Tutorial: XML and scripting languages
Manipulating XML documents with Perl and other scripting languages

Parand Tony Daruger
Co-founder, Binary Evolution, Inc.
February 2000

In this first tutorial of his series on using scripting languages to manipulate and transform XML documents, Binary Evolution's Parand Tony Daruger takes you through the first steps of using these techniques with Perl. You'll see a method for transforming XML to HTML, followed by a simple stock trading application that uses Perl, XML, and a database to evaluate trading rules. You can apply the techniques using other scripting languages too, including Tcl and Python.

Contents:

XML and scripting languages have had a natural relationship since the early days of XML's inception. One of the original goals of the XML design group was to enable a Perl hacker to write an XML parser in two weeks. Handling and manipulating XML has also been fertile ground for using scripting with XML: XML is designed to be human legible, and thus mainly text based. Scripting languages have historically been extremely adept at manipulating text. Scripting's flexibility and power make it a perfect complement to XML's descriptive abilities.

XML is a medium for the expression, storage, and exchange of information. The addition of scripting allows this information to become active: to affect actions, undergo transformations, connect with existing systems, and take action instead of simply expressing information.

In this article we will use Perl to manipulate and transform XML. You can apply the same techniques using other scripting languages too, including Tcl and Python. First a technique for transformation of XML to HTML will be shown, followed by a simple stock trading application that uses Perl, XML, and a database to evaluate trading rules.

Converting XML to HTML

For the purposes of this article, we will use a stock quote, expressed as XML, as our input file:

```
<stock_quote>
 <symbol>IBM</symbol>
 <when>
   <date>12/16/1999</date>
   <time>4:40PM</time>
 </when>
 <price type="ask"     value="109.1875"/>
 <price type="open"    value="108"/>
 <price type="dayhigh" value="109.6875"/>
 <price type="daylow"  value="105.75"/>
 <change>+2.1875</change>
 <volume>7050200</volume>
</stock_quote>
```

This simple encoding captures information typically found in a stock quote. The formatting demonstrates certain XML features, such as attributes and empty tags. The actual XML file used in this article contains several `stock_quote` elements, to form a portfolio of stocks.

This XML file was created using a script to convert the Spreadsheet Format stock quotes provided by the finance.yahoo.com Web site into XML. The scripts used to perform the conversion, as well as a live XML stock quote server, are available from the XML Today Web site (see Resources).

Simple substitution

A simple method for transforming the XML source into HTML is to define pieces of HTML to be substituted for each XML tag. Using the popular XML::Parser module for Perl (see Resources), based on James Clark's Expat parser, we can parse the XML document and define callback routines for performing the substitutions.

Here is a simple invocation of the XML::Parser:

```
use XML::Parser;

my $parser = new XML::Parser(ErrorContext => 2);
$parser->setHandlers(Start => \&start_handler,
                     End   => \&end_handler,
                     Char  => \&char_handler);

$parser->parsefile($file);
```

This parses the given file, invoking the function `start_handler` each time a tag is started, and `end_handler` each time a tag is ended. The contents of the tag are processed by the `char_handler` function.

Given these callback functions, we can implement our simple substitution algorithm. First, we define a few substitutions:

```
 %startsub = (
"stock_quote"        =>      "<hr><p>",
"symbol"             =>      "<h2>",
"price"              =>      "<br><b>Price:</b>"
);

%endsub = (
"stock_quote"        =>      "",
"symbol"             =>      "</h2>",
"price"              =>      ""
);
```

And now we write the handlers to perform the substitutions:

```
sub start_handler
{
    my $expat = shift; my $element = shift;

    # element is the name of the tag
    print $startsub{$element};
}

sub char_handler
{
    my ($p, $data) = @_;
    print $data;

}
```

The `start_handler` function simply prints the value to be substituted for the given tag. `char_handler` outputs the data it receives, which is the content of the tags. (The full program, with a few additions to handle attributes, is listed separately.) Running the program on our XML file, we get the following output:

```
<hr><p>
 <h2>IBM</h2>

   <br><b>Date:</b><i>12/16/1999</i>
   <br><b>Time:</b><i>4:40PM</i>

 <br><b>Price:</b>type=ask value=109.1875
 <br><b>Price:</b>type=open value=108
 <br><b>Price:</b>type=dayhigh value=109.6875
 <br><b>Price:</b>type=daylow value=105.75
 <br><b>Change:</b>+2.1875
 <br><b>Volume:</b>7050200
```

The full output is available. Using this methodology, we can make simple XML to HTML transformations by defining substitutions.

