UNIVERSITY OF CALABRIA
Master of Science Program in Computer Engineering
Formal Languages Course

The CalcuList Release 4 Tutorial
Version 3.5

Lecture Notes by

Domenico Saccà
Full Professor of Computer Engineering

Table of Contents

1. Introduction to CalcuList

The *functional programming paradigm*, a form of *declarative progra*mming, was explicitly created to support a pure functional approach to problem solving. In contrast, most mainstream languages, including object-oriented programming languages such as C++, and Java, were designed to primarily support *imperative programming*. With an imperative approach, a developer writes code that describes in detail the steps that the computer must take to accomplish the goal, i.e., a program consists of a sequence of commands, which are executed strictly one after the other. In contrast, a functional approach involves composing the problem as a set of functions to be executed: the developer must define carefully the input to each function, and what each function returns.

A recent trend in programming languages is a renewed interest towards functional programming, particularly in combining it with other paradigms, mainly imperative programming. The new language *CalcuList* ("Calculator with List manipulation") is a simple functional programming language for teaching the functional paradigm, suitably extended with some imperative features involving side effects. The programmer may use CalcuList as a *pure* functional language, in that side effects are required to be explicitly enabled. The use of side-effects, in some cases, simplifies the developed code or even results in a better computational efficiency.

CalcuList syntax is rather similar to that of Python and natively supports processing of strings, lists and JSON objects. The language is strongly typed but most of the type checking is done at run-time. An interactive computation session with the user is established by means of a REPL (Read-Evaluate-Print-Loop) shell. During a session, a user may define a number of global variables and functions and run suitable computations on them as queries, that are computed and displayed on the fly. CalcuList expressions and functions are first compiled and then executed each time a query is issued. CalcuList can be thought of as an ``interactive compiler'' rather than an interpreter and both static and dynamic type-checking are performed. The object code produced by a compilation is a program that will be eventually executed by the CalcuList Virtual Machine (CLVM).

It is stressed that CalcuList is a didactical "toy" tool for a course on Formal Languages and has no ambitions to be a general-purpose language, even though it is Turing complete (i.e., it may define all Turing–computable functions).

CalcuList is invoked by typing the command "java –jar CalcuList_R_4_0_5.jar" to the shell[1]. Then the following welcome message will be printed:

```
$java –jar CalcuList_R_4_0_5.jar
     ***********************************************
                    ***  CalcuList   ***
     ***********************************************
     ******** Release 4.0.5 of August 7, 2017 ********

** Importing file "Utilities.cl" **
** Import ended **
-- 0 imported commands
>>
```

[1] The jar file name refers to the release R.4.0.5. The suffix "R_4_0_5" in the name changes if a new release is used. As illustrated in Section 6.1, it is possible to include a number of arguments at the end of the *java* command to increase the sizes of some data structures, that are internally used and have been dimensioned for a typical usage of CalcuList.

At the start CalcuList imports possible pre-defined commands from the text file “Utilities.cl”, that must be stored into the current working directory. In the example the file does not contain any command. The cursor is located next to the prompt “>>” waiting for a command to be entered by the user. The classical example of a first command for a language can be issued as follows:

```
>>^"Hello World";
Hello World
>>
```

The symbol “^” states that the user is issuing the query consisting of displaying a quoted string. A command as well as any definition written by the user must be ended by “;” and can span over various lines. The result of a query will be displayed on a new line once a semicolon has been entered. To finish a CalcuList session, the command “halt” must be typed.

```
>>halt /* comment: halt closes the session */
Bye
```

To facilitate the reading, in all examples the input provided by the user is blue colored. The user may also include a comment, which starts with the string “/*” and ends with “*/”, possibly spanning over various line – comments will be red colored.

The Tutorial is organized as follows. Section 2 introduces the basic notions for using CalcuList: variables (integers, double, characters, Booleans, strings, lists, jsons, null), basic computations, manipulation of lists and jsons, types and casting. Variables may have a label.

Section 3 shows how to define and use a function *f*, which receives a number of arguments at the start and returns a computed value. An argument of *f* is either a variable or another function *g* that will be called inside the definition of *f*. Computation is done using recursion and a unique statement, *if-then-else,* implemented as a conditional operator “?”, similar to the one supported by Java. It is also possible to define functions with side effects, raised both by global setting commands, which update labeled global variables and/or function arguments, and by global printing commands, which print intermediate results.

In Section 4, we describe additional features of CalcuList, such as utilities, service commands and advanced printing facilities.

We illustrate various examples of advanced programming in CalcuList in Section 5, in particular, algorithms to handle lists, strings and jsons, sorting algorithms, algebric manipulation of vectors and matrices, finite state automata for language recognition, transducers, top-down parsers.

In Section 6, we describe some internal features of CalcuList. In particular, we enumerate and discuss all possible errors and exceptions that can be detected by CalcuList and present some insights of the CalcuList Virtual Machine (CLVM) that are necessary to illustrate show how the debugger works.

In Section 7 we draw the conclusion and discuss possible future evolutions of CalcuList.

The tutorial includes a number of appendices providing many insights on CLVM, formal definition of CalcuList and various implementation aspects.

2. Getting Started with CalcuList

2.1. Basic Computations and Variables

CalcuList may act as a simple calculator: one can type "^" followed by an expression, which is immediately computed and displayed. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Java or Python). In addition, the operator "//" performs integer division (i.e. quotient without remainder) like in Python. Parentheses can be used for grouping. For example:

```
>>/* Examples of Section 2 */
>>^2+2; /* sum of two integers */
4
>>^(2+1/3)*(1-1/3); /* result with type double because of the operator "/" */
1.5555555555555558
>>^((2+1/3)*(1-1/3))@int; /* result is truncated to integer by the cast @int */
1
>>^(2+1//3)*(1-1//3); /* terms truncated to integer by the operator "//" */
2
>>^2+2 == 4; /* '==' equality operator returning a bool value */
true
>>^(2+2 == 4)@int; /* the boolean value is converted into 1 or 0*/
1
>>^'A'; /* character */
A
>>^'A'%"; /* %" is a printing option to print a quoted character */
'A'
>>^'A'@int; /* character casted to int (code of 'A') */
65
>>^'\n'; /* escape sequence \n to insert a newline */

>>^'\n'%"; /* because of %" an escape character is not interpreted */
'\n'
>>^'\n'@int; /* escape character casted to int (code of '\n') */
10
>>^'A'+2; /* add 2 to the code of 'A' thus obtaining an integer */
67
>>^('A'+2)@char; /* after adding 2 to the code of 'A', it casts the result to
char to obtain 'C'*/
'C'
>>^'A'+'a'; /* it returns an integer */
162
>>^('A'+'a')@char; /* it returns the cent sign char */
¢
>>^"Hello "+"Worl"+'d'; /* concatenation of two strings and a char */
Hello World
>>^"Hello "+"Worl"+'d' %"; /* the string is quoted because of %" */
"Hello World"
>>^'H'+"ello "+"World"; /* an initial char cannot be concatenated to a string */
-------------^
** ERROR WRONG_EXP_TYPE** Type Mismatch : string - expected type: double or int
or char
-- skipping input until ';'
>>^'H'@string+"ello "+"World"; /* the initial char is casted to string */
Hello World
>>
```

The basic types for CalcuList are five: (1) *double* (i.e., real number in 64-bit double-precision floating-point format), (2) *int* (i.e., a 32-bit integer), (3) *char* (i.e., character represented in the UNICODE format), (4) *bool* (i.e., a Boolean with value *true* or *false*) and (5) *null* (that has a unique value named *null* as well).

