can be a Vector of Strings or a String[].

<initialValue>
The initial String value assigned to the variable. This value must be one of the Strings declared in the <strings> list.

Examples

```
variables {
  Categorical Color = new Categorical(new String[] { "black", "green", "green and black", "orange and black" });
  Categorical Legs = new Categorical(new String[] { "6", "8" });
  Categorical Shape = new Categorical(new String[] { "elongated", "round" });
}
```

## Character

Character variables can hold unsigned Unicode characters values (16 bits) in the range /u0000 to /uFFFF. You must specify an initial value for the variable when you declare it.

Syntax

```
[static] Character<variableName> = 'a' ;
[static] Character <variableName> = new Character ( 'a' );
[static] Character<variableName> = '\t' ;    // a tab whitespace char
```

Parameters

<variableName>

Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

Example

```
variables {
  Character smallA =  'a';
  static Character tab = new Character('\t');
}
```

The initial value of 'smallA' is the lowercase character 'a'. Whenever the ruleset is processed, 'smallA' is reset to its initial value. The initial value of 'tab' is '\t'. This variable is not reset, so it retains the value it had at the last process cycle.

## Continuous

Continuous variables are defined over a domain of continuous (double) numbers.

Syntax

```
[static] Continuous <variableName> = new Continuous(<low> , <high> );
[static] Continuous <variableName> = new Continuous(<low> , <high> , <initialValue>);
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<low>
Specifies the lowest value that this variable can be assigned.

<high>
Specifies the highest value that this variable can be assigned.

<initialValue>
The initial value that is assigned to the variable when it is reset.

Examples

```
variables {
```

```
  Continuous Percent_Complete = new Continuous(0.0, 100.0);
  Continuous Range = new Continuous(-100.0, +100.0);
}
```

## Discrete

Discrete variables essentially hold numeric data. However, these variables may not be assigned any arbitrary number; they must be assigned a value from a predefined list of numbers that you declare along with the variable. While you can declare numbers using different syntax, numbers are all stored internally as Java doubles. This means that the numbers 1, 1.0, and +1.000, for example, are all the same number and compare equally.

You must define at least one number for a discrete variable. The default initial value of a discrete variable is 0.0. However, you may explicitly assign an initial value by using the 2 argument constructor.

The list of valid values can be set in a rule or programmatically using the setValueList() method on the AbleRuleSet bean.

Syntax

```
[static] Discrete <variableName>;     // the initial value list is empty
[static] Discrete <variableName> = new Discrete(new Double[] {<numbers>+) });
[static] Discrete <variableName> = new Discrete(new Double[] {<numbers>+) },
<initialValue>);
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<numbers>
Is a list of one or more numbers or numeric expressions , any one of which can be assigned to the variable at any given time. This list can be a Vector of Doubles or a Double[].

<initialValue>
Is a double value assigned to the variable when it is reset. This must be one of the list of <numbers>.

Examples

```
variables {
  Discrete Legs = new Discrete(new Double[] { 2, 4, 6, 8 });
  Discrete Wings = new Discrete(new Double[] { 0.0, 2.0 });
}
```

## Double

Double variables can hold double precision IEEE floating point numbers (64 bits) from 1.8 x $10^{308}$ to 4.9 x $10^{-324}$. If not specified, the initial value is set to 0.0. be assigned any numeric value; internally, they are represented as doubles.

Syntax

```
[static] Double <variableName> = <initialValue>;
[static] Double <variableName> = new Double (<initialValue>);
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<initialValue>
A number or numeric expression that is the initial value of the variable. The 'd' or 'D' characters can be used as a suffice for Double literal values.

Examples

```
variables {
  Double NumberOf_Enrollees = new Double(50);
  Double PercentComplete = new Double(0.34d);

  static Double decrementAmount = new Double(-5.0);
  static Double incrementAmount = new Double(+10);
}
```

## Expression

An Expression variable holds an AbleExpression literal object. This can represent a simple or compound expression and its value can be used anywhere an expression is valid as a rule parameter.

Syntax

```
[static] Expression <variableName> =  <initialValue>;
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<initialValue>
Is a simple or complex expression (with or without parentheses) that is the initial value of the variable. It must be a valid expression.

Examples

```
variables {

 Expression isGoldCustomer =  income > 100000 ;
 Expression isSeniorCitizen  =  age > 65 ;
 Expression isGoldSenior   =   (income > 100000) and (age > 65) ;
 Expression discountRate   =   (income * 0.10) /  age  ;

}
```

## Float

Float variables can hold single precision IEEE floating point numbers (32 bits) from $3.4 \times 10^{38}$ to $1.4 \times 10^{-45}$. If not specified, the initial value is set to 0.0.

Syntax

```
[static] Float <variableName> ;      // initial value is 0.0
[static] Float <variableName> = <initialValue>;
[static] Float <variableName> = new Float (<initialValue>);
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<initialValue>
A number or numeric expression that is the initial value of the variable. The 'f' or 'F' suffix can be used to denote Float literal values.

Examples

```
variables {
  Float NumberOf_Enrollees = new Float(50);
  Float PercentComplete = new Float(0.34f);

  static Float decrementAmount = new Float(-5.0);
  static Float incrementAmount = new Float(+10);
}
```

## Fuzzy

Fuzzy variables are defined over a domain of continuous numbers. Within this range of numbers, one or more fuzzy sets must be defined; therefore, at least one fuzzy set definition is always required to follow a Fuzzy variable declaration.

Syntax

```
[static] Fuzzy <variableName> = new Fuzzy (<low>,<high>) {
   <fuzzySetDeclarations>+ )
}
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<low>
Specifies the lowest value that this variable can be assigned.

<high>
Specifies the highest value that this variable can be assigned.

<fuzzySetDeclarations>
Specifies the fuzzy sets defined over this variable.

Examples

```
variables {
  Fuzzy Percent_Complete = new Fuzzy(0 to 100) { <fuzzySet declarations>+ } ;
  Fuzzy Percent_Complete = new Fuzzy(0,100) { <fuzzySet declarations>+ };
}
```

## FuzzySets

Fuzzy sets are declared in the Fuzzy variable declaration.

There are many types of predefined fuzzy sets that you may declare over a Fuzzy variable. If none of the predefined types meet your needs, you can use the Segments fuzzy set to define your own fuzzy shape.

| | | |
|---|---|---|
|  Beta curve |  Gaussian curve |  Linear<br>Up  Down  |
|  Pi curve |  Segments | Shoulder<br>Left shoulder:  <br>Right shoulder:  |
|  Sigmoid<br>Up  Down  |  Trapezoid |  Triangle |

*WARNING*: Some sets are defined by specifying points describing line segments. Because of the way in which the surface of fuzzy sets are stored internally, you must never describe a perfectly vertical line. For example, you must never describe a trapezoidal set as (5, 5, 10, 10) which represents perfectly vertical left and right sides. As a work around, use (4.9, 5.1, 9.9, 10.1) or something similar instead.

To define "upside-down" sets, use a *Complement* constructor. A *complement* set has exactly the same shape and end points as the original set, but the truth values are inverted. For example, to make an "upside-down" trapezoid, code:

```
Trapezoid myTrapezoid = new ~Trapezoid(myTrapezoid);
```

**Beta curve**

A bell-shaped curve  for representing fuzzy numbers; more tightly compacted than the PI curve and the membership function goes to zero only at extremely large widths.

Syntax

```
  Beta <setName> = new Beta ( <centerPoint>, <width>, <weight>];
  Beta <setNameComp> = new ~Beta( <setName>) ;
)
```

Parameters

<setName>
Is an identifier that names a fuzzy set. The name must not already exist for the containing Fuzzy variable.
<centerPoint>
Is a number within the Fuzzy variable's universe of discourse on which the curve is to be centered.
<width>
Is a number that is the distance from the <centerPoint> to the curve's inflexion point at the 0.5 truth value.
<weight>
Is a number that can attenuate the shape of the curve. This parameter is optional, and if omitted, defaults to 1.0, which produces an unattentuated curve.

Examples

```
Beta medium = new Beta(50,5);     // Inflection points at 45 and 55
Beta fatMedium = new Beta(50,10); // Inflection points at 40 and 60
Beta skinnyCurve = new Beta(50,1); // Inflexion points at 49 and 51
```

**Gaussian curve**

A  bell-shaped curve  for representing fuzzy numbers; the slope of membership goes to zero very quickly with a very short tail.

Syntax

```
Gaussian <setName> = new Gaussian ( <centerPoint> , <widthFactor>);
Gaussian <setNameComp> = new ~Gaussian(setName) ; // complement
```

Parameters

<setName>
Is an identifier that names the fuzzy set. The name must not already exist for the containing Fuzzy variable.
<centerPoint>
Is a number within the Fuzzy variable's universe of discourse on which the curve is to be centered.
<widthFactor>
Is a number that influences the width of the curve, whose overall shape is hard to predict. Typical values are from 0.9 through 5.0 inclusive, although any value greater than zero is allowed. The larger the value, the wider the curve; the smaller the value, the narrower the curve.

Examples

```
Gaussian gauss1 = new Gaussian(50,0.5);
Gaussian gauss2 = new Gaussian(50,9);
Gaussian gauss3 = new Gaussian(50,1);
```

**Linear**

A straight-line  fuzzy set.

Linear Up:  Linear Down: 

Syntax

```
Linear <setName> = new Linear (<beginPoint>,<endPoint>,<direction>);
Linear <setNameComp> = new ~Linear(setName) ; // complement
```

Parameters

<setName>
Is an identifier that names a fuzzy set. The name must not already exist for the containing Fuzzy variable.
<beginPoint>
Is a number within the Fuzzy variable's universe of discourse at which the line is to begin.
<endPoint>
Is a number within the Fuzzy variable's universe of discourse at which the line is to end. The number must be greater than beginPoint.
<direction>

Is a constant, either ARL.Up or ARL.Down, that specifies whether the line is to slope up or down. If the line slopes up, the beginning point (and all points to the left) will have a truth value of 0.0; the end point (and all points to the right) will have a truth value of 1.0. If the line slopes down, the truth values are reversed.

Example

```
Linear tall = new Linear(4.5, 6.5, ARL.Up);
```

**Pi curve**

A bell-shaped curve  for representing fuzzy numbers; the membership value becomes zero at a discrete point.

Syntax

```
Pi <setName> = new Pi (<centerPoint>,<width>,<weight>);
Pi <setNameComp> = new ~Pi(setName) ;    // complement
```

Parameters

<setName>
Is an identifier that names a fuzzy set. The name must not already exist for the containing Fuzzy variable.

<centerPoint>
Is a number within the Fuzzy variable's universe of discourse on which the curve is to be centered.

<width>
Is a number that is the distance from the <centerPoint> to the curve's end point at the 0.0 truth value.

<weight>
Is a number that can attenuate the shape of the curve. This parameter is optional, and if omitted, defaults to 1.0, which produces an unattentuated curve.

Examples

```
Pi foo = new Pi(50,5);   // End points at 45 and 55 (Inflection at 47.5 and 52.5)
Pi bar = new Pi(50,10); // End points at 40 and 60 baz Pi(50,1) // End points at 49 and 51
```

**Segments**

A surface specified by point/truth-value pairs  , with line segments interpolated between the points. At least two pairs must be given, and any values falling outside the range of points have a 0.0 truth value.

Syntax

Segments <setName> = new Segments (<point1, truthValue1, point2, truthValue2 [, ..., pointn, truthValuen]>);
Segments <setNameComp> = new ~Segments( <setName> );  // complement

Parameters

<setName>
Is an identifier that names a fuzzy set. The name must not already exist for the containing Fuzzy variable.
<point>
Is a number within the Fuzzy variable's universe of discourse for which a truth value is specified. Points must be increasing from left to right.
<truthValue>
Is a number from 0.0 to 1.0 inclusive, and specifies the truth value at the given point.

Example

Segments  risk = new Segments(5, 0.0, 10, 0.5, 15, 0.75, 20, 0.75, 25, 0.5, 30, 0.5);

**Shoulder**

Left shoulder:  Right shoulder: 

Syntax

Shoulder <setName> = new Shoulder (<beginPoint>,<endPoint>,<direction>);
Shoulder <setNamecomp> = new ~Shoulder(<setName>) ;

Parameters

<setName>
Is an identifier that names a fuzzy set. The name must not already exist for the containing Fuzzy variable.
<beginPoint>

Is a number within the Fuzzy variable's universe of discourse at which the vertical part of the shoulder is to begin. For Left shoulders, the horizontal part of the shoulder extends from the universe of discourse low value to this point.

<endPoint>

Is a number within the Fuzzy variable's universe of discourse at which the vertical part of the shoulder is to end. The number must be greater than beginPoint. For Right shoulders, the horizontal part of the shoulder extends from to this point to the universe of discourse high value.

<direction>

Is a constant, either ARL.Left or ARL.Right, that specifies whether the horizontal part of the shoulder (that is, the segment with membership values of 1.0) appears on the left or right of the vertical part of the shoulder. The vertical part of left shoulders decrease, while the vertical part of right shoulders increase.

Examples

For some fuzzy variable 0 to 100:
Shoulder foo = new Shoulder( 5, 10, ARL.Left );  // 0-5 horizontal; 5-10 decreasing bar
Shoulder bar = new Shoulder(80, 90, ARL.Right);  // 80-90 increasing; 90-100 horizontal

**Sigmoid**

An S-shaped curve  .

Up:   Down: 

Syntax

Sigmoid <setName> = new Sigmoid (<leftPoint>,<flexPoint>,<rightPoint>,<direction>);
Sigmoid <setNameComp> = new ~Sigmoid( <setName> ) ;  // complement

Parameters

<setName>

Is an identifier that names a fuzzy set. The name must not already exist for the containing Fuzzy variable.

<leftPoint>

Is a number within the Fuzzy variable's universe of discourse that specifies the point at which the curve is to begin.

<flexPoint>

Is a number within the Fuzzy variable's universe of discourse that specifies the point at which the curve is to flex. The number must be greater than leftPoint.

<rightPoint>

Is a number within the Fuzzy variable's universe of discourse that specifies the point at which the curve is to end. The number must be greater than flexPoint.

<direction>

Is a constant, either ARL.Up or ARL.Down, that specifies whether the membership function of the S-shaped curve is to increase from truth value 0.0 to truth value 1.0, or decrease from truth value 1.0 to truth value 0.0, respectively.

Examples

Sigmoid foo = new Sigmoid(50, 55, 60, ARL.Up ); // A symmetrical, increasing S-curve bar Sigmoid(50, 59, 60, ARL.Down) // An asymmetrical, decreasing S-curve

**Trapezoid**

A rectangular-shaped set  , sometimes used in place of bell-shaped sets.

Syntax

Trapezoid <setName> = new Trapezoid
(<leftPoint>,<leftCorePoint>,<rightCorePoint>,<rightPoint>);
Trapezoid <setNameComp> = new ~Trapezoid( < setName>); // complement

Parameters

<setName>

Is an identifier that names a fuzzy set. The name must not already exist for the containing Fuzzy variable.

<leftPoint>

Is a number within the Fuzzy variable's universe of discourse that specifies the point at which the lower left corner of the trapezoid is placed. The point has a truth value of 0.0.

<leftCorePoint>

Is a number within the Fuzzy variable's universe of discourse that specifies the point at which the upper left corner of the trapezoid is placed. The number must be greater than leftPoint. The point has a truth value of 1.0.

<rightCorePoint>

Is a number within the Fuzzy variable's universe of discourse that specifies the point at which the upper right corner of the trapezoid is placed. The number must be greater than leftCorePoint. The point has a truth value of 1.0.

<rightPoint>

Is a number within the Fuzzy variable's universe of discourse that specifies the point at which the lower right corner of the trapezoid is placed. The number must be greater than rightCorePoint. The point has a truth value of 0.0.

Example

Trapezoid foo = new Trapezoid( 50,55,60,65 );

**Triangle**

A triangular-shaped set  .

Syntax

Triangle <setName> = new Triangle (<leftPoint>,<centerPoint>,<rightPoint>);
Triangle <setNameComp> = new ~Triangle(<setName>);  // complement

Parameters

<setName>
Is an identifier that names a fuzzy set. The name must not already exist for the containing Fuzzy variable.
<leftPoint>
Is a number within the Fuzzy variable's universe of discourse that specifies the point at which the lower left corner of the triangle is placed. The point has a truth value of 0.0.
<centerPoint>
Is a number within the Fuzzy variable's universe of discourse that specifies the point at which the peak of the triangle is placed. The number must be greater than leftPoint. The point has a truth value of 1.0.
<rightPoint>
Is a number within the Fuzzy variable's universe of discourse that specifies the point at which the lower right corner of the triangle is placed. The number must be greater than centerPoint. The point has a truth value of 0.0.