Function-based substitution
Substitution-based transformations are easy to implement and understand, but don't give us the ability to implement logic. We may want to take different actions based on the contents or attributes of a tag, or connect to a database to compare the contents of the tag with the stored value. We need more than simple, one-to-one substitutions; we need the ability to perform functions for each tag.

XML::Parser provides a method for invoking functions for each tag in the XML document. For each tag, the parsing module calls a function with the tag's name. Thus we can define a set of functions that perform the transformations, connect to databases, and implement our business logic.

To enable the function callbacks based on tag names, we need to invoke the parser with the "Subs" style. We also need to specify which namespace the function callbacks reside in, via the "Pkg" option:

```
my $parser = new XML::Parser(Style=>'Subs',
                             Pkg=>'SubHandlers',
                            ErrorContext => 2);

$parser->setHandlers(Char  => \&char_handler);
```

This will cause a series of function callbacks based on the tags and the contents of the XML file. The start of a tag will invoke a function with the same name as the tag in the `SubHandlers` namespace. The contents of the tag will be handled by the `char_handler` function, and the end of the tag will invoke a function with the same name as the tag, with an "`_`" appended (for example, for the end of the tag `symbol`, the function `SubHandlers::symbol_()` will be called).

Our XML file will cause the following sequence of function calls:

```
SubHandlers::stock_quotes();
SubHandlers::stock_quote();
SubHandlers::symbol();
char_handler();
SubHandlers::symbol_();
SubHandlers::when();
....
```

Now, it is a simple matter to write the transformation functions. We can still perform simple substitutions:

```
sub symbol {
  print "<img src=images/";
}

sub symbol_ {
  print ".gif>\n";
}
```

which use the stock symbol, contained in the contents of the `symbol` tag, to insert an image of the same name in the resulting HTML page. We can also implement more complicated logic:

```
sub price {
  my $expat = shift; my $element = shift;

  # Read the attributes
  while (@_) {
    my $att = shift;
    my $val = shift;
    $attr{$att} = $val;
  }

  my $type  = $attr{'type'};
  my $price = $attr{'value'};

  if ($type eq 'ask') {
    $label="Ask Price";
  } elsif ($type eq 'open') {
    $label="Opening Price";
  }

  print "<td align=left>\n<b>$label</b></td>\n";
  print "<td align=right>$price</td>\n";

}
```

The first parameter passed to the function is a handle to the parser itself, followed by the name of the tag (`element`), optionally followed by the tag attributes as `attribute_name, attribute_value` pairs. In the above case we print a different label based on the `type` attribute of the `price` tag.

The full program is available as a separate listing. Running the program on our XML file produces much more attractive output. A sample looks like:

```
<table width=100%>
 <tr>
  <td>
   <img src=images/IBM.gif><br><br>  12/16/1999 4:40PM
  </td>
  <td>
   <table width=100%>
    <tr>
     <td align=left><b>Ask Price</b></td>
     <td align=right>109.1875</td>
     <td align=left><b>Opening Price</b></td>
     <td align=right>108</td>
```

```
      </tr>
      <tr>
       <td align=left><b>Today's High</b></td>
       <td align=right>109.6875</td>
      </tr>
     </table>
    </td>
   </tr>
  </table>
```

Tree-based processing
The methodologies we have discussed so far are based on processing the XML document as a stream -- in the course of parsing the file, handlers are called as each tag is encountered. This provides an efficient means of processing XML, both in terms of memory usage and processing time. Certain tasks, however, are somewhat difficult to do. Imagine, for example, needing to move or rearrange certain segments of the document, or sorting items within the document. Because we receive the document as a stream, we would need to store the components before sorting or rearranging them. A mechanism that would store the components automatically would make such tasks substantially easier.

XML documents are required to be *well balanced*, making it easy to store them as trees. A popular technique for working with XML documents is to first parse them into a tree data structure, and then to operate on the tree. The Document Object Model (DOM), as well as Grove and Twig (see Resources), use this model. This enables a great deal of flexibility in dealing with the documents: the components of the document can be accessed in random order, rearranged, added, or removed.