The first three basic types are *numbers* and they can be used in arithmetic expressions. The terms of an operation are automatically casted to their most general numeric type: *double* is more general than *int* that, in turn, is more general than *char*. Moreover, the result of an arithmetic operation of two *char* values returns an *int* value, as it happens in Java. By issuing a query "`^`*expr*" on an expression *expr* with type *char*, the character is printed without any quote. To get the character quoted (actually, single quotes), we add the printing option `%"` (or `%*` that is a more general option).

The casting operator `@int,` next to an expression of type *t,* modifies the expression type into *int*. In particular: (1) it truncates the expression value to the next nearest integer towards 0 if *t* is *double*, (2) it replaces the character with its UNICODE code if *t* is *char,* or (3) it replaces *true* with 1 and *false* with 0 if *t* is *bool.* The operator "/" always returns a *double* while the operator "//" returns an *int* after having truncated the result, i.e., `x//y` corresponds to `(x/y)@int`.

The casting operator `@char` is applicable to an *int* term: if the term value is between 0 and 65533, it is replaced by the corresponding UNICODE character, otherwise it remains unmodified. The casting `@char` is applicable to a double term as well: to this end, it is first implicitely casted to *int*. The syntax for character constants is the same as for Java. The valid escape sequences are: \b (*backspace*), \t (*tab*), \n (*newline*), \f (*formfeed*), \r (*carriage return*), \" (*double quote*), \' (*single quote*), \\ (*backslash*).

CalcuList also supports three compound types: *string,* whose values are immutable sequences of characters, *list*, that are sequences of elements of any type, and *json* (*JavaScript Object Notation*), that is is a lightweight data-interchange format. Some details on the usage of strings are given next, whereas list and jsons will be treated in the next two sections.

By issuing a query "`^`*expr*" on an expression *expr* with type *string*, the string is printed without any quote. As for characters, the printing option `%"` or `%*` encloses the string between quotes – in this case quotes are double and not single. The elements of a string S are numbered with an index starting from zero. The *k*-th element of S is denoted by S[*k*-1] – so the first element is S[0], the second element is S[1] and so on. A string is concatenated with another string or with a character by means of the operator '+'. On the other hand, a character cannot be used as first term of a concatenation. A char term can be transformed into a one-character string by the casting operator `@string`.

CalcuList allows the user to define variables. An identifier name is a string of up to 64 symbols: letters, digits and underscore ("_"), starting with a letter[2]. CalcuList's identifier names are case sensitive and the reserved words are: *true*, *false, null, double, int, char, bool, string, list, json, type, lambda* and *halt*.

In a definition, the variable identifier must be followed by the assign symbol ("=") and by an expression, whose value and type is assigned to the variable. The format of an expression is:

- (*Numeric Expression*): A sequence of operations, whose operands are numeric (i.e, *double*, *int* or *char*) constants or numeric variables or calls of functions returning a numeric value (see next section about functions) and the operators are the arithmetic

[2] Note that, differently from Java, the symbol "$" is not allowed in an identifier name and the underscore cannot be the starting symbol of it.

ones: “+”. ”-“, “*”, “/”, “//” (division with integer truncation) – the latter three operators have higher priority w.r.t. the first two;

- (*Boolean Expression*): A sequence of operations, whose operands are Boolean constants or Boolean variables or calls of functions returning a Boolean value and the operators are the logical ones: “||” (*OR*), ”&&“ (*AND*), “!” (*NOT* applied to a single operand) with the precedence order that “!” comes before “&&” that in turn comes before “||”. Shortcuts are used in the evaluation of “||” and “&&”, i.e., the second operand of a logical expression is not evaluated if the expression is “A || B” and A is true (the value of the whole expression is then evaluated to true) or if the expression is “A && B” and A is false (the value of the whole expression is then evaluated to false);
- (*Comparison*): An expression EXPR1 COMP EXPR2, where EXPR1 and EXPR2 are two expression of compatible types and COMP is any comparison operator (“==”, “!=”, “>”, “>=”, “<”, “<=”), which returns the result of the comparison as a Boolean value – the comparison operator is applied after the evaluations of both expressions. The equality and inequality operators (“==” and “!=”) apply to operands of any type and they have lower priority w.r.t. “>”, “>=”, “<”, “<=”.

  The comparison EXPR1 == EXPR2 is equal to *true* if and only one of the following cases arise:

  1. both EXPR1 and EXPR2 have the same type and value,
  2. the two expressions are both numeric (i.e., *double* or *int* or *char*) and their values are equal or become equal after casting them to the most general numerical type (*double* is more general than *int* that, in turn, is more general than *char*) – e.g., `2==2.0` returns *true*.

  The inequality operator “!=” has the same characteristics as “==” except that it reverses the result – e.g., `2!=2.0` returns *false*.

  The other comparison operators (“>”, “>=”, “<”, “<=”) only apply to operands that either are both numeric or have the same type *string* or *bool* (types *list* and *null* are not supported) - the comparison refers to the lexicographic order for two strings and the order *false* < *true* for two Booleans.
- (*String concatenation*): An expression EXPR1 + EXPR2, where EXPR1 is of type *string* and EXPR2 is of type *string* or *char* implements the concatenation of the two string (after having casted the second expression to *string* if it is of type *char*). Concatenation has the same priority as the arithmetic sum.
- (*Char or Substring extraction*): Let an expression EXPR1 of type *string* be given, followed by [α], that is an operator with the highest priority – say that *s* is the string denoted by EXPR1 and the length of *s* is *n*:
  - α = “:” – then *s*[:] returns the whole string *s* (the string is not actually replicated as strings are immutable and CalcuList uses a string pool in order to avoid multiple representation of the same string) – it turns out that this notation is useless.
  - α = “EXPR2”, or “EXPR2:”, or “:EXPR2”, where EXPR2 is an expression of type *int* or *char* – let *i* be the value of EXPR2. Then:
    - *s*[*i*] returns the *i*-th character of *s* – if $i < 0$ or $i \geq n$, an error is reported;

- $s[i:]$ returns the substring of s starting from the i-th character up to its end – if $i < 0$, an error is reported while if $i \geq n$, the empty string is returned;
- $s[:i]$ returns the substring of s starting from the initial character up to $(n-1)$-th character of s – if $i < 0$, an error is reported while if $i \geq n$, the whole string is returned.

- α = "EXPR2:EXPR3", where EXPR1 and EXPR2 are two expressions of type *int* or *char* – let i and j be the value of EXPR2 and EXPR3, respectively. Then $s[i:j]$ returns a new string that is the substring of s that begins at the index i and ends at the index $j-1$ – if $i < 0$ or $j < 0$, an error is reported while if $i \geq j$, the empty string is returned

A sequence of operators with the same priority are evaluated from left to right and the precedence order can be modified by adding suitable parentheses.