Example

Triangle foo = new Triangle(50,55,60);

## Integer

Integer variables can hold signed integer values (32 bits) from $-2^{31}$ to $2^{31}-1$. If not specified, the initial value is set to 0.

Syntax

[static] Integer <variableName> = <initialValue>;
[static] Integer <variableName> = new Integer (<initialValue>);

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<initialValue>
Is an integer number or numeric expression that is the initial value of the variable.

Examples

```
variables {
  Integer NumberOf_Enrollees = 50;
  Integer PercentComplete =  new Integer(34);

  static Integer decrementAmount = -5;
  static Integer incrementAmount = new Integer(+10);
}
```

## Short

Short variables can hold signed integer values (16 bits) from –32768 to 32767. If not specified, the initial value for the variable is set to 0.

Syntax

```
[static] Short <variableName> ;      // initial value is 0
[static] Short <variableName> = 4235;
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

Example

```
variables {
```

```
  Short age =  100;
  static Short counter = 0;     // increment each time process is called
}
```

The initial value of age is 100. Whenever the ruleset is processed, age is reset to its initial value of 100. The initial value of counter is 0. This variable is not reset, so it retains the value it had at the last process cycle.

## String

String variables can be assigned any string value. String comparisons are always case sensitive.

Syntax

```
[static] String <variableName> ;    // initial value is set to “” ???
[static] String <variableName> = <initialValue>;
[static] String <variableName> = new String (<initialValue>);
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<initialValue>
Is a string that is the initial value of the variable.

Examples

```
variables {

  String Size = new String("Large");
  String ErrorMessage = new String("Fatal Error");

}
```

## Object

Object variables can be assigned any arbitrary Java object. If not specified, the initial value is set to null.

Syntax

```
[static] Object <variableName> ;              // initial value is null
[static] Object <variableName> = null;     // initial value is null
[static] Object <variableName> = new Vector();
```

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

Examples

```
variables {

 Object Subruleset1 ;;
 Object something = new Vector();

}
```

Later on, you might refer to the variables in the following way:

```
Rules(

 a1: Subruleset1 = getNewRuleset("foo"); // Assign function value

 a2: something = 67.89;                  // Assign a number
 a3  something = "What's up";            // Assign a string

)
```

## Selector

A Selector variable represents an AbleSelector object. Selectors are used as part of when/do pattern match rules and to query objects in working memory using the find() and findAll() methods on the working memory (wm) built-in variable.

Syntax

[static] Selector <selectorName> = new Selector (<varRef> , < constraintExpr >, <positive>);

Parameters

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

< selectorName >
Is an identifier that names the selector.

<varRef>
Is a reference to the selector variable. This is obtained by using the getVariable() built-in function.

<constraintExpr>
Is a simple or compound expression object used to constrain the values returned by the selector.

<positive>
If true then an object is returned if the constraints evaluate to true. If false, then the selector is a test to see if any of the objects subject to the constraints exist.

Examples

```
variables {

Selector goldCustomer = new Selector(getVariable("income"), new Expression(this, "income > 5"), true);
Selector seniorCitizen  = new Selector(getVariable("age"), new Expression(this,"age > 65"), true) ;

}
```

## TimePeriod

The TimePeriod data type provided functionality equivalent to the IETF TimePeriod condition. There is a start and end timestamp defining the TimePeriod as well as optional masks for month of year, day of month, day of week, and time of day.

The most common use of the TimePeriod data type is in rule preconditions, where it is used with the checkTimePeriodPreConditions() ruleset method to enable or disable rules based on the current date/time and the specified TimePeriod.

Syntax

[static] TimePeriod <variableName> = new TimePeriod (<initialValue>);

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<initialValue>
Is a string that is the initial value of the variable. It must contain a valid date and time format.

Examples

```
     variables {


    // show TimePeriod masks for months, day of week, and day of month
    Integer hexString = 0xFC ;  // test to see that this works ok
    Integer spring = ARL.MARCH + ARL.APRIL + ARL.MAY ;
    Integer summer = ARL.JUNE + ARL.JULY + ARL.AUGUST ;
    Integer quarter3 = ARL.THIRD_QUARTER ;  // show helper constants
    Integer monWedFri = ARL.MONDAY + ARL.WEDNESDAY + ARL.FRIDAY ;
    Integer tueThurs = ARL.TUESDAY + ARL.THURSDAY ;
    Long days1and15 = ARL.DAY_1 + ARL.DAY_15 ;
    Long firstAndLast = ARL.DAY_1 + ARL.LAST_DAY_1 ;

    TimeStamp now = this.getCurrentDateAndTime();  // current time (or simulated
time!)
    TimeStamp start = new TimeStamp("01/01/2002 12:00 AM");
    TimeStamp phase0end = new TimeStamp("07/01/2002 12:00 AM") ;
    TimeStamp phase2start = new TimeStamp("01/01/2003 12:00 AM") ;
    TimeStamp end =   new TimeStamp("01/01/2004 12:00 AM");

    // coarse grained time periods -- campaign example
    TimePeriod phase0 = new TimePeriod(start, phase0end) ;
    TimePeriod phase1 = new TimePeriod("07/01/2002 12:00 AM", "01/01/2003 12:00 AM") ;
    TimePeriod phase2 = new TimePeriod(phase2start, end) ;

    // day of week example
    TimePeriod MonWedFri = new TimePeriod(start, end, 0xFFF0,  monWedFri) ;
    TimePeriod TueThurs =  new TimePeriod(start, end, 0xFFF0, tueThurs) ;

   // time of day example
    TimePeriod morning = new TimePeriod(start, end,   "06:00:00 AM/11:59:00 AM") ;
    TimePeriod afternoon =  new TimePeriod(start, end,"12:00:00 PM/04:30:00 PM") ;
    TimePeriod evening =  new TimePeriod(start, end,  "04:30:01 PM/11:59:00 PM") ;

    // IETF encoding string ctor
    TimePeriod IETFTest = new TimePeriod("20020701T121510/20030101T123005", 0xFFF0,
ARL.DAY_1, 0xFE, "T061015/T123020");

  }


void preProcess() using Script {
     : now = this.getCurrentDateAndTime() ;
     : this.checkTimePeriodPreConditions(now) ;  // allow "what-ifs"
```

```
    : println("preConditions checked!") ;
  }

  void process() using Script {

    RuleX [1]               : println("Running as if it is " + now.getTime()) ;

    Rule7a { morning } [2]  : println(" rule7a - Good morning!") ;
    Rule7b { afternoon } [2]: println(" rule7b - Good afternoon!");
    Rule7c { evening } [2]  : println(" rule7c - Good evening!");


    Rule0a { phase0 }  : println( " rule 0a - phase 0");
    Rule0b { phase1 }  : println( " rule 0b - phase 1");
    Rule0c { phase2 }  : println( " rule 0c - phase 2");
    Rule1 { phase1, phase2 }  : println( " rule 1 - phase 1 or 2" );
    Rule2 { phase0, phase1, phase2 }  : println( " rule 2 - phase 0,1, or 2 ");
    Rule3 : println( " rule 3 -- always") ;

    Rule5 { new TimePeriod(start, end)  }
         : println( " rule 5 ");

    Rule6a { new TimePeriod("01/01/02 12:00 AM", "01/01/05 12:00 AM", ARL.AUGUST,
ARL.DAY_24)  }
         : println(" rule 6a - happy birthday ");
    Rule6b { new TimePeriod("01/01/02 12:00 AM", "08/23/02 11:59 PM", 0xFE, "00:01:00
AM/11:59:00 PM"),
             new TimePeriod("08/25/02 12:00 AM", "01/01/02 11:59 PM", 0xFE, "00:01:00
AM/11:59:00 PM")  }
         : println(" rule 6b - happy unbirthday ");


    Rule8a { MonWedFri } : println(" rule 8a - Mon, Wed or Fri ");
    Rule8b { TueThurs }  : println(" rule 8b - Tues or Thurs ");

  }
```

## TimeStamp

A TimeStamp variable represents a Date and Time. It is implemented using a Java Calendar object.

Syntax

[static] TimeStamp <variableName> = new TimeStamp (<initialValue>);

Parameters

<variableName>
Is an identifier that names the variable.

static
Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

<initialValue>
Is a string that is the initial value of the variable. It must contain a valid date and time format.

Examples

```
variables {

 TimeStamp t3 = new TimeStamp("11/27/01 11:59 PM");
 TimeStamp startTime = new TimeStamp("03/18/02 10:00 AM") ;

}
```

## User-Defined (imported) types

Standard Java language classes and application classes can be imported and used in ABLE rulesets. Once imported, they can be referenced like any of the built-in data types. They can be used to declare variables, used as parameters in function or method calls.

## Arrays

You can define arrays of any built-in or imported data types.

Syntax

```
[static] <type>[]  <variableName> ;              // initial value is null
[static] <type>[]  <variableName> = null;     // initial value is null
[static] <type>[] <variableName> = new <type>[ <index expr>];
[static] <type>[] <variableName> = new <type>[ ] { <initializerList > } ;
```

Parameters

<type>
Is one of the built-in or imported data types (including Java language types).

<variableName>
Is an identifier that names the variable.

< index expr>
Is an expression which evaluates to an integer value. Note: if an index expression is used, an initializerList cannot be used.

< initializerList >

Is a comma-delimited list of literals or expressions which evaluate to objects compatible with the declared type. Note: if an initializer list is used, then an index expression cannot be used.

static

Is an optional keyword that indicates that the variable is not to be reset to its initial value when the ruleset is reset.

For example:

```
variables {

  Boolean[] listOfFlags = new Boolean[3] ;
  Boolean[] listOfGags = new Boolean[5 * 2] ;
  Boolean[] listOfBags = new Boolean[] { true, false, true } ;
}
```

Arrays can hold instances of the corresponding data type (literals) or variables. Attempts to assign values of the incorrect data type will cause a run-time error. If not specified via an array initializer list, the array elements will be initialized to null values. ARL Arrays are represented internally by Object[] arrays.

# Chapter 8. Variables

## Definitions

Variables are defined in a special **variables { }** section of an ARL rule set. All variables are defined using Java-like syntax. The data types must be one of the built-in ARL supported types or an imported user-defined or Java language class. All variables defined in the variables {} section are global to the ruleset and are visible to all ruleblocks and rules. The variables sections is optional.

## Modifiers

There are only two modifiers supported in the ABLE rule language, **static** and **template**.

### Static

The most commonly used one is to declare **static** variables. All global variables are reset to their initial values at the start of each AbleRuleSet process() invocation. However, variables with the static modifier are NOT reset. This allows counters or other statistics to be accumulated over multiple process() invocations.

### Template

The second modifier, **template**, is used to denote variables and rules which are to be used as part of the ARL template processing. Variables and rules which have the template modifier are not executable in the current ruleset. They provide customization points for rule authoring. Please see the section on templates for a more detailed discussion.

## Inputs and Outputs

ABLE rulesets maintain an input buffer and an output buffer through which data can be exchanged with other AbleBeans or other Java programs. A ruleset's input buffer can be used to set the values of specific variables just before inferencing starts, and the output buffer can be used to make the values of specified variables available to external code after inferencing is complete. Variables can have their values set from the input buffer by a calling program and have their values at the end of processing written to the output buffers. These variables are specified by using the inputs and outputs statements.

The inputs and outputs statements are optional. If used, the statements must appear immediately after the **variables** {...} section of the ruleset.

```
ruleset <nameOfRuleSet> {
```

```
 variables {
          . . .
}

inputs { var1, var2,  varN} ;

outputs { varA, varB, var1, var2  } ;

  . . .
}
```

## Inputs

When variables are declared, they can be given an initial value. Thus, when inferencing starts, a variable always starts out with the initial value (which could be null). But sometimes you may want variables to start out with different values than those with which they are declared. One way to do this is to use assertion statements in a preProcess() rule block and assign a variable a value returned from a function call.
For example:

```
a1: myVar = someFunction();
```

where someFunction must determine what the variable's value must be. Sometimes this is necessary, but this technique involves the overhead of a call each time. Another way to accomplish setting a variable is to use an inputs statement to tell ABLE to set the variable's value from its input buffer. Then, it is up to the calling Java program to set the buffer before invoking the process() method on the ruleset bean. However, ABLE is setup to do this as a normal way of business (called dataflow) and is very efficient at it. In fact, you can pump variable values from a file or database so that each inferencing cycle uses new data values.

Inputs is an optional statement and if used, its content may be empty. Note that if you leave the inputs statement empty, the ruleset will not expect to have an input buffer at runtime, and will not do any related input buffer processing.

Syntax

```
inputs { <variableNames>* };
```

Parameters

<variableNames>
Is a list of zero or more identifiers that name previously defined variables. If there is an input buffer at runtime, the first element in the input buffer is assigned to the first named variable, the

second element in the input buffer is assigned to the second named variable, and so on. If there is no input buffer at runtime, or if there are not enough values in the input buffer, a runtime error is signaled. Extra values in the buffer are ignored.

Examples

```
 ruleset <nameOfRuleSet> {

  variables {

  }

 inputs { }        // No input variables expected at runtime

  . ..
}

 ruleset <nameOfRuleSet> {

   variables {
     String a;
     Boolean b ;
  }

  inputs { a, b };
  …
}
```

The input buffer must contain at least two values: the first value is assigned to variable a, the second value to b.

```
 ruleset <nameOfRuleSet> {

  variables {
    Continuous c = new Continuous(0, 100);
    Categorical d =  new Categorical(new String[] {"A", "B", "C"});
  }

 inputs { c,  d };

// Input buffer must contain at least two  values: the first value is assigned
/ to variable c, the second value to d.
  . .  .
 }
```

## Outputs

The outputs statement is similar to the inputs statement. Use this statement to designate those variables whose values are to be placed into an external output buffer after inferencing is complete, so that the values may be read by another Java program.

Outputs is an optional statement, although if used, its content may be empty. If you leave the outputs statement empty, the inference engine will not write any values to an output buffer.

Fuzzy variable values are always presented to the external world as a defuzzified crisp value.

Syntax

```
outputs { <variableNames>* };
```

Parameters

<variableNames>

Is a list of zero or more identifiers that name previously defined variables. At the end of inferencing, the first element placed in the output buffer is the value of the first named variable, the second element placed in the output buffer is the value of the second named variable, and so on.

Examples

```
ruleSet noInputsOrOutputs (

  variables {
    Continuous c = new Continuous(0, 100);
    Categorical d = new Categorical(new String[] { "A", "B", "C"});
    ...
  }

  inputs {};        // No input variables expected at runtime
  outputs {};      // No output variables written at runtime

  . . .
)

ruleset outputsOnly (

  variables {
```

```
      Continuous c = new Continuous(0, 100);
      Categorical d = new Categorical(new String[] { "A", "B", "C" });
      ...
    }

    inputs {};      // No input variables expected at runtime
    outputs {c, d };  // Output buffer will contain two elements:
                      //   the first element is the value of
                      //   variable c, the second element the
                      //   value of variable d.

  . . .
  }



  ruleset outputsOnly {

    variables {
      Continuous c = new Continuous(0, 100);
      Categorical d = new Categorical(new String[] {"A", "B", "C"});
    }

    inputs  {  };
    outputs  { c,  c  };
  }

}
```

In the example above, no input variables expected at runtime. The output buffer will contain two elements: the first element is the value of variable c, the second element is also the value of variable c.

## Global versus Local

All variables declared in the variables section of a ruleset have global scope. They are visible to the entire ruleset. There are two cases where variables are not global. In When/do pattern match rules, all selector variables are local to the rule. In predicate rules, any variable that starts with an uppercase character is defined local to the rule.

## Built-in variables

ARL provides the following built-in variables, all of which may be referenced in rules. These variables are, for the most part, read-only variables. You must not attempt to redefine them in the variables section of a ruleset.

| | |
|---|---|
| this | A reference to the ruleset itself. You can invoke AbleRuleSet Methods using this.methodXYZ(). See the JavaDoc for a list of |

| | |
|---|---|
| | methods. |
| parent | A reference to the object, if any, that contains the ruleset (typically an AbleAgent). |
| functionList | The list of library functions. (Hashtable) |
| variableList | A list of global variables declared in the ruleset variables section. (Hashtable) |
| inputBuffer | A reference to the ruleset's external input buffer. It is an Able Object array. |
| outputBuffer | A reference to the ruleset's external output buffer. It is an Able Object array. |
| wm | A reference to the ruleset's working memory. (AbleWorkingMemory) |
| null | The value null. |
| exception | A run-time exception |
| event | The event passed on a processAbleEvent() method call (AbleEvent) |

# Chapter 9. Rule Blocks

All rules in an ABLE ruleset are grouped into rule blocks. A ruleset can have many rule blocks, and rules in one rule block can invoke rules in other rule blocks by using the invokeRuleBlock() built-in method.