Tree-based methodologies do have some drawbacks, however. They require the parsing of the entire XML document, as well as the creation of the tree data structure, before the processing and business logic take place. Since the tree data structure is generally stored in memory, these methods have much larger memory footprints than stream based methods. The problem is exacerbated by the fact that storing the document in memory as a tree takes several times as much storage as the original XML document did. For larger documents both of these can be significant -- the parsing and tree creation time become substantial, and the memory requirements can overrun the available resources.

Tree-based processing of XML documents will be discussed in a future article. The remainder of this article will use stream-based processing, as described above.

Active XML documents
Converting XML documents for display is a typical first task in working with XML, and serves as a good introduction to the machinery involved. The real power of XML, however, lies in its ability not only to transmit information, but also to trigger actions based on the transmitted information. We will examine a sample application that uses these active documents to implement simple stock trading rules.

The basic scenario is as follows: a stock quote service will periodically send an XML document with the latest prices and volume for our chosen stocks (in the format of the XML file we have been using thus far). Our application will decide whether to buy or sell based on the stock quotes and a set of rules stored in our database.

For this simple application we will only buy or sell, using the asking price and volume as the criteria. The price and volume will be received from the XML file, and the rules will be retrieved from a MySQL database (see Resources). The rules will be evaluated, and if buying or selling is required, the corresponding command will be issued.

Storing tag contents
In the earlier display-oriented applications, we only needed to output the tag contents. For our stock application, we need to access and store the contents of certain tags (such as price and volume) to compare them with the buy/sell criteria.

Our strategy for storing the contents of the tags using the stream-based processing model will be: prepare a storage place for the contents when the start of the tag is encountered, and store the contents of the tag via the `char_handler` function. Since the document is processed as a stream, first the start tag will be encountered, allowing us to set the stage for the storage of the contents. Next the contents of the tag will be encountered, and stored in their prepared location. Finally the end of the tag will be encountered, allowing any necessary cleanup and closeup of the storage location.

The storage location will be set up in the tag start function, and closed in the tag end function:

```
sub volume {
  $::state::store_contents = "volume";
}

sub volume_ {
  undef $::state::store_contents;
}
```

`volume` defines the `store_contents` variable, setting the storage location for the contents of the tag. `volume_` subsequently undefines `store_contents`, making sure contents of other tags do not get stored in the same location.

`char_handler` needs to be modified to allow storage of the contents:

```
sub char_handler {
    my ($p, $data) = @_;
    if ($::state::store_contents) {
      $::state::storage{$::state::store_contents} .= $data;
    }
}
```

This checks if the variable `store_contents` is defined, and, if so, stores the data in the `storage` hash. The `::state` namespace is used to separate the storage and state variables from the parser and handler namespaces.

Using this technique we can store the contents of the tags we are interested in. In the case of the price tag, the values of interest are expressed as attributes of the tag. We can store these as we encounter them:

```
sub price {
  my $expat = shift; my $element = shift;

  # Read the attributes
  while (@_) {
    my $att = shift;
    my $val = shift;
    $attr{$att} = $val;
  }

  if ($attr{'type'} eq "ask") {
    $::state::storage{'price'} = $attr{'value'};
  }
}
```

Retrieving the rules
Our buy/sell rules are stored in the following table:

```
CREATE TABLE rules (
       symbol  CHAR(5),
       field   CHAR(8),
       value   CHAR(16),
       action  CHAR(5)
);
```

`symbol` is the stock symbol. `field` describes which field will be used in the criterion (in this case either price or volume). `value` is the value of `field` which would trigger an action. `action` describes the type of action to take (in this case either buy or sell).

Thus the following row from the rules table:

```
INSERT INTO rules VALUES ("IBM", "price", "120.0", "buy");
```

means *if the price of the IBM stock is greater than 120, issue a buy order*. And

```
INSERT INTO rules VALUES ("MSFT", "volume", "65000000", "sell");
```

means *if the trading volume of Microsoft stock is over 65000000, issue a sell order*.

Retrieving these rules from the database is a simple matter using the Perl DBI/DBD extensions. The connection to the database can be created at the start of processing, and kept open until the end. For each stock, the applicable rules can be retrieved by selecting from the rules tables based on the stock symbol.

