isCOBOL Evolve: JOE

Java Objects Executor: a polymorphic script environment

Key Topics:

- Basic syntax
- Variables and literals
- Blocks
- Outer Blocks
- Control transfer

Introduction

isCOBOL programs can be distributed from a server with isCOBOL Application Server: this is the architecture of choice for multiuser applications due to its many benefits, however, in order to get the best results, it requires all the processing made using isCOBOL environment.

Many legacy applications are made intermixing COBOL programs and interpreted scripts of some kind (e.g. Bourne shell); in order to get these applications running in isCOBOL Application Server is currently necessary to translate the scripts into COBOL programs and compile them.

This process can be lengthy and in any case the result could be less appealing than the starting point because:

- interpreted procedures are now compiled procedures;
- procedures written with a language oriented to manage operating system tasks are now written using COBOL (a Business Oriented Language).

So, in order to speed up the migration process getting at the same time a better result, it would be useful to have a scripting language whose features are:

- ability to access any isCOBOL/Java resource: isCOBOL (as well as Java) is operating system independent therefore the Java environment is its virtual operating system;
- easy to change in order to get similar to any script language in terms of capability and readability;
- easy to customize in order to get frequently used operations at hand;
- easy to extend in order to be useful also for future applications' enhancements, not only for the migration process;
- easy to understand and use;
- 100% compatible with the isCOBOL Application Server architecture.

The Java Objects Executor (JOE for short) complies with the above requests.

JOE's only task is to execute methods of Java/isCOBOL objects in sequence on the fly: how it can be used to mimic any scripting language it will become clearer later.

Getting Started

JOE is installed along with isCOBOL.

You can run it interactively with the following command:

```
iscrun -joe
```

The shell is started and waits for commands:

```
CobShell interactive ready, type 'exit' to exit the session

cs>
```

In order to run a script you must supply the script name as argument, e.g.

```
iscrun -joe myscript.joe
```

JOE scripts can also be edited and executed in the isCOBOL IDE.

To edit a script within the IDE, add it to your project and open it with the JOE Editor.

To run a script from the IDE, right click on the script name in the File View and choose *Run As > Joe Application*. The output is shown in the Console view.

Basic syntax

In order to invoke methods we need an object oriented syntax: the obvious choice could be the COBOL OO syntax or, as an alternative, the Java syntax. We choose instead to use a syntax close to Smalltalk since it has a better readability, especially when used extensively. The JOE syntax for invoking a method is:

```
[ variable-name := ] object [method-name { ; | argument[,argument…]} …] .
```

In JOE any data is an object and any invocation is supposed to return an object. You can use variables to hold objects; a variable doesn't need to be declared and has no type, it can contain any type of object.

JOE has not reserved words, but it has few reserved symbols:

:=	assignment
;	no argument
,	parameter separator
.	end of the message

The basic core of JOE is the triplet object-method_name-argument. If you want invoke a method with no argument then this must be explicitly stated in the code: the meaning of the semicolon character is exactly this, it means 'there are no arguments'.

Let's see some examples of this syntax compared with the equivalent Java syntax in order to get a better understanding.

JOE syntax	Java syntax	
A B C.	A.B(C);	Basic method invocation, A is an object, B is a method name and C is an object passed to the method as argument.
A B C D E.	A.B(C).D(E);	A is an object, B is a method name and C is an object passed to the method as argument. This invocation returns an object, so D is a method name and E its argument and so on. There is no theoretical limit to the length of a message.
A B; D E.	A.B().D(E);	This case differs from the previous one in that no argument is passed to method B.
A B C,D,E.	A.B(C,D,E);	In this case instead 3 arguments are passed to method B.
A B (C D E).	A.B(C.D(E));	The evaluation order can be altered using the parentheses (), which allow to execute a method and use its result as argument of another method. In this case the method D of object C is executed and its result is passed as argument to method B of object A.

You can assign the result of the last invocation in a message to a variable, e.g.:

```
var := A B C D E.
```

JOE supports two types of comments:

- in-line comments. JOE ignores everything from *> to the end of the line, e.g.:

```
*> this is a comment
```

- multi-line comments, JOE ignores everything from /* to */, e.g.:

```
/*
 This is a comment
 distributed on
 multiple lines
*/
```

Moreover a line starting with the sequence #! is ignored in order to support the shebang interpreter directive of the Unix-like operating systems.