Once defined, a variable is accessible and can be inquired and/or used in subsequent computations. The value assigned to a variable as well as its type can be modified by subsequent definitions.

```
>>x=2+.1;/* x is a double variable initialized to 2.1 */
>>x *= 2; /* compound assignment operator corresponding to x=x*2  */
>>y=(2e-2+1)*2; /* y is a double variable initialized to 2.04 */
>>z=x @int; /* the value of x is truncated and is assigned to z */
>>^z;
2
>>^z+y; /* result with type double */
4.04
>>st = z < y; /* st is a bool variable initialized to true because z < y */
>>^st;
true
>>st="CalcuList"; /* st is now a string variable */
>>^st;
CalcuList
>>!about st; /* details on the variable st are displayed */
(GV4)@[8] st: string = 63943 (->Heap)
"CalcuList"
>>^st[5]; /* returns the character at the index 5*/
L
>>^ans; /* returns the result of the last query*/
L
>>!about ans; /* details on the predefined variable ans are displayed */
(GV0)@[0] ans: char = 'L'
>>!variables; /* list of all variables used in the session */

------GLOBAL VARIABLES------
(GV0)@[0] ans: char = 'L'
(GV1)@[2] x: double = 4.2
(GV2)@[4] y: double = 2.04
(GV3)@[6] z: int = 2
(GV4)@[8] st: string = 63943 (->Heap)
---End of GOLBAL VARIABLES---

>>st1=st[:]; /* assigns the whole string st to st1 (the same as st1=st)*/
>>!about st1; /* details on the variable st1 are displayed */
(GV5)@[10] st: string = 63943 (->Heap)
"CalcuList"
>>^st[5:]; /* returns the substring from the index 5 up to the end */
List
>>^st[:5]; /* returns the substring from the beginning up to the index 4 */
Calcu
```

```
>>^st[2:5]; /* returns the substring from the index 2 up to the index 4 */
lcu
>>!about y; /* details on the variable y */
(GV3)@[4] y: double = 2.04
>>y=null; /* y is now a variable with type null and value null */
>>^y; /* null is printed as an empty string */

>>^y %"; /*  because of the print option %", null is printed */
null
>>!about y; /* updated contents of the variable y */
(GV2)@[4] y: null
>>
```

As shown in the above session, arithmetic expressions (including compound assignment operators such as "+=", "-=", etc.) mostly adopt the syntax of Java with some influence from Python. The main difference is for casting operators that are postfix in CalcuList (e.g., "EXPR @int") while they are prefix in Java (e.g., (*int*) EXPR). Manipulation of strings is done using the postfix operator [α]. Observe that `!about st` is a service command that displays a number of details about the *string* variable *st* such as:

- "(GV4)@[8]" indicates that *st* is the *fifth* global variable that has been defined in the session and it is therefore stored at the address 2×(5-1) = 8 of the memory (in the stack area) of CLVM – note that every variable occupies two memory words: one for its value e one for its type;
- 63943 is the value stored at the address 8, that is a link to the CLVM heap area where the actual content of the string is stored;
- "CalcuList" is the string stored at the address 63943.

The careful reader may have noticed that, indeed, we have only defined four variables (*x, y, z, st*) and not five. Actually, there is an additional pre-defined variable with name *ans*, that is located at the start of the stack and stores the result of the last query. In the above session, we type `^ans` after the query `^st[5]` so that we get the same answer. The subsequent service command `!about ans` replies that it is the first variable, located at the address 0 and storing the character 'L'. The list of all variables is displayed with the service command `!variables`.

Observe that the "about" command on the variable *s*displays that `st1` is the fifth variable (and then stored at the address 10) and its link to the content is the same as the one of `st.` This happens because CalcuList uses a string pool in order not to duplicate the storage of a string. This technique has been adopted as strings are considered immutable.

The "about" command on a scalar variable, such as *y* in the above session, displays information on its position in the stack (*y* is the third global variable and, therefore, stored at the address 4), its *type* (*y* is a double) and *value* (directly stored at the address 4).

2.2. List Manipulation

In addition to String, CalcuList supports a second compound data type: the *list*, that is a powerful type used by the language for constructing dynamic data structures and implementing recursive algorithms. A list consists of a number (possibly zero) comma-separated elements between square brackets. The elements can be of any type: *double, int, char, bool, string, null, list,* and *json* (see next session). It turns out that a list of lists can be recursively defined.

The elements of a list L are numbered with an index starting from zero. The *k*-th element of L is denoted by L[*k*-1] – so the first element (called the *head* of the list) is L[0] (also denoted simply by L[.]), the second element is L[1] and so on. An indexed element can be used as a term of an expression or it can be modified by means of a list element redefinition – in the

latter case, it is the left side of an assignment. As mentioned before, the first element can be also denoted by L[.] but this notation cannot be used when the element is the left side of an assignment. List assignment is a shallow operator as the list elements are shared by the two lists involved in the assignment. This implies that a change in some element of a list affects the second list as well. Such a situation cannot arise after defining a string variable S1 equals to another string variable S2 simply because a string is immutable. Thus, the only way to change S2 is to assign another string to it but S1 will continue to have the previous string value.

```
>>squares=[1, 4, 9, 16, 25]; /* squares is a list variable */
>>^squares; /* the element of the list are displayed */
[1, 4, 9, 16, 25]
>>squares[1]=squares[0]+squares[1];
>>squares[2]=squares[1]+squares[2];
>>squares[3]=squares[2]+squares[3];
>>squares[4]=squares[3]+squares[4];
>>^squares;
[1, 5, 14, 30, 55]
>>^squares[5]; /* the element with index 5 does not exist */
** ERROR LIST_OUT_BOUND** List Index Out of Range:
-- skipping input until ';'
>>^squares[-1]; /* negative indices are not allowed */
** ERROR NEGATIVE_LIST_INDEX** Negative list index:
-- skipping input until ';'
>>sq= squares; /* sq is a shallow copy of squares */
>>^sq;
[1, 5, 14, 30, 55]
>>squares[.]= 0.5; /* "[.]" cannot be used in the left side of an assignment */
----------^
** ERROR WRONG_TOKEN** Wrong Token: '.' – expected an expression
-- skipping input until ';'
>>squares[0]= 0.5;
>>^squares[.]; /* "[.]" is correct here */
0.5
>>?sq ; /* the update on squares[0] affects on sq as well */
[0.5, 5, 14, 30, 55]
>>
```

The elements of a list of type *char* or *string* are always printed into a quoted format, independently from whether the option `%"` is used or not. A list of chars can be transformed into a string by means of the casting operator `@string`. In turn, a string is transformed into a list of characters by issuing the casting operator `@list`.

```
>>charList=['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']; /* list of
chars, including space */
>>^charList; /* characters are always quoted inside a list */
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
>>^charList %>; /* list elements indented by the pretty print option "%>"  */
[    'H',
     'e',
     'l',
     'l',
     'o',
     ' ',
     'W',
     'o',
     'r',
     'l',
     'd'
]
>>^charList @string; /* the list is transformed into a string */
Hello World
>>^charList @string %"; /* the string is now printed with a quoted format */
```

```
"Hello World"
>>charList=["Hello", ' ', "World"]; /* charList now contains 2 strings and the
space char */
>>str = charList[0]+charList[1]+charList[2]; /* str is a string */
>>^str;
Hello World
>>^str %";
"Hello World"
>>^str @list; /* the string is transformed into a list of characters */
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
>>strangeList= ['H',2,3.5, [1,"World"], true, null]; /* a heterogeneous list */
>>^strangeList;
[ 'H', 2, 3.5, [ 1, "World" ], true, null ]
>>^strangeList[3][1]; /* element of a list that is an element of another list */
World
>> ^strangeList %*; /* print option %* */
[    'H',
     2,
     3.5,
     [    1,
          "World"
     ],
     true,
     null
]
>> ^strangeList %>; /* print option %> */
[    'H',
     2,
     3.5,
     [ 1, "World" ],
     true,
     null
]
>> !about strangeList; /* details on a list */
(GV10)@[20] strangeList: list = 63817(->Heap)
[    'H',
     2,
     3.5,
     [ 1, "World" ],
     true,
     null
]
>>
```