The tag stock_quotes is the outermost tag, meaning its start will trigger the first handler callback, and its end the last. This provides the perfect place for establishing and closing the database connection.

```
sub stock_quotes {
  use DBI;

  $dsn = "DBI:mysql:database=test;";
  $::state::dbh = DBI->connect($dsn);
}

sub stock_quotes_ {
  $::state::dbh->disconnect();
}
```

The rules can be retrieved by selecting based on the stock symbol:

```
my $sth = $::state::dbh->prepare("select * from rules where symbol='$symbol'");
  $sth->execute();

  while (my $ref = $sth->fetchrow_hashref()) {
    # Act on the retrieved rules
  }
  $sth->finish();
```

Acting on the rules
Each stock quote is contained within a stock_quote tag. By the time the end tag for stock_quote is reached, all of the necessary information has been stored (the stock symbol, price, and volume). Thus we can act on the rules in the `stock_quote_` function:

```
sub stock_quote_ {
  my $symbol = $::state::storage{'symbol'};

  # Grab the rules for the given stock from the rules table
  my $sth = $::state::dbh->prepare("select * from rules where symbol='$symbol'");
  $sth->execute();

  while (my $ref = $sth->fetchrow_hashref()) {
    my $field   = $ref->{'field'};
    my $value   = $ref->{'value'};
    if ($::state::storage{$field} > $::state::storage{$value}) {
      # This rule applies
      print "Rule \"$field > $value\" applies for $symbol\n";
      take_action($symbol, $ref->{'action'});
    }
  }
  $sth->finish();
}
```

The applicable rules for the given stock symbol are retrieved, and the comparison is performed. If the rule applies, the `take_action` function is called, which in this case is simply a stub.

The complete program is available as a separate listing, as well as the schema for creating the rules table. Running the program with the original XML file produces the following output:

```
Rule "price > 120.0" applies for IBM
Taking action "buy" on stock "IBM" .
Rule "volume > 65000000" applies for MSFT
Taking action "sell" on stock "MSFT" .
```

Next steps
You can apply these techniques to larger projects, yielding fast and flexible XML-based systems. Solutions built using scripting languages as the transformation and command language -- with high-performance C/C++-based parsers handling the parsing of the XML document -- offer a best-of-breed approach. This approach provides the speed of lower level languages while providing the ease of scripting.

Resources

- finance.yahoo.com provides stock quotes in spreadsheet format.
- The scripts used to perform the conversion, as well as a live XML stock quote server are available from the XML Today Web site.
- Expat is a fully conforming, non-validating XML parser written in C.
- XML::Parser is a Perl interface to James Clark's XML parser, expat.
- XML::Twig is a tree interface to XML documents allowing chunk-by-chunk processing of huge documents.
- XML::DOM is a Perl extension to XML::Parser to build an Object Oriented datastructure with a DOM Level 1-compliant interface.
- XML::Grove provides simple access to the information set of parsed XML, HTML, or SGML instances using a tree of Perl hashes.
- The Document Object Model (DOM) provides a standard set of objects for representing HTML and XML documents, and a standard interface for accessing and manipulating them.
- MySQL Database is a free SQL database available for most operating systems, including most flavors of UNIX, as well as Windows and OS/2.
- *High Performance Web Applications using Perl, XML, and Databases* discusses issues and techniques for building high-performance Web applications using Perl, XML, and databases.

About the author
Parand Tony Darugar, president and co-founder of Binary Evolution Inc., has been involved in Internet-based ventures since 1992. His interests include Internet commerce, database and legacy system to Web connectivity, XML, and artificial intelligence. He can be reached at tdarugar@binev.com.

What do you think of this article?

○ Killer! ○ Good stuff ○ So-so; not bad ○ Needs work ○ Lame!

Comments?

