Jess and Java

Jess can be used to call Java code.

There are various reasons which justify calling Java from Jess:

- Overcome limitations of Jess by using Java.
- Link Jess to existing Java code.
- Access Java API specialised libraries.

Creating Java Objects

The Jess function `new` is used to create Java objects from within Jess code.

Jess> (bind ?x (new java.util.Date))

or

Jess> (import java.util.*)

TRUE

Jess> (bind ?x (new Date))

Examples:

(bind ?y (new ArrayList))

; an empty String

(bind ?z (new String))

; a string containing the characters "abc"

(bind ?v (new String "abc"))

The third example above, shows how to pass arguments to the constructor of a class.
Calling Methods on Objects

The call function is used to call a method on a Java object. Arguments to a method, follow the name of the method.

(bind ?y (new ArrayList))

; put an element in the ArrayList
(call ?y add "dog")

; put an additional element in the ArrayList
(call ?y add "cat")

; get the second element of the ArrayList
(call ?y get 1)

The call symbol can be omitted, because Jess assumes that if the first atom of a list is an object, an attempt to call a method on it is made:

(?y add "dog") ; equivalent to: (call ?y add "dog")

Conversion of Jess types to Java types

When a Java method is called, Jess converts the arguments according to the table below:

<table>
<thead>
<tr>
<th>Jess types</th>
<th>Java types</th>
</tr>
</thead>
<tbody>
<tr>
<td>RU.EXTERNAL_ADDRESS</td>
<td>The wrapped object</td>
</tr>
<tr>
<td>nil</td>
<td>null</td>
</tr>
<tr>
<td>TRUE, FALSE</td>
<td>String, java.lang.Boolean or boolean</td>
</tr>
<tr>
<td>RU.ATOM (a symbol), RU.STRING</td>
<td>String, char or java.lang.Character</td>
</tr>
<tr>
<td>RU.FLOAT</td>
<td>float, double and their wrappers</td>
</tr>
<tr>
<td>RU.INTEGER</td>
<td>long, short, int, byte, char and their wrappers</td>
</tr>
<tr>
<td>RU.LONG</td>
<td>long, short, int, byte, char and their wrappers</td>
</tr>
<tr>
<td>RU.LIST</td>
<td>A Java array</td>
</tr>
</tbody>
</table>

Calling Static Methods

The call function can be used with the name of the class.

;; pause for 3 seconds
(call Thread sleep 3000)

In this case, the call symbol cannot be omitted.
Calling get and set Methods

Jess includes the **get**, **set** functions, which can be used instead of **call** to invoke a Java method.

For example, consider the following Java class:

```java
class A {
    int x;
    void setX(int m) {
        x = m;
    }
    int getX() {
        return x;
    }
}
```

Then:

```lisp
(bind ?a (new A))
(get ?a x)
(set ?a x 123)
```

Accessing Java data members

Instance fields of a Java object can be accessed using the Jess functions **get-member**, **set-member**.

```lisp
(bind ?pt (new java.awt.Point))
(set-member ?pt x 100)
```

Static fields are accessed in the same way:

```lisp
(bind ?s (get-member System out))
(bind ?t (new String "abc"))
```

;; call System.out.println() on string containing "abc"
```

Dealing with Exceptions

When a Java method throws an exception, Jess displays a stacktrace with information about it.

Jess> (call Integer parseInt "abc")
Jess reported an error in routine call
while executing (call Integer parseInt "abc").
Message: Called method threw an exception.
Program text: ( call Integer parseInt "abc" ) at line 10.
Nested exception is:
For input string: "abc"
java.lang.NumberFormatException: For input string: "abc"
at java.lang.NumberFormatException.forInputString
    (NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:447)
at java.lang.Integer.parseInt(Integer.java:497)
......
```
Catching Exceptions
Jess code can catch the Java exceptions so as to provide descriptive error message to the non-technical user.