Since there are no built-in instructions, in order to run something JOE needs an object that act as starting point. This object, let's call it 'command', is automatically loaded at the beginning of the execution. Since the command object is supposed to have useful and often used methods, it has been named '!' (the character for exclamation mark or bang) so that you need to type only one character and it is easy to see it in the source code.

The key point is that this command object has nothing special, it is a plain isCOBOL/Java object accessed using the Java reflection: you can write your own version if you like, inheriting the behaviors from the supplied one or even creating a brand new environment.

You can find a list of the currently available methods in Javadoc documentation.

We are now ready to do the very first program using JOE. We are going to use CobShell, a version of JOE that has been made similar to COBOL. The string "cs> " is the prompt and it is not part of the commands.

So the first program is the classic "Hello".

```
cs> ! display "Hello #1".
Hello #1
cs>
```

You can see the triplet object-method_name-argument very clearly here. The COBOL syntax would be *invoke command "display" using "Hello #1"* while the Java syntax would be *command.display("Hello #1")*. Since the bang cannot be used in a JOE name, you can also write:

```
cs> !display "Hello #1".
Hello #1
cs>
```

The meaning is the same as the previous one.

The method *display* accepts any number of parameters, shows them, issue a new line and returns the command object itself. If the method is invoked without parameters, only a new line is issued. There is an equivalent method, *displayNoAdv*, that does the same things without issuing a new line.

So you can get the same result with the following line:

```
cs>!display "Hello #",1.
Hello #1
cs>
```

More than one invocation can be concatenated: when it happens the first triplet is executed and the result is the object of a second triplet and so on until the line is closed. The dot character used to stop the evaluation. So you can issue the following line:

```
cs> !display "Hello #",1 display "Hello #",2.
Hello #1
Hello #2
cs>
```

Note that the second *display* doesn't need the bang since the command object is returned by the first one. Another example is:

```
cs> ! display "Hello #",1 display; display "Hello #",2.
Hello #1

Hello #2
cs>
```

Note that the second *display* is followed by a semicolon in order to inform the interpreter that that method has no arguments.

The evaluation is done left to right but you can change the evaluation order by enclosing the object expression to evaluate before between parenthesis. In order to see that, it comes handy to know that a literal string is an object itself and it is equivalent to the Java String object, therefore it has the method *length* that returns the length of the string. So you can issue the following line:

```
cs> !display "Length",("Length" length;).
Length6
cs>
```

The *length* method is executed before the *display* method and the result is used as parameter by it.

In this case you can avoid the use of the semicolon since the parameters of a method cannot be placed after a closed parenthesis.

Variables and literals

JOE allows you to store an object reference in a variable through the symbol :=. A variable name consists in a sequence of characters that are not reserved for other uses, as space, ! etc. (the exact set is still to be defined, the Java names will be valid for sure). The variables are not typed, so you can use them for any kind of object. They can also change type during the execution. For example:

```
cs> a := "Length".
cs> !display a,(a length).
Length6
cs> a := !.
cs> a display "Hello!".
Hello!
cs> a := "Length" length.
cs> ! display a.
6
cs>
```

If a variable is used without any previous assignment, its value will be null.

```
cs> !display b.
(null)
cs>
```

Currently JOE manages only three types of literals, integer numbers, floating point double precision numbers and strings. They are objects equivalent to java.lang.Integer, java.lang.Double and java lang.String but they are actually wrapped in internal objects in order to get more functionalities. For example the Java String allows you to easily see if two objects are equal through the method *equals*, however if you want compare two instances of String in order to know which is the greater, you need to use the method *compareTo* and then look for the result. The JOE wrapped objects all have the methods *gt*, *ge*, *lt*, *le*, *ne* that allow you to easily compare to literals of the same kind, e.g.:

```
cs> ! display ("A" gt "B").
false
cs> ! display ("A" lt "B").
true
cs>
```

Numbers are wrapped as well in order to get all the arithmetic operation at hand. JOE doesn't need the use of arithmetic operators characters nor logic operators characters, so they are automatically translated in words according the following list:

=	equals
<	lt
>	gt
<=	le
>=	ge
<>	ne
+	add

-	subtract
*	multiply
/	divide
%	mod

This translation allows you to issue invocations as the following:

```
cs> ! display ("A" > "B").
false
cs> ! display ("A" < "B").
true
cs> ! display (1 + 2).
3
cs> ! display (1 + 2 * 3).
9
cs> ! display (1 + (2 * 3)).
7
cs>
```

Note that the common arithmetic operations precedence is not respected, the evaluation is always left to right and if you want to change it you have to use the parenthesis.