Submit feedback

```
<stock_quotes>

<stock_quote>
 <symbol>IBM</symbol>
 <when>
   <date>12/16/1999</date>
   <time>4:40PM</time>
 </when>
 <price type="ask" value="109.1875"/>
 <price type="open"    value="108"/>
 <price type="dayhigh" value="109.6875"/>
 <price type="daylow"  value="105.75"/>
 <change>+2.1875</change>
 <volume>7050200</volume>
</stock_quote>

<stock_quote>
 <symbol>MSFT</symbol>
 <when>
   <date>12/16/1999</date>
   <time>4:01PM</time>
 </when>
 <price type="ask" value="113.6875"/>
 <price type="open"    value="109.25"/>
 <price type="dayhigh" value="115"/>
 <price type="daylow"  value="108.9375"/>
 <change>+5.25</change>
 <volume>64282200</volume>
</stock_quote>

<stock_quote>
 <symbol>CSCO</symbol>
 <when>
   <date>12/16/1999</date>
   <time>4:01PM</time>
 </when>
 <price type="ask" value="97.875"/>
 <price type="open"    value="98.390625"/>
 <price type="dayhigh" value="98.75"/>
 <price type="daylow"  value="97.0625"/>
 <change>+2</change>
 <volume>21327400</volume>
</stock_quote>

<stock_quote>
 <symbol>INTC</symbol>
 <when>
   <date>12/16/1999</date>
   <time>4:01PM</time>
 </when>
 <price type="ask" value="80.25"/>
 <price type="open"    value="79.75"/>
 <price type="dayhigh" value="81.125"/>
 <price type="daylow"  value="79.375"/>
 <change>+1.3125</change>
 <volume>25059700</volume>
</stock_quote>

<stock_quote>
 <symbol>ORCL</symbol>
 <when>
```

```
   <date>12/16/1999</date>
   <time>4:01PM</time>
 </when>
 <price type="ask" value="89.5625"/>
 <price type="open"    value="89.625"/>
 <price type="dayhigh" value="90.625"/>
 <price type="daylow"  value="87.375"/>
 <change>-0.8125</change>
 <volume>14881500</volume>
</stock_quote>

<stock_quote>
 <symbol>SUNW</symbol>
 <when>
   <date>12/16/1999</date>
   <time>4:01PM</time>
 </when>
 <price type="ask" value="74.9375"/>
 <price type="open"    value="75.625"/>
 <price type="dayhigh" value="77.75"/>
 <price type="daylow"  value="74.375"/>
 <change>+1.5</change>
 <volume>21366600</volume>
</stock_quote>


</stock_quotes>
```

```
# ---------------------------------------------

use vars qw(%startsub, %endsub);

%startsub = (
"stock_quote"         =>      "<hr><p>",
"symbol"              =>      "<h2>",
"price"               =>      "<br><b>Price:</b>",
"date"                =>      "<br><b>Date:</b><i>",
"time"                =>      "<br><b>Time:</b><i>",
"change"              =>      "<br><b>Change:</b>",
"volume"              =>      "<br><b>Volume:</b>"
);

%endsub = (
"stock_quote"         =>      "",
"symbol"              =>      "</h2>",
"price"               =>      "",
"date"                =>      "</i>",
"time"                =>      "</i>",
"change"              =>      "",
"volume"              =>      ""
);

# ---------------------------------------------

use XML::Parser;

my $file = shift;

my $parser = new XML::Parser(ErrorContext => 2);
$parser->setHandlers(Start => \&start_handler,
                     End   => \&end_handler,
                     Char  => \&char_handler);

$parser->parsefile($file);

# ---------------------------------------------
#
# The handlers for the XML Parser.
#

sub start_handler
{
    my $expat = shift; my $element = shift;

    # element is the name of the tag

    print $startsub{$element};

    # Handle the attributes
    while (@_) {
        my $att = shift;
        my $val = shift;
        print "$att=$val ";
    }

}

sub end_handler
```

```
{
    my $expat = shift; my $element = shift;
    print $endsub{$element};
}


sub char_handler
{
    my ($p, $data) = @_;
    print $data;

}
```

```
use XML::Parser;

my $file = shift;

my $parser = new XML::Parser(Style=>'Subs', Pkg=>'SubHandlers', ErrorContext => 2);
$parser->setHandlers(Char  => \&char_handler);

$parser->parsefile($file);

# --------------------------------------------

sub char_handler
{
    my ($p, $data) = @_;
    print $data;

}

# --------------------------------------------

package SubHandlers;

sub stock_quotes {
  print "<title>Stock Quotes</title>\n";