(deffunction parseInt (?st)
  (try
    ;; call the Java Integer.parseInt() method
    (bind ?i (call Integer parseInt ?st))
    (printout t "The answer is: " ?i crlf)
    catch
    (printout t "Invalid argument" crlf))

Now:
Jess> (parseInt "10")
The answer is: 10
Jess> (parseInt "1b")
Invalid argument

Practical Examples with Lists
Example 1:
Define a function which calculates the difference between two lists provided as arguments. Given the lists (a, b, c, d) and (b, c, e), the function should return (a, d).

(deffunction difference (?a ?b)
  (foreach ?e ?a
    (if (not (member$ ?e ?b)) then
      (printout t ?e " ")))

Example 2:
Implement a function accepting two lists as arguments and returns a list which is the concatenation of them.

(deffunction concat (?a ?b)
  ; create an empty list
  (bind ?result (create$))
  (foreach ?e ?a
    ;; insert all elements of ?a in the end of the new list
    (bind ?result (insert$ ?result (+ (length$ ?result) 1) ?e)))
  (foreach ?e ?b
    ;; insert all elements of ?b in the end of the new list
    (bind ?result (insert$ ?result (+ (length$ ?result) 1) ?e)))
  (return ?result))

Example 3:
Write a function substitute with three arguments a, b and l that returns the list l with all occurrences of a in l replaced by b.

(deffunction substitute (?a ?b ?l)
  ; create an empty list
  (bind ?result (create$))
  (foreach ?e ?l
    (if (eq ?e ?a) then
      (bind ?result (insert$ ?result (+ (length$ ?result) 1) ?b))
    else
      (bind ?result (insert$ ?result (+ (length$ ?result) 1) ?e))))
  (bind ?l ?result))
**Example 4:**

Write a function `cartesian-product` which given two lists of atoms returns a list of pairs of atoms such that each element of the first list is paired with each of the second. E.g., `(cartesian-product (a b) (c d))` returns `(a c a d b c b d)`.

```lisp
defunction cartesian-product (?a ?b)
    ; create an empty list
    (bind ?result (create$))

    (foreach ?e1 ?a
        (foreach ?e2 ?b
            ;; create a list for pair (?e1 ?e2) and
            ;; insert it into ?result
            (bind ?p (create$ ?e1 ?e2))
            (bind ?result (insert$ ?result
                (+ (length$ ?result) 1)?p)))))
```

**Example 5:**

Define a function `inner-product` to compute the inner product of two lists. For example, `(inner-product (2 3) (4 5))` should calculate the value $2 \times 4 + 3 \times 5 = 23$. The function should return an error message if the number of elements in the two lists is not the same.

```lisp
defunction inner-product (?a ?b)
    (bind ?sum 0)
    (if (neq (length$ ?a) (length$ ?b)) then
        (printout t "The two lists do not have the same length" crlf)
    else
        (bind ?n (length$ ?a)) ;; calculate the length of the lists
        ;; calculate inner product
        (bind ?i 1)
        (while (<= ?i ?n) do
            (bind ?product (* (nth$ ?i ?a) (nth$ ?i ?b)))
            (bind ?sum (+ ?sum ?product))
            (bind ?i (+ ?i 1)))

        (return ?sum))```

**Example 6:**

Write a function which which accepts a list as an argument. The function returns `TRUE` if the list contains exactly the numbers 1, 2, 3, ..., n in the list, where n is the length of the list.

I.e. the numbers should appear only once in the list. The list is NOT sorted.

For example, given the list `(2 1 3 4 5)` the function should return `TRUE`, but it should return `FALSE` for the list `(1, 2, 3, 4, 3)` as the latter contains 3 twice.
(deffunction uniquen (?a)
    ; calculate the length of the list
    (bind ?n (length$ ?a))

    (bind ?i 1)
    (while (<= ?i ?n) do
        ;; check that ?i appears only once in ?a
        (bind ?count 0) ;; holds how many times ?i appears in list
        (foreach ?e ?a
            ;; make sure element ?e is in the valid range 1 to ?n
            (if (or (< ?e 1) (> ?e ?n)) then
                (return FALSE))
            (if (eq ?e ?i) then
                (bind ?count (+ ?count 1)))
        )
        ;; increase ?i by 1
        (bind ?i (+ ?i 1))
    ) ;; end of while

    (if (neq ?count 1) then
        (return FALSE))
    (return TRUE))