Any time you use a literal to call an external object, it is converted into the correspondent Java object and the returned object is converted to the internal object when needed. In order to see that, you must know that the supplied command object has the method *newInstance* that allows you to instance any isCOBOL/Java object. The following example shows you how standard Java object can be handled by JOE:

```
cs> bd1 := !newInstance "java.math.BigDecimal","5.0".
cs> bd2 := !newInstance "java.math.BigDecimal",7.
cs> ! display "bd1=", bd1, "; bd1^2=", (bd1 pow 2).
bd1=5.0; bd1^2=25.00
cs> ! display "bd1 scale=",(bd1 scale).
bd1 scale=1
cs> ! display "bd1+bd2=", (bd1 + bd2).
bd1+bd2=12.0
cs>
```

You can also create your own objects and easily handle them in the JOE environment; let's say you want to handle dates in your procedures, you could write a Java class like the following one:

```
import java.util.Date;
public class MyDate {
   private static long msPerDay = 1000 * 60 * 60 * 24;
   private final java.util.Date date;
   public MyDate(long time) {
      date = new java.util.Date(time);
   }
   public MyDate(int year, int month, int day) {
      date = new java.util.Date(year - 1900, month - 1, day);
   }
   public long subtract (MyDate d) {
      return (date.getTime() - d.date.getTime()) / msPerDay;
   }
   public MyDate subtract (int days) {
      return new MyDate (date.getTime() - (days * msPerDay));
   }
   public MyDate add (int days) {
      return new MyDate (date.getTime() + (days * msPerDay));
   }
   public boolean equals (Object d) {
      if (d instanceof MyDate)
         return date.equals (((MyDate) d).date);
      else
         return false;
   }
   public boolean lt (MyDate d) {
      return date.before (d.date);
   }
   public boolean gt (MyDate d) {
      return date.after (d.date);
   }
   public String toString()  {
      return date.toString();
   }
}
```

After compiling this class and having it accessible through CLASSPATH, you can issue the following invocations:

```
cs> amRev := !newInstance "MyDate",1775,04,19.
cs> frRev := !newInstance "MyDate",1789,05,05.
cs> ! display "American revolution start=",amRev.
American revolution start=Wed Apr 19 00:00:00 CET 1775
cs> ! display "French revolution start=",frRev.
French revolution start=Tue May 05 00:00:00 CET 1789
cs> ! display "years between the revolutions=",((frRev - amRev) / 365).
years between the revolutions=14
cs> ! display (frRev > amRev).
true
cs> ! display (frRev < amRev).
false
cs> ! display (amRev = (!newInstance "MyDate",1775,04,19)).
true
cs> ! display "15 days after=",(amRev + 15).
15 days after=Thu May 04 00:00:00 CET 1775
cs> ! display "15 days before=",(amRev - 15).
15 days before=Tue Apr 04 00:00:00 CET 1775
cs>
```

JOE doesn't handle arrays however you can get the same behavior through the use of objects. For example the standard command implements the method *array* that returns the equivalent of a Java array containing the arguments as elements, e.g.:

```
cs> myArray := !array 1,"two",3.0.
cs> !display (myArray get 0).
1
cs> !display (myArray get 1).
two
cs> !display (myArray get 2).
3.0
cs>
```

The method *set index,value* allows you to set a value in an array. Since an element of an array can be any type of object, you can have elements that are arrays themselves, recreating the behavior of multidimensional arrays.

At this point you should see that, through the application of few simple rules, you can get an easy-to-use powerful environment customized on your needs. However, in order to get a complete language, it is necessary to have some decisional control structure. We need then to introduce a further concept, the Block.