The print option `%*` for a list produces the indentation of all list elements and, in turn, of possible nested sub-elements; on the other hand, with the print option `%>`, indentation is limited only to the first level list elements. We point out that the print option `%*` has the same effect as `%"` when applied to a character, a string or a null value. On the other hand, the print option `%>` has no effect in such cases.

Concerning the result of the command `!about strangeList`, observe that it is the 10-*th* global variable stored at the address 20 (in the stack area) and the first element of this list is stored at the address 63817 (in the heap area). The contents of the list are indented in the same way as for a query with the pretty print option `%>`.

It is possible to construct a list by adding additional comma-separated elements on top to an existing list by using an append operator (the bar "|") followed by the list. For instance, given a list L, `M=[x,y|L]` extends L by adding two new elements, *x* and *y*, on top of L; L's content does not change but L now shares its elements with M. Thus, append is a shallow operator so that, for instance, if an element of L is changed, the corresponding element in M is changed as well. Recall that also the assignment of a list to another list is made in a shallow way.

It is possible to concatenate two lists by using the operator "+", for instance, given two lists L1 and L2, `L1+L2` add the elements of L2 at the end of L1. The operation is shallow as the elements of L2 are afterwards shared by L1.

```
>>L1=[0, 1, 2, 3, 4];
>>L2=[5, 6];
>>L3 = L1 + L2; /* concatenation of two lists */
>>^L3;
[0, 1, 2, 3, 4, 5, 6]
>>^L2;
[5, 6]
>>^L2;
[0, 1, 2, 3, 4, 5, 6]
>>^L1; /* L1 has been modified by the above concatenation */
[0, 1, 2, 3, 4, 5, 6]
>>L2[0] = -5;
>>^L3; /* the change on L2 affects L3 ... */
[0, 1, 2, 3, 4, -5, 6]
>>^L1; /* and L1 as well */
[0, 1, 2, 3, 4, -5, 6]
>>L=[-4 | L2 ];
>>^L;
[-4, -5, 6]
>>
```

Another shallow operator on a list L is L[>], which returns the (possibly empty) *tail* of L, i.e., the list starting from the second element L[1]. If L contains exactly one element then L[>] is the empty list "[]". On the other hand, if L is empty then the evaluation of L[>] will report an execution error. Given an index i, the operator L[>i] is equivalent to the (i+1) sequence of operators L[>][>]...[>] – so L[>] is equivalent to L[>0].

Deep list operators (i.e., cloning the elements) on a list L, say with n elements, are:

- L[:], a clone of the whole list L is constructed.
- L[i:], the sublist of L from the element L[i] to the end is cloned – if the index i is negative then an error is reported and if $i \geq n$ then the result is the empty list.
- L[i_1:i_2], the sublist of L from the element L[i_1] to the element L[i_2-1] is cloned – if any of the two indices is negative then an error is reported and if $i_1 \geq i_2$ or $i_1 \geq n$ then the result is the empty list.
- L[:i], the sublist of L from the first element L[0] to the element L[i-1] is cloned – if i is negative then an error is reported and if $i \geq n$ then the result is the empty list.

Note that the above operators are available in Python as well and are deep also in that language. There is a slight difference: negative indices are not allowed in CalcuList, whereas in Python they are used to numerate the elements starting from the end of the list.

```
>>L = [1,2,3,4,5];
>>L1 = L[>]; /* shallow operations: L1 shares the elements of L's tail */
>>?L1;
[2, 3, 4, 5]
>>L[3]=-4; /* the change in L is reflected in L1 as well */
>>^L1;
[2, 3, -4, 5]
>>^L1[>1]; /* tail of the tail of L1 (i.e., sublist from the 2^ element on) */
[-4, 5]
>>L2=L1[:]; /* deep assign: L2 is a clone of L1 */
>>L1[0]=-2; /* the change does not affect L2 */
>>^L1;
[-2, 3, -4, 5]
>>^L2; /* L2[0] is not changed */
[2, 3, -4, 5]
```

```
>>L2=L1[2:]; /* deep assign: L2 replicates the elements of L1 from the index 2
on */
>>L1[3]=-5; /* the change does not affect L2 */
>>^L2;
[-4, 5]
>>L2=L1[:2]; /* deep assign: L2 replicates the elements of L1 from the start up
to the index 1 */
>>L1[1]=-3; /* the change does not affect L2 */
>>^L2;
[-2, 3]
>>L2=L1[1:2]; /* deep assign: L2 replicates the element L1[1] */
>>L1[1] += 1; /* the update by the compound assignment does not affect L2  */
>>^L2;
[-3]
>>^L1;
[-2, -2, -4, -5];
>>
```

2.3. Json Manipulation

Json stands for *JavaScript Object Notation* and was originally intended to be a subset of the JavaScript scripting language and is commonly used with Javascript, but it is a language-independent data format.

In CalcuList[3] a json object is a (possibly empty) sequence of fields separated by comma and enclosed in curly braces. A field is a pair (*key*, *value*) separated by a colon: *key* is a string and *value* can be of any type: *double, int, char, bool, string, null, list* and (recursively) *json.* Two fields of a json object must not have the same key.

```
>>emp1={"firstName":"Mimmo", "lastName":"Sacca", "age": 30};
>>^emp1; /* display of a json */
{ "firstName": "Mimmo", "lastName": "Sacca", "age": 30 }
>>^emp1 %*; /* display of a json with pretty print (indentation) option */
{
     "firstName": "Mimmo",
     "lastName": "Sacca",
     "age": 30
}
>>^emp1["age"]; /* display of a field value for a given key */
30
>>^emp1["projects"]; /* null for a missing key, displayed as empty string */

>>^emp1["projects"] %"; /* null is now displayed */
null
>>^emp1@list; /* cast from json to list */
[ [ "firstName", "Mimmo" ], [ "lastName", "Sacca" ], [ "age", 30 ] ]
>>emp2L = [ ["firstName","Anna"], ["lastName","Rossi"], ["age", 32],
 [ "projects", ["p1", "p2"] ] ]; /* list of pairs (key,value) */
>>emp2= emp2L@json;/* cast from list to json */
>>^emp2 %*; /* nested indendation */
{
     "firstName": "Anna",
     "lastName": "Rossi",
     "age": 32,
     "projects": [
          "p1",
          "p2"
     ]
}
>>^emp2 %>; /* first-level only indendation */
```

[3] Calculist jsons have a simplified structure as they do not support some advanced features of general jsons.

```
{
     "firstName": "Anna",
     "lastName": "Rossi",
     "age": 32,
     "projects": [ "p1", "p2" ]
}
>>^emp2["projects"]; /* the value of the field "projects" is a list */
[ "p1", "p2" ]
>>^emp2["projects"][1]; /* second element of the list */
p2
>>
```

The variable *emp1* is defined as a json. When queried with the printing option `^emp2 %*`, the json fields are recursively indented. The selection of the value for the field with key "age" is done by issuing `emp2["age"]`; if the field is not defined for a key, the value *null* is returned. Observe that *null* is written as an empty string except when the printing option `%"` is used.