Rule block methods are the last statements in a ruleset.

Syntax

```
<returnType>  <nameOfRuleBlock>() using <inferenceEngine> {
    <rule>+                          // One or more rules in a rule block method
  }
```

Parameters

<returnType>
The return data type, either a built-in or imported type or void.

<nameOfRuleBlock> ()
A valid ARL identifier used as the globally unique name of the ruleblock. The name must be followed by empty parentheses. Ruleblock parameter lists are not supported in ARL.

<inferenceEngine>
The name of one of the built-in or user-defined inference engines, including Script, Forward, Backward, PatternMatch, PatternMatchRete, and Predicate.

<rule>
One or more rules to be processed by the specified inference engine.

There are several special rule blocks with reserved names. Only the **process**() rule block is required. All of the others are optional.

The optional rule blocks are **init**(), **preProcess**(), **postProcess**(), **processAbleEvent**(), **processTimerEvent**(), **quitAll**() and **catch**(). The init() and quitAll() rule blocks, if present, are processed when the respective AbleRuleSet bean init() or quitAll() method is called. It can be used, as the name implies, to perform one-time initialization of the ruleset. The preprocess() and postProcess() rule blocks, if present, are called prior to and after the process() rule block each time the AbleRuleSet bean's process() method is called. All of these rule blocks typically use the sequential or Script processing, but other inference engines could be used if required.

The optional processTimerEvent() ruleblock is processed when the AbleRuleSet bean's timer pops. The AbleRuleSet bean must be configured to have the timer running and the sleep time must be specified. For example, you could configure a RuleSet bean to wake up every 5 seconds and have the processTimerEvent() ruleblock perform some polling behavior and analysis of data. Mixing asynchronous timer processing and synchronous processing (via process() method) is not recommended.

The optional processAbleEvent() ruleblock is processed when the AbleRuleSet bean's processAbleEvent() method is invoked either directly by another AbleBean or indirectly due to the arrival of an asynchronous AbleEvent. The AbleRuleSet bean must be configured to queue and process events. Mixing asynchronous timer processing and synchronous processing (via process() method) is not recommended.

The optional catch() ruleblock is processed when an exception occurs during the evaluation of the AbleRuleSet bean's process() method. So any errors in the preprocess(), process() or postProcess() rule blocks will result in control being passed to the catch() ruleblock . Once in control, the catch() ruleblock can determine the nature of the error by accessing the exception built-in variable. The major options are to post a message and "eat" the exception or to post a message and then rethrow the original exception or a new exception.

Ruleblocks are a great way to partition rulesets into smaller logical chunks. All rulesets share access to the global ruleset variables and they can share working memory objects as desired. Any number of ruleblocks can be defined. The names of rule blocks follow the same rules as other Identifiers in ARL. User-defined rule blocks are only processed when an invokeRuleBlock() function is used from one of the pre-defined ruleblocks. They can use any of the ABLE inference engines specified through the using clause.

## Rule Block Statement Syntax

Syntax

```
<returnType> <nameOfRuleBlock>() using <engine> {
 <rule>+              // One or more rules in a rule block
}
```

Parameters

<returnType>
Specifies the return value (if any) returned by this ruleblock when it is called via invokeRuleBlock() function. The return type can be "void" or any ARL built-in or imported data type. Return values are returned using the returnFromRuleBlock() built-in function.

<nameOfRuleBlock>

One of init(), process(), postProcess(), processTimerEvent(), and catch() are case sensitive, designating one of the predefined rule blocks. Other rule blocks can use any unique identifier name. A ruleblock name and inference engine specification are required.

<rule>
Is one or more rule statements.

Examples

```
ruleset <nameOfRuleSet> using Forward {
 . . .

  void init() using Script {                // Special rule block
   a1: someVar = someFunction();
   . . .
  }


  void process() using Script {       // Special rule block
   m1: if   (someVar >= 100)
       then result = invokeRuleBlock("fooBlock");
   .  .  .
  }


  String fooBlock() using Script {          // Callable rule block
   f1: someVar = someOtherFunction();
   .  .  .
      : returnFromRuleBlock(“the result”) ;
  }

}
```

## Using clause

using <InferenceEngine>

The using clause allows you to specify which inference engine should be used to process the rules in a ruleblock. If left unspecified, the default InferenceEngine is Script. The specified InferenceEngine applies only to the rules in that rule block.

Syntax

```
void process() using <inferenceEngine>
void process() using <com.myCompany.myInferenceEngine>
```

void process() using Script | Forward | Fuzzy | PatternMatch | PatternMatchRete | Predicate | Backward

## Inference Engines

The ABLE toolkit provides several different inference engines for processing rules. In this section, we briefly describe these engines. See the chapter on inference engines for more details on their configuration or control parameters and how they process rules.

**Fuzzy**

Specifies that the fuzzy forward inference engine should be used. There are three main inference methods used by the fuzzy inference engine. These control parameters are set via the setControlParameter() built-in function.

### FuzzyAdd

Specifies that the fuzzy solution set is updated by adding the minimum truth value of the consequent fuzzy region, bounded by 1.0. This method is generally used with CorrelationMethod(Product). See the AlphaCut, CorrelationMethod and DefuzzifyMethod control parameters.

### MinMax

Specifies that the fuzzy solution set is updated by using the maximum of the minumum truth value of the consequent fuzzy set. This method is generally used with CorrelationMethod(Minimise). See the AlphaCut, CorrelationMethod and DefuzzifyMethod control parameters.

### ProductOr

Specifies that the fuzzy solution set is updated by using 1 minus the product of (1 minus the truth value of the consequent fuzzy set) and (1 minus the predicate truth). This method is generally used with CorrelationMethod(Product). See the AlphaCut, CorrelationMethod and DefuzzifyMethod control parameters.

### Backward

Specifies that the solution is obtained by doing backward chaining.

### Forward

Specifies that the boolean solution is obtained by doing simple forward chaining. This is the lightest weight of the forward chaining algorithms and is adequate for small to medium sized rulesets.

**PatternMatch**

Specifies that the solution is obtained by doing forward chaining that makes use of working memory.

**PatternMatchRete**

Specifies that the solution is obtained by doing forward chaining that makes use of working memory and Rete networks. This is the heaviest duty of the forward chaining algorithms and should be used for very large rulesets.

**Predicate**

Specifies that the solution is obtained by doing backward chaining with backtracking to find a predicate goal.

**Script**

Specifies that the rules are processed sequentially (in definition order) with no special inferencing. Priorities are ignored.

## ARL-defined rule blocks

### init()

The optional init() ruleblock is evaluated **once**, when the AbleRuleSet bean init() method is called. The init() block is meant to perform one-time initializations such as loading working memory or initializing variables using external property files, etc. **Variables assigned in the init() ruleblock must be static.** Otherwise they will be reset to their initial values prior to process().

### preProcess()

The optional preProcess() ruleblock is evaluated when the AbleRuleSet bean process() method is called. It is evaluated prior to the process() ruleblock.

### process()

The process() ruleblock is evaluated when the AbleRuleSet bean process() method is called. Optional preProcess() and postProcess() ruleblocks are evaluated prior to and after the process() ruleblock. This ruleblock is required.

### postProcess()

The optional postProcess() ruleblock is evaluated when the AbleRuleSet bean process() method is called. It is evaluated after the process() ruleblock.

### processTimerEvent()

The optional processTimerEvent() ruleblock is evaluated when the AbleRuleSet bean's timer pops. Note that the bean's timer event processing must be configured (sleepTime set) and enabled.

### processAbleEvent()

The optional processAbleEvent() ruleblock is evaluated when the AbleRuleSet bean's processAbleEvent() method is invoked. Note that the bean's event processing and posting must be enabled. The rules in the ruleblock can access the current event via the built-in variable **event**.

### catch()

The catch() ruleblock is optional. If defined, control will be passed to it whenever an exception occurs during the evaluation of the bean's process() method.

### quitAll()

The quitAll() ruleblock is optional. If defined, it is called at the end of the AbleRuleSet bean's lifecycle, when the bean's quitAll() method is invoked.

## User-defined rule blocks

The user can define any number of rule blocks. Each rule block must have a unique name and must specify the associated inference engine to be used with the rule block.
User-defined rule blocks can be called from the init(), process(), or other built-in rule blocks using the invokeRuleBlock() function. Results are usually returned by setting values on global variables. However, you can also directly return a value using the returnFromRuleBlock() built-in function.

# Chapter 10. Rules

All rule statements within a ruleset are grouped within one or more rule block statements.

ABLE provides many different types of rules, from simple assertion statements, to If-Then rules, to more complex When-Do pattern matching rules. Rules in Script rule blocks are processed sequentially, so the order in which you enter them may be important. Rules appearing in rule blocks using inference engines other than Script, are processed in an order that is determined by the inference engine.

All rules may have a priority, but not all inference engines use the priority, even if specified. Other inference engines rely on priority to do a good job. Rules may also have a label, but these are only for the author's use; they are completely ignored by the inference engines. However, if you plan to use the AbleRuleSet APIs to manipulate a ruleset, you must use rule labels as the rule identifiers.

There are no limits on the number of rules you can have in a rule block, or in the number of ruleblocks in a ruleset.

## General Rule Syntax

A rule contains some header information, including a label and priority and the rule body.

Syntax

```
/**  rule comment */
<label>  <{ preConditionA <,preCondition>* } >  <[priority]> : <ruleBody> ;
```

Parameters

<comment>
Is an optional JavaDoc style comment which is parsed and maintained as part of the rule object.

<label>
Is an optional Identifier that you can use to tag your rule with meaningful information. Labels show up in trace and debug output, but are not otherwise used by Able.

< { preConditionA <,precondition>* } >
Is an optional list, in braces or curly brackets, that represents preconditions on the rule. The most common usage is to have references to TimePeriod variables or to new TimePeriod instance expressions. Rules with preconditions can be processed using the built-in function checkTimePeriodPreConditions(Calendar) or by a user-defined function. Note: the type of objects or variables is not limited by ABLE. Users can add their own objects to the

preconditions list and then provide functions to evaluate and set the rule enabled flag in the preprocess() ruleblock.

<[priority]>
Is an optional number, numeric variable, or numeric expression in square brackets, that represents the priority of the rule. Priorities are represented as doubles in ABLE and have meaning only to inference engines that use them. When omitted, a rule's default priority is 1.0.

**:**
The colon is a **required** element, even if the rule label, preCondition list, and priority are omittted.

<ruleBody>
Is a basic rule type, either an assertion rule, an If-Then rule, a When-Do rule, or a Predicate rule, all of which are made up of common rule clauses. Common clauses are assignment clauses, comparison clauses, pattern match clauses, and predicate clauses.

Example

```
void exampleRuleBlock() using Script {
  A1: someVar = someOtherVar ;

 // Label is  “A1”,  Priority is omitted, defaults to 1.0
// RuleBody is “someVar = someOtherVar”

  r2 [5] : if (someVar == true) then x=y;

// Label is 'r2', Priority is ‘5’
// RuleBody is  'if someVar == true then x=y'

  :female=false;

// Label is omitted,   Priority is omitted
// RuleBody is 'female=false'
}
```

## Label

Rule labels are optional and must be followed by a **:** (colon). Rule labels are useful for organization and when debugging rulesets. They are used as identifiers by AbleRuleSet APIs. Rule labels, if used, must be unique; that is, two rules cannot have the exact same label:

```
rule4a: c = d; // Labeled rule
rule4b: e=f ; // Labeled rule
rule4b: f=g // causes parser exception: rule label already used!
```

: a = b // Unlabeled rule

## Priority

Each rule has an optional priority value. The priority value (if set) is used by the inference engines as part of the conflict resolution algorithm. All other things being equal, a rule with a higher priority value will be selected to fire before other rules of lesser priority. The Forward, PatternMatch, and PatternMatchRete engines use the priority value. All other inference engines ignore this value.

## PreConditions

Each rule has an optional precondition list. The precondition objects are stored as a rule property (accessed using the getPreConditions() method). Note that this is a generic mechanism for attaching arbitrary meta-data (i.e. Java Objects) to ABLE rules. Once there, this meta-data can be interrogated by calling methods from rules as part of the preProcess() ruleblock or during inferencing.

The built-in function, checkTimePeriodPreConditions(Calendar) is one such method designed to process TimePeriod preCondition objects. The TimePeriod class allows a starting and ending date and time to be specified for the rule, along with masks for months of the year, day of the month, day of the week, and time of day.

# Rule Types

The body of a rule can be one of several different types. These can be separated into two major groups, scripting rules and inferencing rules. The scripting rules include assertions, if-then-else, while-do, do-while, do-until, and for-loop. The inferencing rules include if-then, pattern match when-do rules, and predicate rules. In the following sections we describe each rule type.

## Assertion or Assignment

Assertions are simple assignment statements. Since the inference engines evaluate all assertions before evaluating any conditional rules, you can use assertions to give initial values to variables. However, if you use assertions simply to assign initial values to
variables, it is preferable, in most cases, to accomplish this through an external input buffer (and values assigned in this way always take precedence over any assertion rules in a ruleset).

There are two useful ways to use assertions:

1.You may want to use assertions to test a ruleset from within the ruleset editor; you can quickly change asserted values, compile the ruleset, and then re-run it, repeating the test cycle quickly and easily. When you are done testing, comment-out the assertions and create an **inputs** statement so that values can be assigned through the external input buffer.

2.You may want to assign a variable the value of a user-defined function. In this case, do not pass a value through an input buffer, but let the user-defined function determine the initial value.

Syntax

See Assignment Expressions

Example

```
A1: someVar = someOtherVar ;
    :isMale = determineIfMale(person);
    :  age = 40 ;
```

## Conditional If-Then (Inference) rules

If-Then rules are evaluated conditionally, based on known data. The order of evaluation is determined by the inference engine, but if your ruleset is a pure fuzzy ruleset, all rules are essentially evaluated in parallel, each rule playing a part in the solution.

If-Then rules have any number of antecedent clauses separated by the keyword **and**, and one or more consequent clauses appearing in the then part of the rule. The then keyword is optional. Braces must be used when more than one consequent clause appears in the then part of the rule.

Syntax

```
if (
    <Boolean expression> )
then {
    <Action expresson>+
}
```

Parameters

<Boolean expression>
Is a simple Boolean expression made up of one or more Boolean or Fuzzy comparisons ANDed together.

<Action expression>
Is a list of one or more action expressions separated by semicolons.

Example

```
void  myRuleBlock using Forward() {

 Test_the_Temperature:          // Label, no priority
    if
        ( temp is very hot and   // Comparison clause, connector 'and'
         switch == "on" [0.5])   // Comparison clause
    then
         pressure is high ;      // Assignment clause

 F1: if (a>=b) then c=d ;

 :   if  ( c < d ) then  x = z(parm1, parm2);

}
```

## If-then-else script rules

If-Then-Else script rules are evaluated conditionally, based on known data. They are only processed by the Script engine and are evaluated in the declaration order.

If-Then rules have a single Boolean expression, and one or more action expressions appearing in the then part of the rule. The else part of the rule is optional and, if present, contains one or more action expressions. The **then** keyword is optional. Braces must be used when more than one action appears in the then or else part of the rule.

Syntax

```
if (
    <Boolean expression> )
then {
    <Action expression>+
} else {
    <Action expressoin>+
}
```

Parameters

<Boolean expression>
Is a simple or complex Boolean expresson.

<Action expression>
Is a list of one or more action expressions separated by semicolons.

Example

```
void myRuleBlock using Script()  {

  Test_the_Temperature:          // Label, no priority
     if    ( temp is very hot and    // Comparison clause, connector 'and'
           switch == "on" [0.5])     // Comparison clause
     then pressure is high ;        // Assignment clause

  F1: if (a>=b) then c=d ;

  :   if  ( c < d )  then x = z(parm1, parm2);
}
```

## When-do pattern match rules

When-Do rules are evaluated conditionally, based on known data. The order of evaluation is determined by the inference engine.