Blocks

A block is simply a list of invocations enclosed between braces. It is an object itself so you can assign it to a variable, e.g.:

```
cs> a := { b := 2. ! display (b + 1). }.
cs>
```

The block content is not executed, it is only stored; since it is an object, in order to execute its content, you only need to invoke its method *exec*.

```
cs> a := { b := 2. ! display (b + 1). }.
cs> a exec.
3
cs>
```

The method *exec* of a block returns the result of the last invocation; in the case above it will return the result of the display, i.e. the command object.

```
cs> a := { b := 2. ! display (b + 1). }.
cs> a exec; display "end".
3
end
cs>
```

(Note the use of the semicolon character in order to inform the interpreter that exec has no parameters)

The blocks allow to easily implement a method that issue the behavior of an "if" statement: the following Java method is the implementation issued in the supplied command object:

```
public Object $if (Boolean cond, Block ifTrue) throws Exception {
   Object Return = cond;
   if (cond.booleanValue()) {
      Return = ifTrue.exec();
   }
   return Return;
}
```

You can note that the name of this method is *$if* : Java doesn't allow to have methods names equal to a reserved word, so when the interpreter recognizes a method name that equals a Java reserved word, it automatically prefixes the method name with the character "$".

Now you can issue an invocation like the following one:

```
cs> a:=1. b:=1. !if (a=b),{!display "a=b".}.
a=b
cs>
```

The "else" behavior can be achieved with a further method, similar to the previous one:

```
public Object $if (Boolean cond, Block ifTrue, Block ifFalse)
                                                    throws Exception {
   Object Return;
   if (cond.booleanValue()) {
      Return = ifTrue.exec();
   } else {
      Return = ifFalse.exec();
   }
   return Return;
}
```

As an example:

```
cs> a:=1. b:=2. !if (a=b),{!display "a=b".},{!display "a<>b".}.
a<>b
cs>
```

In the above example all the code is written on a single line, you can improve the readability writing it on multiple lines. CobShell can be executed with a text file name as parameter and in such a case the content of the file is executed. You can then write the above example in the following way:

```
a:=1.
b:=2.
!if (a=b),{
   !display "a=b".
},
{
   !display "a<>b".
}.
```

Blocks are used also to perform loops: the method *until* execute a block until the specified condition (included in a block) is true. For example:

```
cs> a:=0. !until { a=5 },{ a := a + 1. !display "a=",a. }. !display "end".
a=1
a=2
a=3
a=4
a=5
end
cs>
```

The condition must be included in a block because the condition must be re-evaluated at the beginning of each cycle. Since the execution of a block returns the result of the last invocation, the above example can also be written in the following way:

```
cs> a:=0. !until { a:=a+1. a>5 },{ !display "a=",a. }. !display "end".
a=1
a=2
a=3
a=4
a=5
end
cs>
```

(Note that in thes case the condition is a>5 instead of a=5 : this because the increment of the variable is issued before the evaluation of the condition instead of inside the second block).

At this point you have a complete language with all the necessary features. A subroutine can be implemented as a block, assigned to a variable and executed when needed.

The following example is a procedure that guesses a user thought number and summarizes what has been seen so far.

```
answer := "".
high := 1023.
low := 1.
ntry := 1.

!display "Think to a number between ",low," and ",high,
                                    ": I can guess it using 10 tries at most".

!until { answer = "c" },
{
   try := ((high - low) / 2 + low).
   !display "My guess is ", try.
   !display "Is the guess (c)orrect, too (h)igh or too (l)ow?".
   answer := !accept.
   !if (answer = "c"), {
      !display "I guessed the number using ",ntry," guesses".
   } , {
      !if (answer = "h"), {
         high := try.
         ntry := ntry + 1.
      } , {
         !if (answer = "l"), {
            low := try.
            ntry := ntry + 1.
         } , {
            !display "Answer with 'c', 'h' or 'l' please".
         }
      }
   }
}.
```

It is possible to achieve the behavior of more complex statements, like a multi-way branch similar to the COBOL EVALUATE statement.

The *evaluate* method takes an object as an argument and returns an object that has the method *when* that typically has two arguments, an object and a block:

if the argument is equal to the one specified in evaluate then executes the block, updates its state and returns itself in the event of further invocations of *when*.

The object used to implement this feature has its own internal state that allows the execution only of the first block that satisfies the condition.

The method *when* can also be invoked without specifying any block, in which case the condition of equality is still checked and put in OR with the next invocation of *when*.

The *when_other* method takes a block as an argument that runs only when no other block has been executed previously.

The method *end_evaluate* finally makes sure that the result of the last run is returned by the evaluate at the end of all the invocation.