The printing option `%*` provides the indentation of the json elements as well as of their possible sub-elements. Indentation is limited to the first level json elements with the printing option `%>`.

In the above session portion, it is shown how the json *emp1* is transformed into a list of pairs (*key*, *value*) using the cast `@list`. On the other hand, a list of pairs (*key, value*) can be transformed into a json using the cast `@json`, as it is shown for the json *emp2*, which is constructed by means of the casting `emp2L@json`.

Next we present an example of a json containing other jsons as values.

```
>>dept = { "deptName": "unical", "emps": [ emp1, emp2[:], {"firstName":"Luisa",
"lastName":"Cosentino", "age": 28, "projects": ["p1", "p3"]} ] };
>>^dept %*;
{
     "deptName": "unical",
     "emps": [
          {
               "firstName": "Mimmo",
               "lastName": "Sacca",
               "age": 30
          },
          {
               "firstName": "Anna",
               "lastName": "Rossi",
               "age": 32,
               "projects": [
                    "p1",
                    "p2"
               ]
          },
          {
               "firstName": "Luisa",
               "lastName": "Cosentino",
               "age": 28,
               "projects": [
                    "p1",
                    "p3"
               ]
          }
     ]
}
>>^dept["emps"][2]["lastName"]; /* last name of the third employee */
Cosentino
>>emps = dept["emps"]; /* extract the list of employees */
>>^emps %*;
```

```
[    {
          "firstName": "Mimmo",
          "lastName": "Sacca",
          "age": 30
     },
     {
          "firstName": "Anna",
          "lastName": "Rossi",
          "age": 32,
          "projects": [
               "p1",
               "p2"
          ]
     },
     {
          "firstName": "Luisa",
          "lastName": "Cosentino",
          "age": 28,
          "projects": [
               "p1",
               "p3"
          ]
     }
]
>>^>>("dept_unical.dat") dept %*; /* output to a file */
- written output to file 'dept_unical.dat'
>>dept=<<("dept_unical.dat"); /* input from a file */
 - reading data from file 'dept_unical.dat'
 - data are correct
>>
```

The json *dept* has a field with key "emps" that contains a list of three jsons: the first two are *emp1* and *emp2* and the thirs one is created on the spot. Observe that the copy of *emp1* is shallow, thus any change in emp1 will be affect dept and conversely, whereas the copy of *emp2* is deep as the operator `[:]` makes a copy of it. The pretty print display clearly illustrated the structure of *dept*. Next, we define the variable *emps* as the list occurring as the value of the *dept* field with key "emps" – observe that variable and string are simply homonyms. Finally, in the last query, we redirect the output to the file 'dept_unical.dat', which is next read into *dept*. Observe that the new value of *dept* is different from the previous one: the first employee does not share anymore its contents with the variable *emp1*.

The read command "<<(*file-name*)" is useful to preserve values among sessions or to type them offline. A value stored in a file can be a scalar or a list or a json. Comments can be also included in the file and, obviously, they will be skipped during the reading.

2.4. Labeled Variables

We have shown that both the value and the type of a variable may be change without any restriction w.r.t. previous definitions. However, there are situations (e.g., while importing definitions elaborated in previous sessions) in which a user does not want to accidentally erase the value of a variable by introducing one with the same name. To mitigate the risk of creating unwished homonyms, CalcuList allows the user to assign a label to variables. For instance the command "`X: L1,L2,L`" declares three labeled variables `X.L1`, `X.L2` and `X.L3` – they are all initialized to *null*. After its definition, a labeled variable may have assigned a value by a suitable assignment statement. A label definition can be reissued to declare additional variables with that label and to reset previously-defined labeled variables to *null*.

```
>>X : L1, L2, L; /* 3 variables with label X initialized to null */
>>^X.L2 %";
null
```

```
>>X.L1=1;
>>X.L2=true;
>>X.L=L2; /* L2 has been previously defined as the list [-3] */
>>^X.L2;
true
>>!about X;
------VARIABLES WITH LABEL X-------
1: (GV20)@[40] L1: int = 1
2: (GV21)@[42] L2: bool = true
3: (GV22)@[44] L: list = 63730(->Heap)
---End of VARIABLES WITH LABEL X---

>>X : a, L1; /* X.a is initialized to null and X.L1 is reset to null */
>>X.a=X.L2;
>>!about X;
------VARIABLES WITH LABEL X-------
1: (GV20)@[40] L1: null
2: (GV21)@[42] L2: bool = true
3: (GV22)@[44] L: list = 63730 (->Heap)
4: (GV23)@[46] a: list = 63730 (->Heap)
---End of VARIABLES WITH LABEL X---

>>
```

As it will shown later in Section 3, labeled variables are important for defining functions with side effects.

2.5. Delete operation "#null"

An element *i* of a list L can be removed by issuing the command L[*i*]=#null. When preceded by "#", *null* stands for a delete operation.

```
>>L=[1,2,3];
>>L[1]=null; /* setting the element to null */
>>^L;
[ 1, null, 3 ]
>>L[1]=#null;
>>^L;
[ 1, 3 ]
>>L=#null; /* apparently no delete effect: L is set to null */
>>^L %*; /* the printing option to display the value null - the same as %" */
null
>>
```

When applied to the whole list L, the operation `L=#null` sets L to *null* in the same way as the command `L=null` does. However, `#null` also performs a physical deletion of the elements of L as their memory locations are added to the heap garbage list so that they can be later reused. Note that a physical deletion is also made when `#null` is assigned to a list element that is in turn a list.

The delete operation may be used also with jsons, as shown in the next session – note that the json *emp1* is the variable defined in Section 2.3.

```
>>^emp1;
{ "firstName": "Mimmo", "lastName": "Sacca", "age": 30 }
>>emp1["age"]=null; /* setting the field to null */
>>^emp1;
{ "firstName": "Mimmo", "lastName": "Sacca", "age": null }
>>emp1["age"]=#null; /* deleting the field */
>>^emp1;
{ "firstName": "Mimmo", "lastName": "Sacca" }
>>emp1=#null; /* apparently no delete effect: emp1 is set to null */
>>^emp1 %*;
```

```
null
>>
```

Also in this case, when applied to the whole json *emp1*, the operation "#null" does not delete *emp1* but set it to *null* in the same way as the command `emp1=null`. The effect of `#null` is to insert the memory locations storing the fields of *emp1* into the heap garbage list.

We point out that the effect of assigning `#null` to a scalar variable is exactly the same as assigning `null`.

2.6. The meta-type "type"

We have shown that a variable can be defined with one of the following types: *null, double, int, char, bool, string*, *list* or *json* that are all reserved words. By adding the casting operator `@type` next to an expression *exp*, the type of *exp* is returned. For instance, given the double variable x=2.5, the command `y=x@type` returns the value *double,* whose type is in turn the meta-type *type*.