When-Do rules may have any number of Selectors or pattern match expressions separated by the keyword **and** , and multiple consequent clauses.

Syntax

```
when ( <patternMatchClause>+  )
do   {
     <action expresson>+
     }
```

Parameters

<patternMatchClause>
Is a list of one or more pattern match Selectors, each clause separated by an optional ampersand ('&&') character. The entire set of selectors must be within the parentheses that make up the **when** block. At least one selector is required.

<action expression>
Is a list of one or more action expressions separated by semicolons. The entire set of actoins must be within the brackets that make up the **do** block. At least one action is required.

Example

```
void myRuleBlock using PatternMatch() (

: when (
    Puzzle p (  p.level == "hard" and  p.start == postN  ) &
     Ring   r  (  r.diam > 0 )
   )
  do {
     solution.level = p.level;
     solution.ringSize = r.diam;
  }
}
```

## While/do iteration rules

While-Do rules are looping rules, evaluated conditionally, based on known data. The test conditions are evaluated, and, if true, the body of the rule is processed. The cycle repeats until the test conditions are no longer true.

While-Do rules have any number of test conditions separated by the keyword and (case insensitive) in the While part of the rule, and one or more consequent clauses appearing in the Do part of the rule.

Syntax

```
while (  <Boolean expression>  )
  do {
     <Action expression>+
   }
```

Parameters

<Boolean expression>
Is a Boolean test expression that is evaluated at the top of the loop. If no test expression is specified, the rule will loop forever with no means of escape.

<Action expression>
Is a list of one or more action expressions separated by semicolons.

Example

```
void exampleRuleBlock() using Script {

  Check_and_Lower_Temp:
     while (  temp >= 212 and  switch == "on"   )
       do {
             invokeRuleBlock("lowerTemp");
        }
}
```

In the above example it is to be hoped that the invoked rule block, lowerTemp, will set the global variable temp to a new temperature, or set the global variable switch to "off".

## Do/until () rules

Do-Until rules are looping rules, evaluated at least once. The **do** body of the rule is always processed first and then the test expression is evaluated. If the expression evaluates to false, the cycle repeats until the test expression is true.

Syntax

```
do {
   <Action expression>+
  }  until ( <Boolean expression> );
```

Parameters

<Boolean expression>
Is an Boolean test expression. If no test expression is specified, the rule will loop forever with no means of escape.

<Action expression>
Is a list of one or more action expressions separated by semicolons.

Example

```
void exampleRuleBlock() using Script {

 Check_and_Lower_Temp:
     do {
          rc = invokeRuleBlock("lowerTemp");
      } until (temp >= 212 and  switch == "on");


}
```

In the above example it is to be hoped that the invoked rule block, lowerTemp, will set the global variable temp to a new temperature, or set the global variable switch to "off".

## Do/while Rules

Do-While rules are looping rules, evaluated at least once. The **do** body of the rule is always processed first and then the test expression is evaluated. If the test expression is true, the cycle repeats until the test expression is no longer true.

Do-While rules have a single arbitrarily complex Boolean test expression separated by the keyword and (case insensitive) in the **while** part of the rule, and one or more consequent clauses appearing in the **do** part of the rule.

Syntax

```
do {
      <Action expression>+
   }   while ( <Boolean expression> );
```

Parameters

<Boolean expression>
Is a Boolean test expression. If no test expression is specified, the rule will loop forever with no means of escape.

<Action expression>
Is a list of one or more action expressions separated by semicolons.

Example

```
void exampleRuleBlock() using Script {

 Check_and_Lower_Temp:
     do {
          rc = invokeRuleBlock("lowerTemp");
     } while ( temp >= 212 and  switch == "on" );
}
```

In the above example it is to be hoped that the invoked rule block, lowerTemp, will set the global variable temp to a new temperature, or set the global variable switch to "off".

## For loop rules

For-loop rules are looping rules with syntax and semantics equivalent to a Java for(;;) statement. The for section of the rule contains three components, the initialization list, the loop test expression, and the iterator list. If the test expression is true, the body of the for-loop repeats until the test expression is no longer true.

Syntax

```
 for ( initExprList ;  testExpr ;  iterExprList )  {
        <Action expression>+
} ;
```

Parameters

<initExprList>
Is a list of zero or more intialization expressions, with each expression separated by a comma and ending with a semicolon.

<testExpr>

Is a single Boolean test expression that is tested at the top of the loop. If the expression is true, then the body of the for-loop is processed. If the expression is false, then the processing of the rule halts.

<iterExprList>
Is a list of zero or more iteration expressions, with each expression separated by a comma.

<Action expressions>
Is a list of one or more action expressions separated by semicolons.

Example

```
void exampleRuleBlock() using Script {

  ForLoopExample1 :  for ( x = 0 ; x < 5 ; x = x+1 ) {
                         println("forLoop1 = ", x) ;
                     };

  ForLoopExample2 :  for ( x = 0, y = 3 ; x < 5 ; x = x+1, y = y-2 ) {
                         println("forLoop2 x= ", x) ;
                         println("forLoop2 y= ", y) ;
                     };

  ForLoopExample3 :  for ( x = 0 ; x < 5 ; x = x+1 ) {
                       println("forLoop3 = ", x) ;
                   };

   EndlessForLoop :  for (  ; ; ) {
                       println("Help ... I'm caught in an endless loop!") ;
                     };
}
```

In the first three examples, the for-loop will execute 5 times. In the last example, it will run forever.

## Predicate rules

Predicate rules include both single predicates, called predicate facts, and standard predicate rules with one or more antecedent clauses (the body) and a single consequent clause (the head). Like Prolog, Predicate rules in ARL use a period as the end of statement delimiter.

Syntax

<predicate>.  // Fact

<predicate> :- <predicate>+ | <Boolean expression>+ .  // Rule

Parameters

<predicate>
Is a predicate.

<clause>
Is a list of one or more predicates, Boolean expressions or assignment expressions, all separated by commas.

Example

```
ruleset predicateExample {

 predicates { male, female, father, mother, parent, son,
              daughter, brother }
 void init() {
     : setControlParameter(ARL.Goal, brother(milcah,X)) ;
 }

 void process() using predicate {
    : father(abraham, isaac).    // Abraham is the father of Issac
    : father(haran, lot).        // Haran is the father of Lot
    : father(haran, milcah).     // Haran is the father of Milcah
    : father(haran, yiscah).     // Haran is the father of Yiscah
    : male(isaac).               // Issac is male
    : male(lot).                 // Lot is male
    : female(milcah).            // Milcah is female
    : female(yiscah).            // Yiscah is female

    R1: son (X, Y) <- male(X) ,  father(Y, X).
    R2: daughter(X, Y) <- female(X) ,  father(Y, X).
    R3: brother (X, Y) <- male(Y)   ,  parent(P,X) ,  parent(P, Y) ,  X != Y
    R4: parent (X, Y) <- father(X, Y).
    R5: parent  (X, Y) <- mother(X, Y).

  }

 }
```

# Chapter 11. Inference Engines

This chapter presents details on the inference engines provided with the ABLE rule environment.

This chart shows the degree of support that each inferencing engine provides for specific Able Rule Language features. Any feature marked NS is Not Supported, meaning the inferencing engine tolerates the presence of the feature in a ruleblock but will not process it. These features will produce warnings. The presence of any feature marked NO will prevent a ruleset from executing so an error will be issued when the ruleset parses.

| Inference Engine | Assertion | if/then Conditional | if/then/else | Iteration | When/do | Predicate |
|---|---|---|---|---|---|---|
| Script | Yes | No | Yes | Yes | Yes | No |
| Fuzzy | Yes | Yes | No | No | No | No |
| Forward | Yes | Yes | No | No | No | No |
| Backward | Yes | Yes | No | No | No | No |
| PatternMatch | Yes | No | No | No | Yes | No |
| PatternMatchRete' | Yes | No | No | No | Yes | No |
| Predicate | Yes | No | No | No | Yes | Yes |

If/Then conditional rules can have only one consequent clause.

Examples ruleblocks for each inferencing engine

```
void init() { // using Fuzzy process method
        : setControlParameter(ARL.process, ARL.InferenceMethod, ARL.FuzzyAdd) ;
        : setControlParameter(ARL.process, ARL.AlphaCut, 0.1) ;
        : setControlParameter(ARL.process, ARL.CorrelationMethod, ARL.Product);
        : setControlParameter(ARL.process, ARL.DefuzzifyMethod, ARL.Centroid) ;
}

void init() { // using using Forward
     : setControlParameter(ARL.process, ARL.ControlStrategy, ARL.FIRE_ALL_RULES) ;
}

void init() { // using using PatternMatch
     // no control parameters
}

void init() { // using using PatternMatchRete {
    // no control parameters
}
```

```
    // often set in the process ruleblock because the goal changes during inferencing
    void process() using Backward {
          : setControlParameter(ARL.process, ARL.Goal, "customerType");
    }

    void init() { // using Script
        // no control parameters
    }

    void init() { // using Predicate
          : setControlParameter(ARL.process, ARL.Goal, son(X, John)) ;
    }

}
```

## Forward

| Inference Engine | Assertion | if/then Conditional | if/then/else | Iteration | When/do | Predicate |
|---|---|---|---|---|---|---|
| Forward | Yes | Yes | No | No | No | No |

The Forward inference engine processes Assertion and If-Then Conditional rules only. This is the lightest weight of the forward chaining algorithms and is adequate for small to medium sized rulesets.

There are three control strategies used by the inference engine. The default is called, FIRE_ALL_RULES, which repeated cycles and fires rules until no more can be fired.

The processing sequence is:

1) Process all Assertion rules in their declaration order.
2) Build a conflict set by selecting all triggered Conditional rules based on priority, specificity, and declaration order.
3) Select and fire the first rule in the conflict set.
4) Repeat until the conflict set is empty.

The other two control strategies, FIRE_ONE_RULE fires the first triggered rule that is encountered, FIRE_N_RULES will fire the first N triggered rules encountered. The processing sequence is

1) Process all Assertion rules in their declaration order.
2) Evaluate the Conditional rules in their declaration order (priority is ignored) and fire the first rule (or the first N rules) encountered which can be fired.

The control strategies are selected using the setControlParameter(ARL.*ruleBlockName*, ARL.ControlStrategy, ARL.FIRE_ALL_RULES).

## Backward

| Inference Engine | Assertion | if/then Conditional | if/then/else | Iteration | When/do | Predicate |
|---|---|---|---|---|---|---|
| Backward | Yes | Yes | No | No | No | No |

The Backward chaining engine processes Assertions and Conditional if-then rules using a goal-driven backward chaining algorithm without backtracking. The Goal control parameter must be set for this engine to run.

The processing sequence is to:

1) Process all Assertion rules in their declaration order.
2) Attempt to find a set of variable bindings that proves the goal variable or condition true by searching the And/OR graph represented by the Conditional rules.

The Goal control parameter specifies the goal variable for which the backward chaining inference engine is to solve. The Goal control parameter must be specified when using the Backward engine.

The Backward engine tried to find a value for the Goal variable by chaining backward through the rules while simultaneously finding values for any unbound variables. If a value cannot be determined from the rules, the Backward engine will try to invoke the askUser/2 function to prompt the user for a value. A default implementation of askUser is provided in the AbleGUILib library (which can be accessed by adding a **library com.ibm.able.rules.AbleGUILib;** statement to the ruleset). Once a variable is bound, it is never unbound by the Backward inference engine (i.e. there is no backtracking). If the same block of rules is to be used to find multiple goals, then any intermediate or final goal variables must be unbound by using the AbleRuleSet.resetVariable( “name”) or AbleRuleSet.resetVariables(new String[] { “var1”, “var2” } ) built-in methods.

Syntax

```
void process() using Backward {
   : setControlParameter(ARL.Goal, “<variableName>”) ;

}
```

Parameters

<variableName>
Is an identifier that names a previously defined variable.

Example

```
ruleset <nameOfRuleSet> {

 variables {
  String Diagnosis  ;   // note, no initial value so var is unbound
  Boolean Fever = true ;   // note, this variable is considered to be bound
 }

 void process() using Backward {
       : setControlParameter(ARL.Goal, "Diagnosis") ;
         // The backward chaining inference engine tries to determine the diagnosis.
   . . .
 }
```

## Predicate

| Inference Engine | Assertion | if/then Conditional | if/then/else | Iteration | When/do | Predicate |
|---|---|---|---|---|---|---|
| Predicate | Yes | No | No | No | Yes | Yes |

The Predicate chaining engine processes Assertions and Predicate rules using a goal-driven backward chaining algorithm with backtracking. The behavior of this engine is similar to a Prolog inferencing process. The Goal control parameter must be set for this engine to run.

The processing sequence is to:

1) Process all Assertion rules in their declaration order.
2) Attempt to find a set of variable bindings that proves the goal predicate(s) or condition true by performing a depth-first search of the Predicate rules.

The Goal control parameter specifies the goal predicate which the predicate inference engine is to solve. The Goal control parameter must be specified when using the Predicate engine. Results can be obtained using the getSolutionList() function provided by the AblePredicateLib library. The Predicate-specific library, com.ibm.able.rules.AblePredicate, should be included in rulesets. In general its divideBy, minus, plus, and times methods should be used instead of the equivalent operators /, -, +, and *.

Syntax

```
    void process() using Predicate {

     : setControlParameter(ARL.Goal,  father(John, X) ) ;
```

```
}
```

Parameters

predicateClause
Is a list, separated by either commas or ampersands, of zero or more predicate clauses. These clauses represent goals to be solved by the inference engine.

Example

```
ruleset PredicateExample {
   library com.ibm.able.rules.AblePredicateLib;

  predicates { male, female, father, mother, parent,  son, daughter, brother };

  // Who is the brother of Milcah?
  void process() using Predicate {
     : setControlParameter(ARL.Goal,  brother(milcah,X)) ;

     …
  }
}
```

## Fuzzy

| Inference Engine | Assertion | if/then Conditional | if/then/else | Iteration | When/do | Predicate |
|---|---|---|---|---|---|---|
| Fuzzy | Yes | Yes | No | No | No | No |

The Fuzzy inference engine processes Assertion and Conditional If-then rules using a multi-step forward chaining algorithm.

The processing sequence is to:

1) Process all Assertion rules in their declaration order.
2) Divide the rules in the ruleblock into subsets based on their consequent variable. For example all rules that refer to Temperature in their consequents are processed at the same time. A graph of these rule subsets is constructed such that the values of intermediate variables are computed before rules that reference those variables are processed.
3) Process the rule subsets in order, evaluating the antecedents of every rule in the subset using the selected inference method to combine the fuzzy evidence and to combine the consequent fuzzy sets.
4) Defuzzify the output variables using the selected defuzzification method.

There are four control parameters used by the Fuzzy inference engine. These include the fuzzy inference method, the alphaCut parameter, the correlation method, and the defuzzification method.

Fuzzy Inference Methods

FuzzyAdd
Specifies that the fuzzy solution set is updated by adding the minimum truth value of the consequent fuzzy region, bounded by 1.0. This method is generally used with CorrelationMethod(Product). See the AlphaCut, CorrelationMethod and DefuzzifyMethod options.

MinMax
Specifies that the fuzzy solution set is updated by using the maximum of the minumum truth value of the consequent fuzzy set. This method is generally used with CorrelationMethod(Minimise). See the AlphaCut, CorrelationMethod and DefuzzifyMethod options.

ProductOr
Specifies that the fuzzy solution set is updated by using 1 minus the product of (1 minus the truth value of the consequent fuzzy set) and (1 minus the predicate truth). This method is generally used with CorrelationMethod(Product). See the AlphaCut, CorrelationMethod and DefuzzifyMethod options.

AlphaCut

For rulesets with fuzzy clauses, specifies the threshold at which truth values become insignificant. If the truth value of any antecedent clause falls below the alphacut when the clause is evaluated, the inference engine stops evaluating the rule in which the clause appears.

If left unspecified, the default AlphaCut value is 0.1.

Syntax

alphaCut=<numericValue>

Parameters

<numericValue>
Is a number from 0.0 to 0.99. If set at 0.0, any clause that evaluates to anything greater than zero is essentially considered true to some degree. If set at 0.99, all clauses must evaluate to 1.0 (boolean true) to be considered true at all. Values between 0.05 and 0.2 may be considered typical, but of course it all depends on your fuzzy variables and their fuzzy set definitions.

Example

```
: setControlParameter(ARL.AlphaCut, 0.20) ;
```

CorrelationMethod

For rulesets with fuzzy clauses, specifies the manner in which a rule's consequent fuzzy region is correlated with the rule's antecedent fuzzy truth value.