Here is the previous example implemented by the using of the *evaluate* method:-

```
answer := "".
high := 1023.
low := 1.
ntry := 1.

!display "Think to a number between ",low," and ",high,
                              ": I can guess it using 10 tries at most".

!until { answer = "c" or (answer = "C")}, {
   try := ((high - low) / 2 + low).
   !display "My guess is ", try.
   !display "Is the guess (c)orrect, too (h)igh or too (l)ow?".
   answer := !accept.
   !evaluate answer
   when "C"
   when "c", {
      !display "I guessed the number using ",ntry," guesses".
   }
   when "H"
   when "h", {
      high := try.
      ntry := ntry + 1.
   }
   when "L"
   when "l", {
      low := try.
      ntry := ntry + 1.
   }
   when_other {
      !display "Answer with 'c', 'h' or 'l' please".
   }
   end_evaluate.
}.
```

A side-effect of the implementation above described is that you can write an equivalent multi-way branch using a notation that is characteristic of the COBOL EVALUATE, i.e:

```
   !evaluate (1 = 1)
   when (answer = "c" or (answer = "C")), {
      !display "I guessed the number using ",ntry," guesses".
   }
   when (answer = "h" or (answer = "H")), {
      high := try.
      ntry := ntry + 1.
   }
   when (answer = "l" or (answer = "L")), {
      low := try.
      ntry := ntry + 1.
   }
   when_other {
      !display "Answer with 'c', 'h' or 'l' please".
   }
   end_evaluate.
```

You can see how complex behaviors can be achieved using the simple mechanism object-method-args.

The code inside a block can access any variable already used outside the block, however if you use a variable in a block for the first time, it will be not available outside, i.e. that variable will be local to the block e.g.:

```
cs> a := { b := 2. ! display (b + 1). }.
cs> a exec.
3
cs> ! display b.
(null)
cs>
```

As said above the key point is that you can write your own command object in order to customize the scripts as you wish. Let's say you want to do loops using a command similar to the Java style "for", i.e. with an initialization, a condition and an increment: you can write a Java class like the following one:

```
public class MyCommand {
   public Object $for (Block init,
                       Block cond,
                       Block incr,
                       Block code) throws Exception {
      Object Return = null;
      init.exec();
      while ((Return=cond.exec()) != null &&
              Return instanceof WBoolean &&
              ((WBoolean) Return).booleanValue()) {
         Return = code.exec();
         incr.exec();
      }
      return Return;
   }
}
```

Assuming you have your class "MyCommand" available in your CLASSPATH, you can issue messages like these:

```
cs> mycmd := !newInstance "MyCommand".
cs> i := 0.
cs> mycmd for {i := 1},{i < 5},{i := i + 1},{!display i}.
1
2
3
4
cs>
```

Note that the variable *i* must be used outside any block otherwise it will be local to the block itself.

A block may have an internal name and arguments, they can be specified immediately after the open braces. The format is:

```
[name] : [arg1 [,arg2 …]].
```

These are some valid block definitions:

```
cs> a := {aName:anArg. !display "Name & arguments". }.
cs> b := {aName:. !display "Just the name". }.
cs> c := {:a1,a2. !display "Arguments only" }.
cs> d := {:. !display "Useless".}.
```

You can supply any number of argument to a block, if the argument is not supplied then the correspondent variable will contain the null value.

```
cs> blk := {:a,b. !display a,";",b.}.
cs> blk exec 1.
1;(null)
cs>
```

Blocks allows recursion, below is a script that compute the factorial of the given number.

```
fact :=
{:n.
   !if (n > 1), {
      n * (fact exec (n - 1)).
   } , {
      1.
   }.
}.

!display (fact exec 6).
```

The internal name can be used in order to cause a forced exit from the block. For example the default command implements the method *exit_block "internal-name"*. The above example can also be implemented in the following way:

```
fact :=
{all:n.
   !if (n <= 1), {
      1.
      !exit_block "all".
   }.
   n * (fact exec (n - 1)).
}.

!display (fact exec 6).
```

Note that it is not practical to use this approach in order to exit from a loop; consider the following example:

```
cs> i := 0.
cs> !until {i:=i+1. i = 3},{loop:. !display i. !exit_block "loop". !display "never
printed".}.
1
2
cs>
```

You can see that in this case the *exit_block* method interrupts the block execution but it is executed again since the exit condition is in another block. For this reason the *exit_loop* method has been implemented in the default command, e.g.:

```
cs> i := 0.
cs> !until {i:=i+1. i = 3},{!display i. !exit_loop. !display "never printed".}.
1
cs>
```

In this case the inner loop is interrupted, without the need for the block to have a name.