CalcuList allows the user to also define a variable whose value is one of the above types whose type is *type*, which is to be therefore thought of as a meta-type. Two type expressions can be only compared for equality or inequality and no operations are supported for them, except the operator `@type`. The values with type *type* are: *null, double, int, char, bool, string*, *list, json* and the type of the value *type* is *meta-type,* that is however unavailable as a term of an expression, i.e., the command `type@type` is not supported.

Type variables are useful for enforcing type checking at run time. As an example, in the next session, given a tuple scheme *ts* (implemented as a list of types), a tuple value *tv* (i.e., a list with the same length of *ts*) can be checked on whether each element of it has the type indicated in the corresponding element of *ts*.

```
>>ts=[int, char, int, string]; /* tuple scheme */
>>tv = [2,'a',4,"bye"]; /* tuple value */
>>^tv[0]@type==ts[0] && tv[1]@type==ts[1] && tv[2]@type==ts[2] &&
               tv[3]@type==ts[3]; /* type check */
^ts[0]==ts[2]; /* equality and inequality supported for values of type 'type' */
true
>>^ts[0]==type; /* equality and inequality supported for values of type 'type'
and 'metatype' */
false
>>^ts[0]<ts[1]; /* precedence operators not supported for values of type 'type'
*/
** ERROR LT_NOT_SUPPORTED ** operator '<' not supported:  for types 'type' and
'type'
>>^tv[1]!=ts[1];/*  equality  and  inequality  not  supported  for  value  with
different types */
** ERROR EQ_NOT_SUPPORTED ** operator '==' not supported:  for types 'char' and
'type'
>>a=ts[0]@type; /* casting to type */
>>^a;
type
>>^a@type;/* casting of type to meta-type is not allowed */
** ERROR TOTYPE_NOT_SUPPORTED** cast to type not supported:  for metatype 'type'
>>/* End of Examples of Section 2 */
```

2.7. Summing-up on Casting and Operators Precedences

The types supported by CalcuList are 9: *double, int, char, bool, null, string, list, json, type*. A value can be transformed into another one by a cast operator "@*t*", where *t* is one of the types: *int, char, string, list, json, type.* The admitted casting for a value *v* are:

- *v*`@`*int*: *v* must be of type *double, int, char* or *bool.* The result *v*`@`*int* is of type *int* and is equal to: (1) the truncated value of *v* if *v* is a *double* (e.g., `2.5@int` returns 2), (2) *v* if *v* is an *int*, (3) the Unicode of *v* if *v* is a *char* (e.g., `'a'@int` returns 97), (4) 0 or 1 if *v* is a *bool* and is respectively *false* and *true*.
- *v*`@`*char*: *v* must be of type *int* or *double* with value between 0 and 65533 or *char.* The result *v*`@`*char* is of type *char* and is equal to: (1) the *char* whose Unicode is *v* if *v* is an *int* (e.g., 97`@`*int* returns 'a'), (2) the *char* whose Unicode is *v*`@`*int* if *v* is a double or (3) *v* if *v* is a *char*.
- *v*`@`*string*: *v* must be of type *char, list* of characters or *string.* The result *v*`@`*string* is of type *string* and is equal to: (1) the string with exactly the character *v* if *v* is a *char* (e.g., `'a'@string` returns "a"), (2) the concatenation of all characters in *v* if *v* is a list of all *char* elements (e.g., `['b','y','e']@string` returns "bye"), or (3) *v* if *v* is a *string*.
- *v*`@`*list*: *v* must be of type *string, json* or *list.* The result *v*`@`*list* is of type *list* and is equal to: (1) the list of all characters in v if *v* is a *string* (e.g., `"ciao"@list` returns ['c', 'i', 'a', 'o']), (2) the list of all pairs (key, value) in *v* if *v* is a json (e.g., `{"name": "john", "age": 30}@list` returns [["name", "john"], ["age", 30]]), or (3) *v* if *v* is a *list*.
- *v*`@`*json*: *v* must be of type *list or json.* The result *v*`@`*list* is of type *json* and is equal to: (1) the json whose fields are the pairs in v if *v* is a list of pairs (key, value) (e.g., `[["name", "john"],["age",30]]@json` returns {"name": "john", "age": 30}), or (3) *v* if *v* is a *json*.
- *v*`@`*type*: *v* must be of any type and the result *v*`@`*type* is of type *type* and is equal to: (1) *t* if *v* is of any type *t* except the meta-type *type* (e.g., `2.5@type` returns *double*, `2@type` returns *int*, `'a'@type` returns *char*, `"bye"@type` returns *string,* `true@type` returns *bool,* `[1,2,3]@type` returns *list,* `{"name":"john", "age":30}@type` returns *json* and *null*`@`*type* returns *null*), or (2) *type* if *v* is equal to *double, int, char, bool, null, string, list, json.* We recall that `type@type` causes an error. A final remark on the result of *null*`@`*type* is in order. The returned value *null* is a type (i.e., it has value *null* and type *type*) and is different from the source value *null* that has value *null* and type *null*. This possible ambiguity arises for *null* denotes both the value and the type – CalcuList prefers to risk some (indeed, very minor) ambiguity rather than inventing another name for the *null* type.

We point out that casting of an *int* to a *double* is not supported by CalcuList as it is instead done in other languages. This kind of casting is necessary in those languages for enforcing a division between two integers with a double result. But in CalcuList `a/b` returns a double also when both *a* and *b* are of type *int*; on the other hand, if an *int* result is wanted, it is sufficient to write `a//b` – note that an *int* result is returned also when the operands are of type *double*.

We conclude by summarizing the precedence and associativity of all CalcuList operators. As shown in the example, they are a small subset of Java operators and, in general, preserve the same precedence and associativity. The table below shows all CalcuList operators from highest to lowest precedence, along with their associativity (left to right or right to left or not applicable in some cases in the sense the possibility of writing a sequence of some operators is excluded):

Operator	Description	Level	Associativity
( )	invoke a function	1	N/A
[ ] @	access element of a list/json/string cast	2	left to right
+ −	unary plus unary minus	3	N/A

!	logical NOT		
* / // %	multiplicative	4	left to right
+ – +	additive string / list concatenation	5	left to right
< <= > >=	relational type comparison	6	left to right
== !=	equality inequality	7	left to right
&&	conditional AND	8	left to right
| |	conditional OR	9	left to right
? :	conditional	10	right to left
= += –= *= /= //=	assignment	11	N/A

3. Definition and Usage of Functions

3.1. Basic Functions

A function is defined by giving a name for it followed by: its comma-separated parameter names (included in parentheses), the colon symbol ":" and the expression that computes the value of the function. Parameters as well the return value of a function can be of any type and their actual types will be checked at run time. The expression can be either:

- a *simple expression* EXPR, with the format we have sketched while illustrating global variables – its value will be the one returned by the function, or
- a *conditional expression*[4], with the format COND ? EXPR1 : EXPR2, where COND is a Boolean expression, EXPR1 and EXPR2 are simple or conditional ecpressions – the two expressions are alternatively executed according to the value of COND (i.e., EXPR1 if COND holds true and EXPR2 otherwise).