If left unspecified, the default CorrelationMethod is Product.

Syntax

```
: setControlParameter(ARL.CorrelationMethod,  ARL.Product) ;
: setControlParameter(ARL.CorrelationMethod,   ARL.Minimum);
```

Parameters

Product
Specifies that the membership value of the consequent fuzzy region is the product of the fuzzy region and the truth of the premise. The effect is that the consequent fuzzy region is scaled, preserving its shape. This method is generally used with
ProductOr or FuzzyAdd inference methods.

Minimum
Specifies that the membership value of the consequent fuzzy region is the minimum of the fuzzy region and the truth of the premise. The effect is that the consequent fuzzy region is truncated at the truth of the premise, creating a plateau. This method is generally used with the MinMax inference method.

Example

```
void init() {
    : setControlParameter(ARL.process, ARL.InferenceMethod, ARL. FuzzyAdd) ;
    : setControlParameter(ARL.process, ARL.CorrelationMethod, ARL.Minimum) ;
}
```

DefuzzifyMethod

For rulesets with fuzzy clauses, specifies the manner in which a fuzzy set is turned into a crisp numeric value.

If left unspecified, the default DefuzzifyMethod is Centroid.

Syntax

```
void init {
        : setControlParameter(ARL.process, ARL.DefuzzifyMethod, ARL. Centroid) ;
        : setControlParameter(ARL.process, ARL.DefuzzifyMethod, ARL.MaxHeight) ;
 }
```

Parameters

Centroid
Specifies that a fuzzy set's crisp value is calculated as the weighted mean (center of gravity) of the fuzzy region. Also known as composite moments.

MaxHeight
Specifies that a fuzzy set's crisp value is calculated from the point that has the highest truth value. Also known as composite maximum.

Example

```
void init() {
     : setControlParameter(ARL.process, ARL.DefuzzifyMethod, ARL.Centroid) ;
 }
```

## Weighted comparison expressions

The fuzzy inference engine supports optional weights as suffixes to Boolean comparison expressions. A WEIGHT is a number between 0.0 and 1.0. Weights are ignored by other inference engines, and fuzzy supports weights only for comparison operators. In fuzzy inference engines weights can keep a Boolean comparison from wildly skewing a fuzzy result. When omitted, a Boolean expression's default weight is 1.0.

| | |
|---|---|
| == | Tests if operandA is equal to operandB |
| > | Tests if operandA is greater than operandB |
| >= | Tests if operandA is greater than or equal to operandB |
| < | Tests is operandA is less then operandB |
| <= | Tests if operandA is less than or equal to operandB |
| != | Tests if operandA is not equal to operandB |

Multiple comparisons may be connected with a comma.

Example

```
void process() using Fuzzy {
```

```
    : if (a > 10 [.5], b > 10[.8]) println("Double digit increases: a ="+a+ "b="+b);
}
```

## PatternMatch

| Inference Engine | Assertion | if/then Conditional | if/then/else | Iteration | When/do | Predicate |
|---|---|---|---|---|---|---|
| PatternMatch | Yes | No | No | No | Yes | No |

The PatternMatch engine is a forward chaining engine that processes Assertion and When/Do pattern match rules. This engine makes use of working memory. This is a medium weight forward chaning algorithm suitable for use with a moderate number of rules or for inferencing over a small number of objects.

The processing sequence is to:

1) Process all Assertion rules in their declaration order.
2) Build a conflict set by selecting all triggered Conditional rules based on priority, specificity, and declaration order.
3) Select and fire the first rule in the conflict set.
4) Repeat until the conflict set is empty.

## PatternMatchRete'

| Inference Engine | Assertion | if/then Conditional | if/then/else | Iteration | When/do | Predicate |
|---|---|---|---|---|---|---|
| PatternMatchRete' | Yes | No | No | No | Yes | No |

The PatternMatchRete' engine is a forward chaining engine that processes Assertion and When/Do pattern match rules. This engine makes use of working memory and constructs a Rete' network that caches partial matches. This is the heaviest duty of the forward chaining algorithms and should be used for very large rulesets or for inferencing over very large numbers of objects.

The processing sequence is to:

1) Process all Assertion rules in their declaration order.
2) Build a conflict set by selecting all triggered Conditional rules based on priority, specificity, and declaration order.
3) Select and fire the first rule in the conflict set.
4) Repeat until the conflict set is empty.

## Script

| Inference Engine | Assertion | if/then Conditional | if/then/else | Iteration | When/do | Predicate |
|---|---|---|---|---|---|---|
| Script | Yes | No | Yes | Yes | Yes | No |

The Script engine processes Assertion, If-then-else, Iteration (do/while, do/until, while/do, for-loop), and when/do rules sequentially in declaration order. Priority is ignored by this engine.

# Chapter 12. Functions and Methods

The ABLE rule language provides two mechanisms for calling external Java code: functions and methods. In this chapter we describe how functions and methods can be referenced and called from rules.

Functions allow rules to interact with the outside world. For example, a rule can invoke a function to read a gauge, such as a thermometer, and use the returned value to set a variable. Or, a function can be used notify some other program about some action it should take, causing some side-effect. Or a function might be used to send an e-mail. The uses of functions are many and varied.

There are several ways to make functions available to rules. First of all, there are built-in functions provided by all rulesets. These are described in Built-In Functions below. Another way is to load Java classes into a ruleset so that all the public methods in the imported classes become accessible to your rules as user-defined functions. This technique is described below in Function Libraries. And yet another way is to use Externally Attached Functions, also described below.

## Built-in functions

ABLE provides a small set of built-in functions that are always available in all rulesets. The built-in functions listed below may be referenced in any rule without any additional work on the part of the rule writer. There are also a larger number of ruleset methods which can be used in rules by referencing the ruleset (this) and the method name.

Note that built-in functions always take precedence over imported function library functions of the same name and arity, but that externally attached functions can override built-in and imported functions.

`checkTimePeriodPreConditions(Calendar currentTime)`

Evaluates each rule in every ruleblock against the time period preconditions. If the rule is active and if any of the time periods listed in the preconditions are satisfied, the rule is enabled.

`getControlParameter(String name)`

Returns the value of the named control parameter on the current rule engine. Use this form to retrieve parameters from within a ruleblock.

`getControlParameter(String ruleblockName, String name)`

Returns the value of the named control parameter on the rule engine in the specified ruleblock. Use this form to access parameters from outside the named ruleblock.

`invokeRuleBlock(String ruleBlockName)`

Invokes the specified rule block, which must be declared in the current ruleset.

`getVariable(String variableName)`

Returns a reference to the specified variable.

`print(String message)`
Calls Java System.out.print(String). Writes a message to the Java console.

`println(String message)`
Calls Java System.out.println(String). Writes a message to the Java console.

`returnFromRuleBlock(Object value)`
Exits the ruleblock and returns to the caller ruleblock with the specified value.

`setControlParameter(String name, Object value)`
Set the value of the named control parameter on the current rule engine. Use this form for dynamic parameters within a ruleblock.

`setControlParameter(String ruleblockName, String name, Object value)`
Set the value of the named control parameter on the rule engine in the specified ruleblock. Use this form for parameters which do not change within the named ruleblock to avoid the overhead of setting them each time that ruleblock inferences.

`trace(String message)`
Writes a message to the ABLE trace logger for display in a GUI.

`traceFormat(String formatString, Object[] replacementVars)`
Calls Java System.out.println(String). Writes a message to the Java console.

Please refer to the JavaDoc on the AbleRuleSet class, and the AbleWorkingMemory class for additional methods which can be called on the ruleset and working memory objects respectively.

## Function Libraries

**Note that with the Java syntax introduced with ARL 1.4, the value of function libraries is reduced. Methods in existing libraries are unchanged with 1.4 but expect their removal at a later date. Using import to access external Java classes provides more functionality (better signature mapping) and better performance than using Function libraries.**

The entire set of public methods in any arbitrary Java class can be made available to a ruleset all at once by simply loading the function library into the ruleset. The Java class becomes a function library of the ruleset, and means that you can write your own specialized set of functions to meet your unique needs. The only restriction on Java classes you import is that, if you intend to serialize your ABLE ruleset, any library classes must also be serializable. See the library statement for details.

If a library function has the same name and arity as a built-in function, the built-in function takes precedence. Note also that externally attached functions can override both built-in and imported functions.

**Note: ABLE used to provide AbleBeanLib and AbleWorkingMemoryLib built-in function libraries for use in rulesets. Most of these functions are now available to be called directly as methods on the ruleset using this.XYZ() notation or on the built-in working memory variable using wm.assert(), wm.retract() etc. The AbleDebugLib is no longer required to be used in rulesets to enable debugging. The functions in the AbleMathLib can now be replaced by importing the java.lang.Math class. The AbleCalendarLib and AbleStringLib have now been removed from ABLE. Some of the methods have been moved to the ARL.java and are available as static methods.**

ABLE provides these optional function libraries:
com.ibm.able.rules.AbleGUILib
com.ibm.able.rules.AblePredicateLib

The optional libraries are highly specialized and require an explicit library statement to use their methods.

AbleGUILib is used by the backward chaining inference engine and contains methods (askUser()) to allow an inference engine to prompt a user to supply values of variables that can not be determined through inferencing. The askUser() function is useful for testing and debugging by prompting a user for values. If you run your rule agents in an environment that allows an inference engine to interact with an end user, simply import this library to activate potential end user interaction. If you run your rule agents in an environment where it is not possible to interact with end users, do not import this library, and the inference engine will make no attempt to query a user for unknown variable values.

AblePredicateLib is useful only to predicate logic rulesets and provides a few mathematical functions designed to work with the predicate inference engine. You have to explicitly import this library in order to use its methods.

## Externally Attached Functions

Built-in and imported functions must exist at the time a ruleset is compiled and they occupy space in the compiled ruleset. Externally attached functions are functions that are added to a ruleset at runtime under control of some external Java program, just before the ruleset is actually processed, and the functions occupy space somewhere else in the Java virtual machine. Because the ABLE Rule Language compiler needs to know the names of these functions, however, you must use the functions statement to specify, by name and arity, those functions that will be attached to the ruleset at runtime. Then, if declared externally attached functions are not available when the ruleset is processed, a runtime exception is thrown.

ABLE's examples/rules directory contains the source code of an example program that attaches external functions to a ruleset and then processes the ruleset. The program's name is SampleSensorEffector.java.

Note that externally attached functions always take precedence over built-in and imported functions of the same name and arity, so you can use external functions to override the behavior of built-in and imported functions.

## Functions Statement Syntax

Use this optional statement to declare that the specified named methods of arbitrary Java objects will be available to be called from within rules during inferencing. Any methods named in the functions statement or statements can be referenced in any rule. The named methods must be programmatically attached to the ruleset before the ruleset is processed. To attach a user-defined function to a ruleset, a Java program must first create an AbleUserDefinedFunction object, and then call a ruleset's addUserDefinedFunction() method. An example of this coding technique can be found in the SampleSensorEffector Java program provided in ABLE's examples/rules directory. If a rule references a user-defined function that has not been attached to the ruleset before processing takes place, a runtime exception will occur. Rules invoking these externally attached methods must pass in the proper number of arguments, and the methods must return the proper data type expected by the rule.

Note that functions declared in the functions statement always take precedence over built-in and imported functions of the same name and arity. This means, for example, that externally attached functions can be used to override the behavior of built-in functions.

Functions is an optional statement and may be omitted altogether.

Syntax

```
functions { <name/arity>* };
```

Parameters

<name/arity>
Is a list of zero or more identifiers that identify methods that can be referenced in rules.

Note that each entry is made up of a function name suffixed with a slash (/) and the number of arguments the function expects. Beyond number-of-argument checking no other checking, such as argument type checking, is done. See the examples below.

Note also that variables are passed to functions by value, not by reference.

Examples

```
ruleset <nameOfRuleSet> {

 variables {
  <Variable Declaration Statement>+  // One or more statements

 }

 inputs {} ;
 outputs {} ;


 functions {a/0, a/1, b/1 }    // Three externally attached functions available at runtime:
                    //   method 'a' that takes no args,
                    //   method 'a' that takes one arg and
                    //   method 'b' that takes one arg.

   . . .
  }
```

Example rules that use the above declared user-defined functions:

```
 r1: myVar = a();
```

The result of calling function a/0 is placed into variable myVar.

```
 r2: otherVar = b(someVar);
```

The result of calling function b/1 with argument someVar is placed into variable otherVar.

```
 r3: if (myVar == a())  then...
```

myVar is tested against the value returned by function a/0.

```
 r4: if (true)  then   y = a(someVar);
```

If the condition is true  y = a(someVar);  the result of calling function a/1 with argument someVar is placed is placed into variable y.

## Externally attached functions

## Function Arguments

Arguments to functions can be the name of a variable declared in the ruleset (passed by value), literal values, variable fields, variable methods, and other functions:

```
Double Temperature = new Double(98.6); // Declare numeric variable

Double SystemId = new Double(0); // Declare numeric variable

functions { getTemp/1 } // Declare function that takes one arg

functionRule1 : Temperature = getTemp(SystemId); // is valid
functionRule2 : Temperature = getTemp(1234); // is valid
```

The number and type of arguments to a function must match what is expected by the actual Java method; if either is not what is expected, an exception will occur:

```
functionRule3 : Temperature = getTemp(SystemId);
```

This may or may not be valid depending on the number of arguments expected by the method represented by the function getTemp.

At compile time, the ARL parser tries to find a method that matches the expected values of the arguments. ARL built-in types such as Boolean, Double, Float, Integer, and Long can match to the corresponding Java types or the corresponding Java primitive types. Care must be taken when signatures are very similar (example using double and Double) to make sure that the correct method is being called.

## Function Return Value

Functions may return any Java primitive or Object. However, the type must be of a kind expected by the rule, or the results may be unpredictable:

```
Boolean  testValue = new Boolean(false);        // Declare boolean variable

functions {myTest/1} ;  // Declare function that takes one arg

aRule : if testValue == myTest("123") ... // OK if myTest returns  a boolean or Boolean
```

Theoretically, myTest() may return a string, but only if the string is either "true" or "false" (case insensitive); any other string will cause a runtime exception. myTest() may also return a number,

where 0.0 is considered false and any other number is considered true. Such uses, however, are to be thoroughly discouraged as they can lead to unexpected errors.

# Built-in methods

There are several built-in variables that are accessible from every ruleset, including **this** which is a reference to the AbleRuleSet bean itself, and **wm**, which is a reference to the current AbleWorkingMemory object. In addition, the **ARL.java** built-in class provides many constant definitions as well as several static utility methods. All of the public methods on these Java classes are available for use inside a ruleset. This large set of built-in methods is available to access and manipulate the ruleset object itself during rule processing. While this is powerful, it is also somewhat dangerous and so must be used with caution.

## AbleRuleSet methods

The most commonly used methods are available as built-in functions. In this section we list some of the most useful AbleRuleSet bean methods that may be referenced in any rule against the **this** built-in variable or against an AbleRuleSet typed variable. See the AbleRuleSet bean JavaDoc for a complete list of public methods.

### Tracing and Logging methods

void setBaseTraceLevel(int level)

int getBaseTraceLevel()

void setInferenceTraceLevel(int level)

int getInferenceTraceLevel()

void setParseTraceLevel(int level)

int getParseTraceLevel()

### Meta-rule methods

These methods allow you to get references to and in some case manipulate ruleset objects during rule processing:

Hashtable getVariables()

AbleVariable getVariable(String name)

AbleRule getRule(String ruleLabel)

boolean getRuleEnabled(String ruleLabel)

void setRuleEnabled(String ruleLabel, boolean state)

boolean getRuleFired(String ruleLabel)

double getRulePriority(String ruleLabel)

void setRulePriority(String ruleLabel, double priority)

AbleRuleBlock  getRuleBlock(String ruleBlockName)

AbleInferenceEngine getInferenceEngine()

AbleInferenceEngine getInferenceEngine(String ruleBlockName)

AbleWorkingMemory getWorkingMemory()

AbleWorkingMemory getWorkingMemory(String ruleBlockName)

**NLS messages and prompts methods**

ResourceBundle getResourceBundle()
Returns the resource bundle object (if set).

String getResourceBundleName()
Returns the resource bundle name (if set).

void setResourceBundle(String name)
Sets the resource bundle name, containing NLS messages associated with the ruleset.

void setRulePrompt(String name, String promptMsg)
Set the prompt to be used for this rule.

void setVariablePrompt(String name,  String promptMsg)
Set the prompt to be used for this variable.