Outer Blocks

All the source code is implicitly contained in an "outer block". It is an ordinary block with few more features, i.e.:

- you can refer to it through the use of the special sequence "!!";
- you can use it as an object whose methods are the variables referring to a block.

So you can write:

```
cs> a := { b := 2. ! display (b + 1). }.
cs> !!a.
3
cs>
```

Note that if you try to execute !! exec, a variable whose name is "exec" will be searched and, if it exists and it refers to a block, the corresponding block will be executed.

An outer block can have both a name and arguments in the way that ordinary blocks have. When an outer block is executed by the command line, it receives an argument that is an array whose elements are the command line broken by spaces. For example let's say you have the following script named "args.joe":

```
:args.

i := -1.
!until {i := i + 1. i = (args length)},
{
   !display (args get i).
}.
```

you can issue the following command:

```
$ iscrun -joe args.joe 6 aa bb cc
args.joe
6
aa
bb
cc$
```

A script can also be executed from inside another script through the method new implemented in the default command. The script will be executed and its status (i.e. the variables) will be saved. For example let's say you have the following script named "average.joe":

```
:i_cnt,i_avg.

cnt := i_cnt doubleValue.
avg := i_avg doubleValue.

put := {:val.
   avg := avg * cnt + val.
   cnt := cnt + 1.
   avg := avg / cnt.
   !!.
}.

get := { avg. }.
```

You can use it to compute the average of a series of numbers, for example:

```
cs> avg := !new "average.joe",0,0.
cs> avg put 8 put 13 put 21 put 34 put 55.
cs> !display (avg get).
26.2
cs>
```

So a JOE script can be seen as an object from inside another JOE script. The code outer of any block is useful for initializing the object, as a Java constructor. In this object any variable containing a reference to a block will be equivalent to a public method while the other variables will be private (or rather protected, as explained later).

A JOE script can also inherit from another script through the default command method *extends*: this means the inheriting script will see all the variable from the parent script. The *extends* method has 2 arguments, i.e. the inheriting outer block and the parent outer block. For example:

```
cs> !extends !!,(!new "average.joe",0,0).
cs> !! put 8 put 13 put 21 put 34 put 55.
cs> !display avg.
26.2
cs> !display cnt.
5.0
cs>
```

So JOE has the features of a dynamic typed object oriented language using a simple model and simple implementation.

Control transfer

Even if now we can achieve any kind of computation, it could be hard to translate older script languages in it. For this reason some few features has been added, i.e. one-way transfer of control to another line of code (GO TO) and the transfer of control to another line of code with return (similar to a COBOL PERFORM).

In order to allow the transfer of the control to a specified line of code, it is necessary a way to identify a line of code: this is achieved through the use of labels: a label is simply a word followed by a dot. A label can be placed everywhere, however only the labels outside of any block can be referenced. The command object has 3 methods that manage the transfer control, i.e.:

goto	one-way control transfer; it accepts one parameter that can be either a label name or a string.
perform	control transfer with return at the original point; the return is issued when an exit is encountered; it accepts one parameter that can be either a label name or a string.
exit	causes the control to be returned where the last perform has been issued: if it is invoked outside of a perform, the procedure ends. It has no parameters.

The following example shows how the previous procedure for guessing a number can be implemented using control transfer.

```
answer := "".
high := 1023.
low := 1.
ntry := 1.

!display "Think to a number between ",low," and ",high,
                      ": I can guess it using 10 tries at most".
begin.
   try := ((high - low) / 2 + low).
   !display "My guess is ", try.
   !display "Is the guess (c)orrect, too (h)igh or too (l)ow?".
begin1.
   !perform ask.
   !if (answer = "c"), {
       !display "I guessed the number using ",ntry," guesses".
       !goto end.
   }.
   !if (answer = "h"), {
       high := try.
       ntry := ntry + 1.
       !goto begin.
   }.
   !if (answer = "l"), {
       low := try.
       ntry := ntry + 1.
       !goto begin.
   } , {
       !display "Answer with 'c', 'h' or 'l' please".
       !goto "begin1".
   }.

ask.
   answer := !accept.
   !exit.

end.
   !exit.
   !display "this is never executed".
```