No sided effects are allowed in the basic definitions of functions, i.e., functional operations do not modify current data and always create new ones. Therefore, global variables cannot be used inside the expression defining a function and, then, the terms used in the definition can only be parameters and constants as well as calls to functions (both the currently defined function, which may be recursively called, or previously defined ones). As function definitions are heavily based on recursion, the user must provide a termination condition for the recursion – this is up his/her responsibility. No forward declarations are supported but this limit can be easily overcome by first providing a dummy definition (i.e., by simply assigning the value *null*) and later refining it.

```
>> /* Examples of Section 3 */
>>fact(x) : x <= 1? 1: x*fact(x-1); /* the factorial recursive function */
>>^fact(5);
120
>>fib(x) : x <=1? x: fib(x-1)+fib(x-2); /* the Fibonacci recursive function */
>>^fib(10);
55
>>!clops;
35771
>>fibe1(x,f2,f1,k) : x==k? f1: fibe1(x,f1,f1+f2,k+1); /* function that will be
called next */
>>fibe(x) : x <= 1? x: fibe1(x,0,1,1); /* a more efficient Fibonacci function */
>>^fibe(10);
55
>>!clops;
3065
>>^fib(30); /* it may take a long while */
832040
>>!clops;
543892491
>>^fibe(30);
832040
>>!clops;
9005
>>
```

[4] Conditional expressions can be also used for writing a query or assigning a value to a global variable.

In the example above, we present two straightforward of the recursive functions: *fact* (factorial of a non-negative integer) and *fib* (computation of a Fibonacci number). The function *fibe* computes Fibonacci numbers more efficient than *fib* for it avoids that the same value is computed several times, thus getting exponential time execution. The linear time execution of *fibe* is achieved by the usage of a "tail" recursion, which is a sort of recursive implementation of iteration that works bottom-up (from the smaller Fibonacci number to the bigger ones so that any number is computed once) rather than top-down (from the bigger ones to the smaller ones with computation replications). To show that `^fibe(10)` is indeed more efficient than `^fib(10)`, we have issued the command `!clops` that returns the number of micro-operations (*clops*) executed by CalcuList's virtual machine: 3,065 clops against 35,771. If we now increase the value of *n* by a factor of 3 (passing from *n*=10 to *n*=30), *fibe* and *fib* will respectively execute 9,005 (i.e., with a linear increase of $\sim 2.97 \times 3{,}061$) and 543,892,491 (i.e., with an exponential increase $\sim 2^{13.9} \times 35{,}767$) clops!

Examples of simple non-recursive functions for manipulating characters are shown next:

```
 >>isLowerCase(c) : 'a' <= c && c <='z';/* the function returns a boolean */
>>toUpperCase(c) : isLowerCase(c)? ('A'+c-'a')@char: c;
>>?toUpperCase('b');
B
>>
```

Nine built-in functions, denoted by a name preceded by '_' (underscore), are available in CalcuList:

1. *_exp*(*x*), which computes the natural exponential function e^x, where *e* is Euler's number and *x* is a double,
2. *_log*(*x*), which computes the natural logarithm of a double *x*, and
3. *_pow*(*x, y*), which computes the value of the double *x* raised to the power of the double *y*e may define additional math functions based on them, e.g., squared root and a logarithm in any base.
4. *_len*(L), which returns the length of a string, list or a json L – the length of a json is the number of its fields,
5. *_tuple*(J), which returns the list of values for a json J,
6. *_isKey*(J,K), which returns *true* if the json J contains a field with K as a key or *false* otherwise;
7. *_rand*(), which returns a pseudo-random double,
8. *_ind*(S,T), which returns the index within the string S of the first occurrence of the string T or -1 if T does not occur in S, and
9. *_ind*(S,T,*i*), which returns the index within the string S of the first occurrence of the string T, starting at the index *i* or -1 if T does not occur in S, starting at *i*.

Note that the last two functions are overloaded, i.e., they have the same name.

```
>>^_exp(1); /* print Euler's number */
2.7182818284590455
>>^_log(3); /* print the natural logarithm of 3 */
1.0986122886681098
>>^_pow(2,3); /* print 2^3 */
8
>>Log(b,x): _log(x)/_log(b); /* logarithm of a double x in base b */
>>^Log(2,8); /* logarithm of 8 in base 2 */
3.0
>>sqrt(x): _pow(x,1/2); /* squared root of a double x */
>>?sqrt(16);
4.0
>>^_len("bye"); /* length of a string */
```

```
3
>>^_len([1,2,3]); /* length of a list */
3
>>emp : {"firstName":"Luisa", "lastName":"Cosentino", "age": 28, "projects":
["p1", "p3"]};
>>^_len(emp); /* length of a json (i.e., number of fields) */
4
>>^_tuple(emp); /* list of values for a json */
[ "Luisa", "Cosentino", 28, [ "p1", "p3" ] ]
>>^_isKey(emp,"projects"); /* emp contains a field with key "projects" */
true
>>^_isKey(emp,"name"); /* emp contains no field with key "name" */
false
>>^_rand(); /* generation of a ramdom double x, 0 <= x <= 1 */
0.7678770510445073
>>st = "01010101";
>>^_ind(st,"10"); /* returns the index within st of the first occurrence of "10"
*/
1
>>^_ind(st,"100"); /* returns -1 as "100" does not occur in s */
-1
>>^_ind(st,"10",4); /* returns the index within st of the first occurrence of
"10", starting at the index 4 */
5
>>^_ind(st,"10",6); /* returns -1 as "10" does not occur in st, starting at the
index 6 */
-1
>>
```

In the following we show two examples of functions using lists: (1) inverting a list (the function has a parameter of type list and returns a list) and (2) merging two ordered lists (the function has two parameters of type list and returns a list) .

```
>>L1= [1,3,5,7,9];
>>L2 = [0,2,4,6,8,10,12];
>>listReverse(L) : L==[]? []: listReverse(L[>])+[L[.]];
>>^listReverse(L1);
[9, 7, 5, 3, 1]
>>listMerge(O1,O2) : O1==[]? O2[:]: O2==[]? O1[:]: O1[.]<O2[.]?
          [O1[.] | listMerge(O1[>],O2)] : [O2[.] | listMerge(O1,O2[>])];
>>^listMerge(L1,L2);
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12]
>>^L1;
[1, 3, 5, 7, 9]
>>^L2;
[0, 2, 4, 6, 8, 10, 12]
>>
```

Let us now raise an important issue about possible side effects that could be induced by the usage of shallow list operations in a function. For instance, the function `listRev` contains the shallow operation `listRev(L[>])+[L[.]]`. However it is easy to see that the operation does not have any side effect on the parameter `L` since the list corresponding to `listRev(L[>])` is constructed from an empty list (at the last step of the recursion) that is later extended with elements copied from `L` at each previous step. Also the function `merge(O1,O2)` makes use of shallow operations. In this case, the exit conditions do not return an empty list as for `listRev` but residual portions of the two parameters `O1` and of `O2` that are, however, cloned to avoid possible side effects.