## AbleWorkingMemory methods

These methods can be called on the built-in working memory variable, **wm**, or on a typed variable that references a working memory object. Please refer to the JavaDoc on the AbleWorkingMemory class for additional methods that can be called.

Note that each ruleblock in a ruleset has its own working memory instance. They can be made to share a single instance by assigning a value to the built-in **wm** variable using the AbleRuleSet. getWorkingMemory("ruleblockName") method to retrieve the desired working memory instance.

assert(Object fact)
Add a fact to the working memory object

assertAll(List facts)
Add a list of facts to the working memory object

clear()
Remove all facts from the working memory

retract(Object fact)
Remove a fact from the working memory object

retractAll(List facts)
Remove a list of facts from the working memory object

modify(Object fact)
Tell the working memory object that a fact has changed.

exists(Object fact)
Test whether the fact is contained in the working memory object.

find(Selector query)
Return a single fact that matches the constraints specified by the Selector

findAll(Selector query)
Returns a Collection of facts that match the constraints specified by the Selector

findAllInstances(String className)
Returns a Collection of facts that are of the specified className.

getClasses()
Returns an Enumeration of all classes for which instances exist in working memory.

isEmpty()
Returns true if the working memory is empty, false otherwise

## ARL methods

The following static methods are available by referencing the ARL.java class. Methods can be invoked by **ARL.<methodName>(args*).**

invokeCommand(String command)
Invoke the command provided and prepend any necessary operating system arguments. Useful for system independant rulesets.

invokeCommandOS(String command)
Invoke the command provided as is. You provide any operating system arguments in the command.

loadVarsFromPropertiesFile(AbleRuleSet theRuleSet, String theFileName)
Load a property file and set the value of any variables with the values taken from the properties file.

setAbleEventProcessingStates(AbleBean theAbleBean, **boolean** processingEnabled, **boolean** postingEnabled)
Set the ABLE event processing states on the ruleset bean. Both need to be set to true to enable event processing by the ruleset. If defined, the processAbleEvent() ruleblock will be called to process the incoming event.

getSystemProperty(String thePropertyName)
Returns the value of a system property.

getSystemProperty(String thePropertyName, String defaultValue)
Returns the value of a system property. If the system property is not set, the defaultValue is returned.

setSystemProperty(String thePropertyName, String theValue)
Set the value of a system property.

Object processRuleSet(AbleRuleSet theRuleSet, Object theInputBuffer)
Invoke a sub-ruleset passing an Object[] as the input buffer Object and receive an Object[] as the result.

Vector getBeanOutput(AbleBean theBean)
Retrieve the specified bean's output buffer as a Vector of Objects.

fine(AbleRuleSet ruleset, text)
Write low level text to the ruleset's trace logger. Only the most significant of messages should use this level. Equivalent to `trace(text).`

finer(AbleRuleSet ruleset, text)
Write medium level text to the ruleset's trace logger. Use this for typical messages.

finest(AbleRuleSet ruleset, text)
Write high level text to the ruleset's trace logger. Use this for very detailed messages that typically would not need to be shown.

info(text)
Write informational text to the global message logger.

severe(text)
Write severe error text to the global message logger.

warning(text)
Write warning text to the global message logger.

# Chapter 13. Templates

In this chapter, we describe the template support provided by the ABLE rule language.

The purpose of the template support is to allow the ruleset author to write an application, and designate certain parts of that application as available for later customization by business or application domain experts. Complex internal rule logic can be hidden and protected from changes by these domain experts, while simplified domain-specific user interfaces can be provided for these users.

The ABLE Rule Language template design allows ruleset authors to specify rule-level and ruleset-level templates. From a ruleset author's perspective, the template support is straightforward; simply add the **template** modifier to rulesets, variables or rules to designate them as templates. This means any ruleset, any built-in or imported data types can be used as template variables, and any ARL rule type can be templatized.

Rules that are generated from templates can be saved as ARL text files or as ARML XML documents and then re-loaded and edited using the original template replacement values. The meta-data associated with rule template usage is saved in the ruleset in a special ruleblock called initRuleTemplates(). This ruleblock is automatically processed to restore the template context for re-editting of generated rules.

At the rule template level, a ruleset author can write templates that allow:
1. customization of rule label (required). Each rule must have a unique label.
2. customization of rule priority (optional)
3. customization of rule preConditions (optional)
4. customization of variables based on constraints. The template author can set constraints via the choice of template variable data types and values (Categorical, Discrete, Continuous, etc.).
5. customization of rule logic through use of Expression variables ... for example:

    **if ( cond ) then { action } ;**

    where **cond** is an Expression variable or **action** is an Expression variable ..
    This is a more powerful, less constrained type of rule template.

A ruleset author can create rulesets and templates using the ARL Swing RuleSet editor or the WebSphere Studio Application Developer ARL Editor plugin. There are no external artifacts required to use templates. The ARL compiler parses and generates an AbleRuleSet bean with the template information contained in it. An AbleRuleSet bean with templatized components can be customized with PC client or web-based user interfaces via the AbleRuleSet bean template APIs.

The following steps are used by a ruleset author to create and use templates:
1) Create an ARL ruleset source file.
2) Define variables, ruleblocks, rules, etc.
3) Define template variables, rules, or rulesets

4) Compile the ruleset into a run-time AbleRuleSet bean
5) Edit .. allow domain experts to author new rules

```
Ruleset  myRuleTemplateExample {
   import  com.ibm.myclass.Customer;

  variables {
              Customer    customer = new Customer() ;  // myclass type
    template  Categorical customerLevel = new Categorical(new String[]{"gold",
"silver", "platinum"});
    template  String      salesMsg = new String("Thank you for shopping IBM");
    template  Continuous  discountValue = new Continuous(0.01, 0.50); // allow range
from 1% to 50%
              Double        discount = new Double(0.0) ;
  }

   inputs {  customer } ;
   outputs { discount } ;

  void process() {
     Rule1: if (a > b) then println("regular old rule") ;
     Rule2:  if (a <= b) then println("another regular old rule") ;

     template myRuleTemplate1: if ( customer.level ==  customerLevel )
                               then  { discount = discountValue  ;
                                       println( salesMsg ) ;  }
  }
}
```

```
Ruleset  myRuleTemplateExample {
   import  com.ibm.myclass.Customer;
  variables {
              Customer    customer = new Customer() ;  // myclass type
    template  Categorical customerLevel = new Categorical(new String[]{"gold",
"silver", "platinum"});
    template  String      salesMsg = new String("Thank you for shopping IBM");
    template  Continuous  discountValue = new Continuous(0.01, 0.50); // allow range
from 1% to 50%
              Double        discount = new Double(0.0) ;
  }
    inputs {  customer } ;
    outputs { discount } ;
  void process() {
     Rule1: if (a > b) then println("regular old rule") ;
     Rule2:  if (a <= b) then println("another regular old rule") ;
     myNewRule : if (customer.level == "gold")             <-- NEWLY GENERATED RULE
                 then { discount = 0.10;
                        println("Happy Holidays";
                      }
     template myRuleTemplate1: if ( customer.level ==  customerLevel )
                               then  { discount = discountValue  ;
                                       println( salesMsg ) ;  }
  }

    // this ruleblock is used to allow re-editting, roundtripping of templates

   void initRuleTemplates()  {

     genmyNewRule: this.setRuleTemplateInfo("myRuleTemplate1", new Object[] {
"comment", "myNewRule", "gold", "0.10", "Happy Holidays");
   }
}
```

```
template ruleset  myRulesetTemplateExample  {
   import  com.ibm.myclass.Customer;

  variables {
              Customer    customer = new Customer() ;  // myclass type
    template  Categorical customerLevel = new Categorical(new String[]{"gold",
"silver", "platinum"});
    template  String          salesMsg = new String("Thank you for shopping IBM"):
    template  Continuous  discountValue = new Continuous(0.01, 0.50); // allow range
from 1% to 50%
                Double        discount = new Double(0.0) ;
    template    String          msg1 ;
    template    String          msg2 ;
  }

   inputs {  customer } ;
   outputs { discount } ;

   void process() {
     Rule1: if (a > b) then println(msg1) ;
     Rule2:  if (a <= b) then println(msg2);

     template myRuleTemplate1: if ( customer.level ==  customerLevel )
                             then  { discount = discountValue  ;
                                     println( salesMsg ) ;  }
  }
}



/** example rule set template */
ruleset  rulesetTemplate1 {
   import  com.ibm.myclass.Customer;

  variables {
                    Customer    customer = new Customer() ;  // myclass type
    template  Categorical customerLevel = new Categorical(new String[]{"gold",
"silver", "platinum"});
    template  String          salesMsg = new String("Thank you for shopping IBM"):
    template  Continuous  discountValue = new Continuous(0.01, 0.50); // allow range
from 1% to 50%
                    Double        discount = new Double(0.0) ;
    template    String          msg1 ;
    template    String          msg2 ;
  }
   inputs {  customer } ;
   outputs { discount } ;

   void process() {
     Rule1: if (a > b) then println("set template var one") ;
     Rule2:  if (a <= b) then println("set template var two") ;
     template myRuleTemplate1: if ( customer.level ==  customerLevel )
                                 then  { discount = discountValue  ;   println(
salesMsg ) ;  }
  }
}
```

# Part 3. Programmer's Reference

This section of the document deals with aspects of the ABLE Rule Language environment for Java programmers. This includes integration of ABLE rules into Java applications, to programmatically create rulesets and rules using the AbleRuleSet bean APIs, as well as extending the ABLE rule environment through development of inference engines.

## Chapter 14. AbleRuleSet bean

The AbleRuleSet bean follows the standard AbleBean processing pattern. The bean is first instantiated. You parse a ruleset file using one of the parseFrom() methods. You initialize the bean by calling the init() method. Then you can ask the bean to process data by calling the process() method.

There are several ruleblocks with special names. For example, if you code an init() ruleblock, it will be called when the bean init() method is invoked. You can code a processTimerEvent() ruleblock to handle processing of timer events, processAbleEvent() ruleblock to handle AbleEvent processing, and a catch() ruleblock to handle any exceptions which occur during process().

A call to the AbleRuleSet bean process() method will trigger the evaluation of up to three ruleblocks. A preProcess() ruleblock (if defined) will be called first. A process() ruleblock (required) is then called to perform the main process logic. When the process() ruleblock is complete, the postProcess() ruleblock (if defined) will be called.

To create and work with a ruleset in a Java program, follow these steps:

**1. Create the ruleset object:**

```
AbleRuleSet myRuleSet = new AbleRuleSet("Name");
```

Customize the ruleset; that is, load it with variables and rules. The best way to do this is to tell the ruleset to instantiate itself from some source file written in either Able Rule Language or XML:

```
myRuleSet.parseFromARL("sourceFile.arl");
myRuleSet.parseFromXML("sourceFile.arml");
```

where sourceFile.arl is a source rule file created with the Able RuleSet Editor or by hand using any text editor. You can also customize the ruleset programmatically by using the ruleset's API to dynamically create and add variables, ruleblocks and rules.

**2. Initialize the ruleset:**

```
myRuleSet.init();
```

The init() method must be called immediately after a ruleset is parsed or when its structure is changed programmatically.

Assume ruleset has 4 variables listed in its inputs{} list as follows :

```
inputs { myBool, myDouble1, myString, myDouble2 } ;
```

Construct an Object[] of data to pass to the ruleset:

```
Object[] lclInBuffer = Object[4];
lclInBuffer[0] = new Boolean(true);
lclInBuffer[1] = new Double(9.9);
lclInBuffer[2] = "some string data";
lclInBuffer[3] = new Double(88.88);
```

**3. Process the ruleset:**

```
Object[] lclOutBuffer = myRuleSet.process(lclInBuffer);
```

Access the output of the ruleset, if any:

```
for (int i=0; i<lclOutBuffer.length; i++) {
  System.out.println("!! Output element[" + i + "]: <" + lclOutBuffer[i] + ">");
}
```

At this point you may change the data in the input buffer and call process() again, repeating the cycle as you wish. The ruleset is automatically reset (all rules and non-static variables are set to initial states) when you subsequently call the process() method.

The above code snippets demonstrate working with a ruleset at its simplest. An example Java program, SampleSensorEffector.java, is provided in Able's examples/rules directory and clearly shows these techniques. Of course, there are many other ways to work with rulesets, such as wiring them to input data sources and filters, sending the output to ABLE agents, and so on.

# Chapter 15. WorkingMemory APIs

This chapter describes the AbleWorkingMemory applcation programming interfaces. These APIS are essentially identical to what you can do via rules using direct method calls on the **wm** built-in variable.

assert(Object fact) - add a fact to working memory

assertAll(List facts) – add all facts in the List to working memory

retract(Object fact) – remove a fact from working memory

retractAll(List facts) – remove all facts in the List from working memory

modify (Object fact) – remove and then add a fact to working memory

Object find(Selector query) – return a single fact that meets the constraints in the query

Collection findAll(Selector query) – return a collection of facts that meet the constraints in the query

clear() – remove all facts from the working memory

boolean exists(Object fact) – returns true if the fact is in the working memory, false otherwise

# Chapter 16. Template APIs

This chapter describes the ABLE template APIs.

The ABLE Rule Language template design allows ruleset authors to specify rule-level and ruleset-level templates. The ARL template design introduces two new classes to ABLE, the AbleRuleSetTemplate and the AbleRuleTemplate. The primary template APIs are provided by the AbleRuleSet bean. User interfaces can extract and set information on the template objects and then use the AbleRuleSet apis to instantiate the new rulesets or rule objects.

## Ruleset templates

An entire ABLE ruleset can be used as a template to generate new rulesets. To enable this, the template modifier needs to be placed before the ruleset and one or more variables referenced in the ruleset have to use the template modifier.

The following steps are required to customize a ruleset template and generate a new ruleset:

1. get the ruleset template (getRuleSetTemplate())
2. get the template vars from it (getTemplateVars()), and set their values using var.setStringValue("value")
3. call createRuleSetFromTemplate(template) to generate a new ruleset

```
AbleRuleSetTemplate getRuleSetTemplate();
AbleRuleSet createRuleSetFromTemplate(AbleRuleSetTemplate theTemplate);
```

## Rule templates

A single ABLE rule can be used as a template to generate new rules. To enable this, the template modifier needs to be placed before the rule label and one or more variables referenced in the rule have to use the template modifier.

The following steps are required to customize a rule template and generate a new rule:

1. get rule templates in the ruleset by calling AbleRuleSet methods (getRuleTemplates()) or single one (getRuleTemplate(name))
2. display list, let user select one
3. get the template vars from the selected rule template (getTemplateVars()), and set their values (using var.setStringValue())
4. call addRuleFromTemplate(template) to create and add it to ruleset

### Edit a Generated Rule

The following steps are required to edit a rule that was previously generated from a template:

1. from a rule, getTemplateFromRule(ruleLabel)
   OR from a template, getGeneratedRulesFromTemplate(templateLabel)
4. from a rule, use getTemplateValues() to get values used to gen the rule, edit as before
5. Call replaceRuleFromTemplate(template) to replace the existing rule in the ruleset

## Roundtrip a ruleset that contains generated rules

1. saveAsARL(filename) or saveAsXML(filename)
2. then create AbleRuleSet bean and reload using parseFrom(filename)
3. call init() on the ruleset
4. call initRuleTemplates(), recreates all rule templates, and processes the initRuleTemplates() ruleblock which re-establishes generated rule refs.

```
// return all rule templates define in ruleset
public Vector getRuleTemplates();

// get single rule template defined in the ruleset
public AbleRuleTemplate getRuleTemplate(String theRuleName);


in AbleRule  …

public Object[] getTemplateVars(AbleRuleSet theRuleSet)  ;
```

1. parse ruleset, original rules and any generated rules get parsed as usual
   the initRuleTemplates() block is there but is not processed
   Absolutly no overhead if template editing and apis are not used…
2. someone wants to edit using templates, they call initRuleTemplates() ; and we
   A. create the rule templates and put them in a myTemplateList
   B. call the initRuleTemplates() block which has a set of setRuleTemplateInfo() rules, with template name and args … used to set members on generated rules?

3. someone wants to edit using templates, call getRuleTemplates() or getTemplate(name)
4. The ruleTemplate contains a ref to the template rule, a list of all rules generated from the template, a list of all args used to gen those rules.
5. User calls getTemplateVars() … to display var prompts, etc. can call setStringValue() to qualify the current value, if ok, calls addRuleFromTemplate() , replaceRuleFromTemplate() to actually gen the new rule and add/replace the meta-rule. Call reset() on the template to reset the template vars to initial values.