  print "<BODY BGCOLOR=\"#ffffff\" LINK=\"#0000ff\" ALINK=\"#ff0000\" TEXT=\"#000000\">";
}

sub stock_quotes_ {
  print "</BODY>";
}

sub stock_quote {
  print "<table width=100%>\n";
}

sub  stock_quote_ {
  print "</tr>\n</table>\n</td>\n</tr>\n</table>\n";
}

sub symbol {
  print "<tr>\n<td>\n<img src=images/";
}

sub symbol_ {
  print ".gif>\n<br>\n";
}

sub when {
  print "<br>\n";
}

sub when_ {
  print "</td>\n<td>\n<table width=100%>\n<tr>\n";
}

sub price {
  my $expat = shift; my $element = shift;

  # Read the attributes
  while (@_) {
    my $att = shift;
    my $val = shift;
    $attr{$att} = $val;
  }

  my $type  = $attr{'type'};
  my $price = $attr{'value'};

  my $rowbreak = 0;

  if ($type eq 'ask') {
    $label="Ask Price";
  } elsif ($type eq 'open') {
    $label="Opening Price";
    $rowbreak = 1;
  } elsif ($type eq 'dayhigh') {
    $label="Today's High";
  } elsif ($type eq 'daylow') {
    $label="Today's Low";
    $rowbreak = 1;
  }

  print "<td align=left>\n<b>$label</b></td>\n";
  print "<td align=right>$price</td>\n";

  if ($rowbreak) {
    print "</tr><tr>\n";
  }
}

sub price_ {
}

sub change {
  print "<td align=left>\n<b>Change</b>\n</td>\n";
  print "<td align=right>";
}

sub change_ {
  print "</td>\n";
}

sub volume {
  print "<td align=left>\n<b>Volume</b>\n</td>\n<td align=right>";
}

sub volume_ {
  print "</td>\n";
}
```

```
use XML::Parser;

my $file = shift;

my $parser = new XML::Parser(Style=>'Subs', Pkg=>'SubHandlers', ErrorContext => 2);
$parser->setHandlers(Char  => \&char_handler);

$parser->parsefile($file);

# ---------------------------------------------

sub char_handler {
    my ($p, $data) = @_;
    if ($::state::store_contents) {
      $::state::storage{$::state::store_contents} .= $data;
    } else {
      # print $data;
    }
}

# ---------------------------------------------

package SubHandlers;

#
# This function would perform the given action (eg. buy or sell)
# on the given stock. In this case it's simply a stub.

sub take_action {
  my ($symbol, $action) = @_;

  print "Taking action \"$action\" on stock \"$symbol\" .\n";
}


sub stock_quotes {
  # Open connection to database
  use DBI;

  $dsn = "DBI:mysql:database=test;";
  $::state::dbh = DBI->connect($dsn);
}

sub stock_quotes_ {
  # Close connection to database
  $::state::dbh->disconnect();
}


sub stock_quote {
  undef %::state::storage;
}

sub  stock_quote_ {
  # The values have been stored in the hash %state::storage

  my $symbol = $::state::storage{'symbol'};

  # Grab the rules for the given stock from the rules table
  my $sth = $::state::dbh->prepare("select * from rules where symbol='$symbol'");
  $sth->execute();

  while (my $ref = $sth->fetchrow_hashref()) {
    my $field   = $ref->{'field'};
    my $value   = $ref->{'value'};

    if ($::state::storage{$field} > $::state::storage{$value}) {
      # This rule applies
      print "Rule \"$field > $value\" applies for $symbol\n";
      take_action($symbol, $ref->{'action'});
    }
  }
  $sth->finish();

  undef %::state::storage;
}


sub symbol {
  $::state::store_contents = "symbol";
}

sub symbol_ {
  undef $::state::store_contents;
}

sub date {
  $::state::store_contents = "date";
}

sub date_ {
  undef $::state::store_contents;
}

sub time {
  $::state::store_contents = "time";
}

sub time_ {
  undef $::state::store_contents;
}

sub price {
  my $expat = shift; my $element = shift;

  # Read the attributes
  while (@_) {
    my $att = shift;
    my $val = shift;
    $attr{$att} = $val;
  }

  if ($attr{'type'} eq "ask") {
    $::state::storage{'price'} = $attr{'value'};
  }
}

sub price_ {
}

sub volume {
  $::state::store_contents = "volume";
}

sub volume_ {
  undef $::state::store_contents;
}
```

```
CREATE TABLE rules (
        symbol  CHAR(5),
        field   CHAR(8),
        value   CHAR(16),
        action  CHAR(5)
);

INSERT INTO rules VALUES ("IBM", "price", "120.0", "buy");
INSERT INTO rules VALUES ("MSFT", "volume", "65000000", "sell");
INSERT INTO rules VALUES ("CSCO", "price", "100.0", "sell");
```