If someone wants to, they can create or replace rules from the ruleset by using the addRuleFromTemplate(name, args) or replaceRuleFromTemplate(name, args) APIs

initRuleTemplates() ;  // creates template objects and invokes initRuleTemplates() ruleblock

```
getRuleSetTemplate();
createRuleSet(AbleRuleSetTemplate theTemplate);

getRuleTemplates()
getRuleTemplate(String theTemplateName) ;
getRuleTemplateFromRule (String theRuleName);

getGeneratedRuleLabels(String theTemplateName);
getGeneratedRules(String theTemplateName);

// use the Template as the input parm
addRuleFromTemplate(AbleRuleTemplate theTemplate); // template vars are set
replaceRuleFromTemplate(AbleRuleTemplate theTemplate); // temp vars are set

// use the template name and string replacement vars as input parms
// could also contain AbleRd objs which resolve to string repl. vars
addRuleFromTemplate(String theTemplateName, Object[] theVarValues) ;
replaceRuleFromTemplate(String theTemplateName, Object[] theVarValues);
```

```
AbleRuleSet rs = new AbleRuleSet("example") ;
 rs.instantiateFrom("example.arl") ; // parse and load the ruleset
 rs.init() ;
 rs.initRuleTemplates() ;  // get ruleset ready to use templates
 AbleRuleTemplate lclTemplate = rs.getRuleTemplate("myRuleTemplate1") ;
 Object[] lclTemplateVars = lclTemplate.getTemplateVars() ;

 // bind the template variables with replacement values
 for (int i=0 ; i < lclTemplateVars.length ; i++)  {
      AbleVariable lclTemplateVar = (AbleVariable)lclTemplateVars[i] ;
      String newValue = "XYZ" ;
      // prompt user for new value then set value on the template variable
     lclTemplateVar.setStringValue( newValue ) ;
  }

 // generate  the new rule object from template (using bound template vars)
 AbleRule newRule =  rs.addRuleFromTemplate(lclTemplate) ;
```

1. use
   AbleRuleSetTemplate template = AbleRuleSet.getRuleSetTemplate()

2. retrieve the template variables (Vector) from the template  using
      Object[] vars =  template.getTemplateVars()

3. set the template variable values subject to constraints
   (provide UI to customize values)
        var.setStringValue("value") ;

4. create the new ruleset instance by invoking:
      AbleRuleSet.createRuleSetFromTemplate(template)

To retrieve rules created from a specific template:
    Vector getGeneratedRulesFromTemplate(String templateName);

1. use  AbleRuleTemplate template = AbleRuleSet.getRuleTemplates();
    or   AbleRuleTemplate template = AbleRuleSet.getRuleTemplate(name);

2. retrieve the template variables from the template
   using  Object[] vars  = template.getTemplateVars() ;

3. set the template variable values subject to constraints
   (provide UI to customize values)
   using   var.setStringValue("value");

4. create the ruleset or rule instance by invoking:
      AbleRule rule = AbleRuleSet.addRuleSetFromTemplate(template)
  or AbleRule rule = AbleRuleSet.replaceRuleFromTemplate(template)

    APIs that can be used from rules to create rules ...
      AbleRuleSet.addRuleFromTemplate("templateName", Object[] values)
      AbleRuleSet.replaceRuleFromTemplate("templateName", Object[] values)

 :  this.addRuleFromTemplate("temp2", new Object[] { "comment", "label", "apples"});

AbleRuleSetTemplate(AbleRuleSet) ;   // ctor

  String getName();    // return template name
  String getComment() ;  // return template description
  String getArlString() ;  // returns source ARL string

  AbleRuleSet createInstance() ;   // create new object
  Object[] getTemplateVars() ;   // returns list of template vars
  void   resetTemplateVars();  // resets all template vars to initial values

AbleRuleTemplate(AbleRule) ;   // ctor

```
String getName();    // return template name
String getComment() ;  // return template description
String getArlString() ;  // returns source ARL string

AbleRule createInstance() ;   // create new rule object
AbleRule replaceInstance();  // create new rule, replace prior one
Object[]   getTemplateVars() ;   // returns list of template vars
void       resetTemplateVars();  // resets all template vars to initial vals

void       addGeneratedRule(rule)
void       removeGeneratedRule(rule)
Vector     getGeneratedRules()
```

# Definition of Terms

Rule – an elementary statement in the ABLE rule language. A rule header has an optional label or identifier, an optional list of preconditions, and an optional priority. The rule body can be one of several rule types including if-then rules, predicate rules, and pattern match rules.

RuleSet – a collection of data, ruleblocks and rules

Expression – a set of literals and variables combined using unary or binary operators resulting in a Boolean, Fuzzy, or Numeric result

Clause – a Boolean expression on the left-hand side or antecedent of an if-then rule.

Variable – A named object that can hold a values of a specific data type

Function – an operation with zero or more arguments

Method – an operation with zero or more arguments defined on an Object

DataType – A built-in or imported Java class

Pattern Match Rule – a rule with one or more Selectors which generates a set of object bindings

If-Then Rule – an inferencing rule used by the Forward and Backward chaining engines.

If-Then-Else Rule – a scripting rule used by the Script engine

Predicate – a functor with zero or more arguments

Predicate Fact - an if-then rule with a single consequent clause and zero antecedent clauses.

Predicate Rule – an if-then rule with a single consequent clause and one or more antecedent clauses.

Inference Engine – a control algorithm for processing a set of rules

Iteration Rule – a rule that loops over some test condition.

# Appendix

## Appendix A. ABLE Rule Language Grammar

Version 2.0.0

This appendix describes the ABLE rule language text grammar using Backus Naur Format. The grammar uses the following BNF-style conventions:

[x] denotes zero or one occurrences of x.
{x} denotes zero or more occurrences of x.
x | y means one of either x or y.

```
ruleset_declaration   ::=  ( [ doc_comment ]  [ “template” ] "ruleset" identifier
                           "{" { import_statement }
                               { library_statement }
                               [ predicates_declaration ]
                               { class_definition }
                               { variable_declaration_block }
                               [ inputs_declaration ]  [ outputs_declaration ] [ functions_declaration ]
                               { ruleblock_declaration }
                            "}"
```

```
doc_comment    ::= "/**" "... text ..." "*/"
```

```
modifier   ::=  “template”  | "static“
```

```
identifier   ::= "a..z,$,_" { "a..z,$,_,0..9" }
```

```
identifier_list ::- “{“ identifier { "," identifier }  “}”
```

```
class_name  ::=  identifier
```

```
import_statement   ::=  "import" ( package_name  “.” class_name  ) ";"
```

```
library_statement   ::=  "library" ( package_name  “.” class_name  ) ";"
```

```
predicates_declaration  ::=  "predicates"  identifier_list “;”
```

```
inputs_declaration  ::=  "inputs"  identifier_list “;”
```

```
outputs_declaration  ::=  "outputs" identifier_list  ";"
```

```
functions_declaration  ::=  "functions" function_identifier_list “;”
```

```
function_identifier_list ::- “{“ identifier “/”{ "0..9" }  { "," identifier “/”{ "0..9" }  }  “}”
```

```
type_declaration    ::=    [ doc_comment ]  class_declaration  ";"

class_declaration   ::=    "class" class_name   "{" { variable_declaration } "}"

type      ::=     type_specifier  [ "[" "]" ]

type_specifier   ::=
       "Boolean"        | "Byte"  | "Character"  | "Short"   | "Integer"  | "Float"  | "Long"   | "Double"
     | “Categorical”  |  “Continuous”  |  “Discrete”  | “Fuzzy” |  “TimeStamp”   | “TimePeriod”
     | “Selector”        | “Expression” | class_name



variable_declaration_block   ::=  “variables”  {  variable_declaration  }

variable_declaration  ::=  { modifier } type variable_declarator   ";"

variable_declarator  ::=   identifier [ “[“ “]” ]   [ "=" variable_initializer ]

variable_initializer   ::=   expression    | ( "{" [ variable_initializer  { "," variable_initializer } [ "," ] ] "}" )


arg_list   ::=  expression { "," expression }

expression     ::=    numeric_expression  | testing_expression   | logical_expression   | string_expression
     | bit_expression    | creating_expression   | literal_expression    | "null"     | "this"
     | identifier   |  function_call   |  method_call |  field_reference  | array_index_expression |
     |   "(" expression ")"

function_call  ::= identifier   "(" [ arg_list ] ")"

method_call ::= identifier  “.” “(“ [ arg_list] “)”

array_index_expression  ::= identifier “[“ expression “]”

field_reference ::= identifier “.” identifier

 string    ::= """ { character } """

string_expression    ::= ( expression   "+"  expression )

bit_expression   ::=   ( "~" expression )   |  ( expression   ( ">>="    | "<<"   | ">>"   | ">>>" )   expression )

creating_expression    ::=  "new" (   ( classe_name "(" [ arglist ] ")" )  |
                                        |  ( type_specifier [ "[" expression "]" ] { "[" "]" } )
                                        | ( "(" expression ")" )     )

testing_expression     ::=    ( expression   ( ">"   | "<"   | ">="  | "<="  | "=="  | "!=" )  expression )

numeric_expression   ::=  ( ( "-"  )  expression )  |  ( expression   ( "+" | "-"  | "*"  | "/" | "%") expression )

decimal_digits   ::= "0..9" { "0..9" }

exponent_part    ::=  "e" [ "+" | "-" ] decimal_digits

float_literal
```

```
    ::=
    ( decimal_digits "." [ decimal_digits ] [ exponent_part ] [ float_type_suffix ] )
    | ( "." decimal_digits [ exponent_part ] [ float_type_suffix ] )
    | ( decimal_digits [ exponent_part ] [ float_type_suffix ] )

float_type_suffix   ::=   "f" | "d"

integer_literal  ::=  ( ( "1..9" { "0..9" } )  | { "0..7" }   | ( "0" "x" "0..9a..f" { "0..9a..f" } ) )  [ "L" ]

literal_expression    ::=   integer_literal   | float_literal   | string   | character

logical_expression    ::=    ( "!" expression )
                | ( expression   ( "ampersand"   | "|"   | "^"   | ( "ampersand" "ampersand" )  | "%")  expression )
                | "true"     | "false"


ruleblock_declaration   ::=  [ doc_comment ] type identifier “(“ “)”  “using” engine_type  rule_block

engine_type ::= ( identifier |  package_name “.” class_name)

rule_block    ::= "{" { rule_statement } "}"

rule_statement ::= [“template”]   rule_header   rule

rule
    ::=
    | ( assertion_rule)
    | ( if_then_else_rule )
    | ( do_while_rule )
    | ( do_until_rule )
    | ( while_rule )
    | ( for_loop_rule )
    | (predicate_fact)
    | (predicate_rule)
    | (pattern_match_rule)


action  ::= expression “;”

action_block ::=   “{“ ( action { action } ) “}”

actions ::= action | action_block

rule_header ::= [ identifier ]  [ preconditions ]  [ priority]  “:”

preconditions ::=  “{“ {argList}  “}”

priority ::= “[“ expression “]”

assertion_rule ::= expression “;”

for_loop_rule
    ::=    "for" "(" ( variable_declaration | ( expression ";" ) | ";" )   [ expression ] ";"   [ expression ] ";"   ")"
action_block

if_then_else_rule    ::=     "if" "(" expression ")" [ “then” ] actions   [ "else" actions ]
```

do_while_rule ::= "do" action_block "while" "(" expression ")" ";"

do_until_rule ::= "do" action_block "until" "(" expression ")" ";"

while_rule ::= "while" "(" expression ")" action_block

pattern_match_rule ::= “when” “(“ { selector_list } “)” “do” action_block

selector ::= type identifier “(“ expression “)”

selector_list ::= selector { “&” selector }

predicate ::= functor “(“ { arg_list } “)”

predicate_list ::= predicate { “,” predicate }

predicate_fact ::= predicate “.”

predicate_rule ::= predicate “:-“ predicate_list “.”

# Appendix B. ABLE Rule Language XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="AbleRuleSet">
        <xsd:complexType>
            <xsd:sequence minOccurs="1" maxOccurs="1">
                <xsd:element ref="import"     minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="library"    minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="predicates" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="variables"  minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="inputs"     minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="outputs"    minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="functions"  minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="ruleBlock"  minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name"     type ="xsd:ID"      use="required"/>
            <xsd:attribute name="comment"  type ="xsd:string"  use="optional"/>
            <xsd:attribute name="template" type ="xsd:boolean" use="optional"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="inference">
        <xsd:complexType>
            <xsd:attribute name="method" default="script">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="backward"/>
                        <xsd:enumeration value="forward"/>
                        <xsd:enumeration value="patternMatch"/>
                        <xsd:enumeration value="patternMatchRete"/>
                        <xsd:enumeration value="predicate"/>
                        <xsd:enumeration value="script"/>
                        <xsd:enumeration value="fuzzy"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
        </xsd:complexType>
    </xsd:element>

     <xsd:element name="import">
        <xsd:complexType>
            <xsd:attribute name="class" type ="xsd:ID" use="required"/>
        </xsd:complexType>
    </xsd:element>


    <xsd:element name="library">
       <xsd:complexType>
           <xsd:attribute name="class" type ="xsd:ID" use="required"/>

       </xsd:complexType>
    </xsd:element>

    <xsd:element name="predicates">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="predicate" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
```

```
    <xsd:element name="predicate">
        <xsd:complexType>
            <xsd:attribute name="name" type ="xsd:ID" use="required"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="variables">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="declareVar" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="declareClass">
        <xsd:complexType>
            <xsd:attribute name="comment"     type ="xsd:string" use="optional"/>
            <xsd:attribute name="className"   type ="xsd:ID"     use="required"/>
            <xsd:attribute name="fieldInfo"   type ="xsd:IDREFS" use="required"/>
        </xsd:complexType>
    </xsd:element>


    <xsd:element name="declareVar">
        <xsd:complexType>
            <xsd:sequence>
              <xsd:element ref="varInitializer"   minOccurs="1" maxOccurs="1"/>
              <xsd:choice minOccurs="1" maxOccurs="1">
                <xsd:element ref="categoricalItem" minOccurs="1"
maxOccurs="unbounded"/>
                <xsd:element ref="discreteItem"    minOccurs="1"
maxOccurs="unbounded"/>
                <xsd:element ref="setDefinitions"  minOccurs="1" maxOccurs="1"/>
              </xsd:choice>
            </xsd:sequence>
            <xsd:attribute name="comment"      type ="xsd:string" use="optional"/>
            <xsd:attribute name="varName"      type ="xsd:ID"      use="required"/>

            <xsd:attribute name="dataType"     type ="xsd:NMTOKEN" use="required"/>
            <xsd:attribute name="static"       type ="xsd:boolean" use="optional"/>
            <xsd:attribute name="template"     type ="xsd:boolean" use="optional"/>
            <xsd:attribute name="length"       type ="xsd:integer" use="optional"/>
            <xsd:attribute name="from"         type ="xsd:NMTOKEN" use="optional"/>
            <xsd:attribute name="to"           type ="xsd:NMTOKEN" use="optional"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="varInitializer">
         <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="argList" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
          </xsd:complexType>
    </xsd:element>

    <xsd:element name="categoricalItem">
        <xsd:complexType>
            <xsd:attribute name="value" type ="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
```

```
<xsd:element name="discreteItem">
    <xsd:complexType>
        <xsd:attribute name="value" type ="xsd:NMTOKEN" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="setDefinitions">
    <xsd:complexType>
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
            <xsd:element ref="betaSet"      minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="gaussianSet"  minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="linearSet"    minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="piSet"        minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="segmentsSet"  minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="shoulderSet"  minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="sigmoidSet"   minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="trapezoidSet" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="triangleSet"  minOccurs="1" maxOccurs="1"/>
        </xsd:choice>
    </xsd:complexType>
</xsd:element>

<xsd:element name="betaSet">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="complementSet" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="setName"     type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="centerPoint" type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="width"       type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="weight"      type ="xsd:NMTOKEN" default="1.0"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="gaussianSet">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="complementSet" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="setName"     type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="centerPoint" type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="widthFactor" type ="xsd:NMTOKEN" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="linearSet">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="complementSet" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="setName"    type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="beginPoint" type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="endPoint"   type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="direction" use="required">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="up"/>
                    <xsd:enumeration value="down"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>
```

```
<xsd:element name="piSet">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="complementSet" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="setName"     type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="centerPoint" type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="width"       type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="weight"      type ="xsd:NMTOKEN" default="1.0"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="segmentsSet">
    <xsd:complexType>
        <xsd:sequence minOccurs="1" maxOccurs="1">
            <xsd:element ref="membership"    minOccurs="1" maxOccurs="unbounded"/>
            <xsd:element ref="complementSet" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="setName" type ="xsd:NMTOKEN" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="membership">
    <xsd:complexType>
        <xsd:attribute name="point"      type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="truthValue" type ="xsd:NMTOKEN" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="shoulderSet">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="complementSet" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="setName"    type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="beginPoint" type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="endPoint"   type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="direction" use="required">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="left"/>
                    <xsd:enumeration value="right"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>

<xsd:element name="sigmoidSet">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="complementSet" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="setName"    type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="leftPoint"  type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="flexPoint"  type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="rightPoint" type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="direction"  use="required">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="up"/>
                    <xsd:enumeration value="down"/>
```

```
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>

<xsd:element name="trapezoidSet">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="complementSet" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="setName"        type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="leftPoint"      type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="leftCorePoint"  type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="rightCorePoint" type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="rightPoint"     type ="xsd:NMTOKEN" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="triangleSet">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="complementSet" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="setName"     type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="leftPoint"   type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="centerPoint" type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="rightPoint"  type ="xsd:NMTOKEN" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="complementSet">
    <xsd:complexType>
        <xsd:attribute name="setName" type ="xsd:NMTOKEN" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="inputs">
    <xsd:complexType>
        <xsd:attribute name="varRefs" type ="xsd:IDREFS" use="optional"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="outputs">
    <xsd:complexType>
        <xsd:attribute name="varRefs" type ="xsd:IDREFS" use="optional"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="functions">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="function" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="function">
    <xsd:complexType>
        <xsd:attribute name="name" type ="xsd:ID" use="required"/>
    </xsd:complexType>
</xsd:element>
```

```
<xsd:element name="ruleBlock">
    <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="inference" minOccurs="1" maxOccurs="1"/>
          <xsd:element ref="rule"      minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="block"   type ="xsd:ID"     use="required"/>
        <xsd:attribute name="type"    type ="xsd:string" use="required"/>
        <xsd:attribute name="comment" type ="xsd:string" use="optional"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="rule">
    <xsd:complexType>
        <xsd:sequence>
           <xsd:element ref="preConditions" minOccurs="0" maxOccurs="1"/>
           <xsd:element ref="priority"  minOccurs="0" maxOccurs="1"/>
           <xsd:choice  minOccurs="1"   maxOccurs="1">
            <xsd:element ref="assert"   minOccurs="1" maxOccurs="1"/>
            <xsd:sequence minOccurs="1" maxOccurs="1">
                <xsd:element ref="if"   minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="then" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="else" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:element ref="when"          minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="predicateRule" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="whileDo"       minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="doWhile"       minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="doUntil"       minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="for"           minOccurs="1" maxOccurs="1"/>
          </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="label"    type ="xsd:ID"      use="required"/>
        <xsd:attribute name="comment"  type ="xsd:string"  use="optional"/>
        <xsd:attribute name="template" type ="xsd:boolean" use="optional"/>
    </xsd:complexType>
</xsd:element>


<xsd:element name="preConditions">
   <xsd:complexType>
     <xsd:sequence>
        <xsd:element ref="arrayLiteral" minOccurs="1" maxOccurs="1"/>
     </xsd:sequence>
   </xsd:complexType>
</xsd:element>


  <xsd:element name="priority">
   <xsd:complexType>
     <xsd:sequence>
        <xsd:element ref="arg" minOccurs="1" maxOccurs="1"/>
     </xsd:sequence>
   </xsd:complexType>
</xsd:element>

<xsd:element name="assert">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="expression" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

```
<xsd:element name="if">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="antecedent" minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="then">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="consequent" minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="else">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="consequent" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>

        <xsd:attribute name="conditionalRule" type ="xsd:boolean" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="when">
    <xsd:complexType>
        <xsd:sequence minOccurs="1" maxOccurs="1">
            <xsd:element ref="selector" minOccurs="1" maxOccurs="unbounded"/>
            <xsd:element ref="do"       minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="selector">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="constraint" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="varName" type ="xsd:ID"      use="required"/>
        <xsd:attribute name="varType" type ="xsd:NMTOKEN" use="required"/>
        <xsd:attribute name="pos" use="optional">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="true"/>
                    <xsd:enumeration value="false"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>


<xsd:element name="whileDo">
    <xsd:complexType>
        <xsd:sequence minOccurs="1" maxOccurs="1">
            <xsd:element ref="antecedentExpr" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="do_"            minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
```

```
</xsd:element>


<xsd:element name="doWhile">
    <xsd:complexType>
        <xsd:sequence minOccurs="1" maxOccurs="1">
            <xsd:element ref="antecedentExpr" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="do_"            minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="doUntil">
    <xsd:complexType>
        <xsd:sequence minOccurs="1" maxOccurs="1">
            <xsd:element ref="antecedentExpr" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="do_"            minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>


<xsd:element name="for">
    <xsd:complexType>
        <xsd:sequence minOccurs="1" maxOccurs="1">
            <xsd:element ref="forInit" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="forTest" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="forIter" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="do_"     minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>



<xsd:element name="forInit">
    <xsd:complexType>
        <xsd:sequence minOccurs="1" maxOccurs="1">
            <xsd:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
    </xsd:complexType>


</xsd:element> <xsd:element name="forTest">
    <xsd:complexType>
        <xsd:sequence minOccurs="1" maxOccurs="1">
            <xsd:element ref="expression" minOccurs="0" maxOccurs="1"/>
         </xsd:sequence>
    </xsd:complexType>


</xsd:element> <xsd:element name="forIter">
    <xsd:complexType>
        <xsd:sequence minOccurs="1" maxOccurs="1">
            <xsd:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="do">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="action" minOccurs="1" maxOccurs="unbounded"/>
```

```
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="do_">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="action_" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="predicateRule">
        <xsd:complexType>
            <xsd:choice minOccurs="1" maxOccurs="1">
                <xsd:element ref="predicateRuleFact" minOccurs="1" maxOccurs="1"/>
                <xsd:sequence minOccurs="1" maxOccurs="1">
                    <xsd:element ref="predicateRuleHead" minOccurs="1" maxOccurs="1"/>
                    <xsd:element ref="predicateRuleBody" minOccurs="1" maxOccurs="1"/>
                </xsd:sequence>
            </xsd:choice>
         </xsd:complexType>
    </xsd:element>

    <xsd:element name="predicateRuleFact">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="predicateDef" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="predicateRuleHead">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="predicateDef" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="predicateRuleBody">
        <xsd:complexType>
            <xsd:choice minOccurs="1" maxOccurs="unbounded">
                <xsd:element ref="predicateDef" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="constraint" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="action" minOccurs="1" maxOccurs="1"/>
            </xsd:choice>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="predicateDef">
        <xsd:complexType>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="predicateDef" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateSymbol" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateVariableArg" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateBooleanArg" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateNumericArg" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateListArg" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateStringArg" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateDontCareSymbol" minOccurs="1"
maxOccurs="1"/>
            </xsd:choice>
```

```
            <xsd:attribute name="name" type ="xsd:ID" use="required"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="predicateBooleanArg" type="xsd:string"/>
    <xsd:element name="predicateListArg">
        <xsd:complexType>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="predicateDef" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateSymbol" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateVariableArg" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateBooleanArg" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateNumericArg" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateListArg" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateStringArg" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="predicateDontCareSymbol" minOccurs="1"
maxOccurs="1"/>
                <xsd:element ref="predicateListArgTail" minOccurs="1" maxOccurs="1"/>
            </xsd:choice>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="predicateListArgTail" type="xsd:string"/>
    <xsd:element name="predicateNumericArg" type="xsd:string"/>
    <xsd:element name="predicateStringArg" type="xsd:string"/>
    <xsd:element name="predicateSymbol">
        <xsd:complexType>
            <xsd:attribute name="value" type ="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="predicateVariableArg">
        <xsd:complexType>
            <xsd:attribute name="varRef" type ="xsd:IDREF" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="predicateDontCareSymbol" type="xsd:string"/>


    <xsd:element name="antecedent">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="expression" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>


     <xsd:element name="antecedentExpr">
        <xsd:complexType>
          <xsd:sequence>
                <xsd:element ref="expression" minOccurs="1" maxOccurs="1"/>
          </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="constraint">
     <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="expression" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
```

```
<xsd:element name="consequent">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="expression" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="action">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="expression" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="action_">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="expression" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>


<xsd:element name="booleanLiteral">
    <xsd:complexType>
        <xsd:attribute name="value" use="required">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="true"/>
                    <xsd:enumeration value="false"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>

<xsd:element name="byteLiteral">
    <xsd:complexType>
        <xsd:attribute name="value" type ="xsd:byte" use="required"/>
    </xsd:complexType>
</xsd:element>


<xsd:element name="characterLiteral">
    <xsd:complexType>
        <xsd:attribute name="value" type ="xsd:string" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="doubleLiteral">
    <xsd:complexType>
        <xsd:attribute name="value" type ="xsd:double" use="required"/>
    </xsd:complexType>
</xsd:element>



<xsd:element name="floatLiteral">
    <xsd:complexType>
```

```
        <xsd:attribute name="value" type ="xsd:float" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="integerLiteral">
    <xsd:complexType>
        <xsd:attribute name="value" type ="xsd:integer" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="longLiteral">
    <xsd:complexType>
        <xsd:attribute name="value" type ="xsd:long" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="shortLiteral">
    <xsd:complexType>
        <xsd:attribute name="value" type ="xsd:short" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="stringLiteral">
    <xsd:complexType>
      <xsd:attribute name="value" type ="xsd:string" use="required"/>
    </xsd:complexType>
 </xsd:element>


 <xsd:element name="arrayLiteral">
    <xsd:complexType>
        <xsd:attribute name="value" type ="xsd:NMTOKEN" use="required"/>
    </xsd:complexType>
</xsd:element>


<xsd:element name="fieldValue">
    <xsd:complexType>
        <xsd:attribute name="varRef"    type ="xsd:IDREF" use="required"/>
        <xsd:attribute name="fieldName" type ="xsd:ID"    use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="functionValue">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="argList" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="functionRef" type ="xsd:IDREF" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="methodValue">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="argList" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="varRef"     type ="xsd:IDREF" use="required"/>
        <xsd:attribute name="methodName" type ="xsd:ID"    use="required"/>
    </xsd:complexType>
</xsd:element>
```

```
 <xsd:element name="newObjectLiteral">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="argList"     minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element ref="expression"  minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="dataType"     type ="xsd:IDREF" use="required"/>
        <xsd:attribute name="className"    type ="xsd:ID"    use="required"/>
    </xsd:complexType>
</xsd:element>


 <xsd:element name="arrayExprValue">
    <xsd:complexType>
       <xsd:attribute name="varRef" type ="xsd:IDREF"  use="required"/>
       <xsd:attribute name="index"  type ="xsd:string" use="required"/>
    </xsd:complexType>
 </xsd:element>


<xsd:element name="argList">
    <xsd:complexType>
     <xsd:sequence>
       <xsd:element ref="arg" minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
    </xsd:complexType>
</xsd:element>


  <xsd:element name="arg">
    <xsd:complexType>
        <xsd:choice minOccurs="1" maxOccurs="1">
            <xsd:element ref="booleanLiteral" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="doubleLiteral"  minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="floatLiteral"   minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="integerLiteral" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="longLiteral"    minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="stringLiteral"  minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="variableValue"  minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="fieldValue"     minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="functionValue"  minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="methodValue"    minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="arrayExprValue" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="newObjectLiteral" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="expression"     minOccurs="1" maxOccurs="1"/>
        </xsd:choice>
    </xsd:complexType>
</xsd:element>

<xsd:element name="token">
    <xsd:complexType>
        <xsd:attribute name="value" type ="xsd:NMTOKEN" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="hedge">
    <xsd:complexType>
        <xsd:attribute name="type" use="required">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="about"/>
                    <xsd:enumeration value="above"/>
                    <xsd:enumeration value="below"/>
```

```
                    <xsd:enumeration value="closeTo"/>
                    <xsd:enumeration value="extremely"/>
                    <xsd:enumeration value="generally"/>
                    <xsd:enumeration value="inVicinityOf"/>
                    <xsd:enumeration value="not"/>
                    <xsd:enumeration value="positively"/>
                    <xsd:enumeration value="slightly"/>
                    <xsd:enumeration value="somewhat"/>
                    <xsd:enumeration value="very"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>

<xsd:element name="fuzzyValue">
    <xsd:complexType>
        <xsd:attribute name="setRef" type ="xsd:NMTOKEN" use="required"/>
    </xsd:complexType>
</xsd:element>


<xsd:element name="fuzzyVar">
    <xsd:complexType>
        <xsd:attribute name="varRef" type ="xsd:IDREF" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="variableValue">
    <xsd:complexType>
        <xsd:attribute name="varRef" type ="xsd:IDREF" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="variableReference">
    <xsd:complexType>
        <xsd:attribute name="varRef" type ="xsd:IDREF" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="weight">
    <xsd:complexType>
        <xsd:attribute name="value" type ="xsd:NMTOKEN" default="1.0"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="expression">
 <xsd:complexType>
  <xsd:sequence>
    <xsd:element name="lhs"  type="expressionArg"  minOccurs="1" maxOccurs="1" />
    <xsd:element name="oper" type="expressionOper" minOccurs="1" maxOccurs="1"/>
    <xsd:element name="rhs"  type="expressionArg"  minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="weight" type="xsd:double" use="required"/>
 </xsd:complexType>
</xsd:element>


<xsd:element name="expressionArgList">
    <xsd:complexType>
     <xsd:sequence>
       <xsd:element ref="expression" minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
```

```
      <xsd:attribute name="length" type="xsd:int" use="required"/>
      </xsd:complexType>
    </xsd:element>

     <xsd:complexType name="expressionArg">
       <xsd:choice>
        <xsd:element ref="variableReference" minOccurs="1" maxOccurs="1" />
        <xsd:element ref="expression" minOccurs="1" maxOccurs="1" />
        <xsd:element ref="booleanLiteral" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="doubleLiteral" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="integerLiteral" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="floatLiteral" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="longLiteral" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="stringLiteral" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="variableValue" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="fieldValue" minOccurs="1" maxOccurs="1"/>
        <xsd:element ref="methodValue" minOccurs="1" maxOccurs="1"/>
       </xsd:choice>
     </xsd:complexType>


      <xsd:complexType name="expressionOper">
       <xsd:choice>
        <xsd:element name="equals"       type="xsd:string"/>
        <xsd:element name="is"           type="xsd:string"/>
        <xsd:element name="isEqualTo"    type="xsd:string"/>
        <xsd:element name="isNotEqualTo" type="xsd:string"/>
        <xsd:element name="isLessThan"   type="xsd:string"/>
        <xsd:element name="isLessThanOrEqualTo"    type="xsd:string"/>
        <xsd:element name="isGreaterThan"          type="xsd:string"/>
        <xsd:element name="isGreaterThanOrEqualTo" type="xsd:string"/>
        <xsd:element name="logicalNot"  type="xsd:string"/>
        <xsd:element name="logicalAnd"  type="xsd:string"/>
        <xsd:element name="logicalOr"   type="xsd:string"/>
        <xsd:element name="plus"        type="xsd:string"/>
        <xsd:element name="minus"       type="xsd:string"/>
        <xsd:element name="multiply"    type="xsd:string"/>
        <xsd:element name="divide"      type="xsd:string"/>
        <xsd:element name="modulo"      type="xsd:string"/>
        <xsd:element name="unaryMinus"  type="xsd:string"/>
        <xsd:element name="unaryPlus"   type="xsd:string"/>
        <xsd:element name="noOperation" type="xsd:string"/>
        <xsd:element name="bitwiseAnd"    type="xsd:string"/>
        <xsd:element name="bitwiseOr"     type="xsd:string"/>
        <xsd:element name="bitwiseXor"    type="xsd:string"/>
        <xsd:element name="bitwiseNot"    type="xsd:string"/>
        <xsd:element name="bitShiftLeft"  type="xsd:string"/>
        <xsd:element name="bitShiftRight" type="xsd:string"/>
        <xsd:element name="bitShiftRightZeroFill"   type="xsd:string"/>
       </xsd:choice>
     </xsd:complexType>


</xsd:schema>
```