As we shall discuss in Section \ref{sec:conc}, such restrictions as well as the check that shallow list operations are used safely within a function definition will be introduced in a forthcoming version of CalcuList. At the moment, the check is left to the programmer. Note

that the values of L1 and L2 are not changed after calling the two functions on them – this because the two functions make a deep copy of them. In general, implementing a deep or shallow copy in a function is a programmer's responsibility. It follows that the usage of shallow copies may indeed introduce side effects into function operations – this is the price we decided to pay to enable a number of powerful operations on list.

We point out that "tail recursion" can be used to provide a more efficient implementation of *listReverse*, called *reverse.* To compare the two implementations, we construct a "long" list, say with 1000 elements, using the function *range*.

```
>>reverse1(L,R) : L==[]? R: reverse1(L[>],[L[.]|R]);
>>reverse(L) : reverse1(L,[]);
>>range(x1,x2) : x1>x2? []: [x1|range(x1+1,x2)];
>>L = range(1000);
>>^listReverse(L);
[1000, 999, 998, 997, 996, … ]
>> !clops;
2795726
>>^reverse(L);
[1000, 999, 998, 997, 996, … ]
>> !clops;
296308
>>
```

We use the service command `!clops`, which returns the number of micro-instructions (CalcuList micro OperationS) of CalcuList Virtual Machine that have been performed in the last execution -- `listRev(L)` is executed by 2,795,726 clops, whereas `^rev(L)` is executed by 296,308 clops, i.e., it is ten times faster. Tail recursion enables avoiding to scan the entire first term list in the concatenation operation -- in CalcuList an element of a list cannot be directly accessed by its index but only after scanning all the previous elements.

A careful reader may observe that the clops difference should be larger. In fact the number of steps required by `listRev` is around 500,000 for n=1000 so that the number of clops per step is around 5. On the other hand, the number of steps for `rev` is 1000, i.e., the number of clops per step is around 300, that is 6 times higher. The difference of clops per step is the result of two factors: (1) the implementation of the concatenation operator {\tt +} for lists is internally optimized by CalcuList during the search of the last element of the first list operand and (2) the implementation of tail recursion is not yet optimized by CalcuList, so that we do not avoid allocating a new stack frame for the tail-recursion.

We now present a rather elaborated example of manipulation of both lists and jsons. To this end, consider the json *dept* introduced in Section 2.3:

```
{
     "deptName": "unical",
     "emps": [
          {
               "firstName": "Mimmo",
               "lastName": "Sacca",
               "age": 30
          },
          {
               "firstName": "Anna",
               "lastName": "Rossi",
               "age": 32,
               "projects": [
                    "p1",
                    "p2"
               ]
```

```
        },
        {
            "firstName": "Luisa",
            "lastName": "Cosentino",
            "age": 28,
            "projects": [
                "p1",
                "p3"
            ]
        }
    ]
}
```

This json was written into the file in Section 2.3. We now shall read *dept* from the file to collect all the projects (without duplications) into a unique list, i.e., the result is the list with three elements: "p1", "p2" and "p3". To solve this problem, we first define three classical list manipulation functions: *member* (list membership), *addEl* (adding an element to a list if it is not in it) and *listUnion* (union of the elements of two lists without element duplication).

```
>>member(X,L) : L==[]? false: L[.]==X? true: member(X,L[>]); /* returns true if
X is an element X of the list L */
>>addEl(L,X) : member(X,L)? L: [X|L]; /* adds the element X to the list L if it
is not in it */
>>listUnion(L1,L2) : L2==[]? L1: listUnion(addEl(L1,L2[.]),L2[>]); /* union of
the elements of the lists L1 and L2 */
>>listAllProj(E) : E==[]? []: E[.]["projects"]==null? listAllProj(E[>]):
 listUnion(listAllProj(E[>]),E[.]["projects"]);
>>dept=<<("dept_unical.dat"); /* reading a json from a file */
>>emps=dept["emps"]; /* extract the list of employees */
>>^listAllProj(emps)/* list of all projects */
[ "p2", "p3", "p1" ]
```

Two functions may be defined in terms of each other (*mutual recursion*). Mutual recursion cannot be directly handled by CalcuList as it requires that any function must be defined before it can be used. Nevertheless, this problem can be easily solved by preliminary adding a "dummy" definition for one of the mutual recursive functions.

```
>>is_odd(x) : null; /* dummy definition */
>>is_even(x) : x < 0? is_even(-x): x==0? true: is_odd(x-1);
>>is_odd(x) : x<0? is_odd(-x): x==0? false: is_even(x-1);/* actual definition */
>>^is_odd(-3);
true
>>^is_even(-3);
false
[ "p2", "p3", "p1" ]
```

The functions parameters of a function that are not used in the definition can be simply indicated by "_", denoting an anonymous variable. For instance, the above dummy definition ca be rewritten as: `is_odd(_) : null;`.

3.2. Passing a Function as a Parameter of a Function

We have shown that a parameter of a function is a variable whose type will be given at the moment the actual parameter is passed. In CalcuList a parameter can also be a function, that is described using its name and its arity (i.e., the number of parameters) – the function cannot have, in turn, functions as parameters. In the example below, we define a function *filter* that has two parameters, a variable *L* and a filtering function *f*, and returns a Boolean. The prototype of the parameter function *f* is "f/1", thus any function passed as argument to *filter* must have exactly one (variable) parameter. Two examples of functions that can be passed as arguments are *d2or3* (that checks whether or not an integer is divisible by 2 or 3) and *d2and3*

(that checks whether or not an integer is divisible by both 2 and 3). As mentioned before, to avoid to get a too complicated formalism, in turn a function passed as a function argument must not have a function as parameter.

The function *filter* extracts the elements of a given list that satisfy the property defined by the function parameter *f.*

```
>>filter(L,f/1) : L==[]? []: f(L[.])? [L[.]|filter(L[>],f)]: filter(L[>],f);
>>d2or3(x) : x%2==0 || x%3==0;
>>d2and3(x) : x%2==0 && x%3==0;
>>^filter(range(1,20),d2or3);
[2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20]
>>^filter(range(1,20),d2and3);
[6, 12, 18]
>>
```

We next show other two classical examples of functions with parameter functions: *map* and *reduce*. The function *map* receives a list *L* and a function argument *f* mapping a double to another double and then it applies the mapping to all elements of *L* and returns the lists of the results. The function *reduce* receives a list *L*, a function argument *f* mapping two doubles into another double and an initializer value to be used for an empty list; then it applies *f* to the list head and the result of *reduce* for the list tail, in order to reduce the list to a single value by recursively applying *f* from right to left.

```
>>map(L,f/1) : L==[]? []: [f(L[.])|map(L[>],f)];
>>square(x) : x*x;
>>cube(x):x*x*x;
>>^map(range(1,10),square);
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>^map(range(1,10),cube);
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
>>reduce(L,f/2,init) : L==[]? init: f(L[.],reduce(L[>],f,init));
>>sum(x,y):x+y;
>>prod(x,y):x*y;
>>^reduce(range(1,10),sum,0);
55
>>^reduce(range(1,10),prod,1);
3628800
>>^reduce(map(filter(range(1,10),d2and3),cube),sum,0);
216
>>
```

The last query composes the three functions to compute the sum of the cubes of all integers in the range from 1 to 20 that are divisible by both 2 and 3.

Another interesting application of passing a function as argument is providing a version of *listMerge*, called *merge*, for which the actual ordering adopted by the two lists is passed as argument. The example below presents the generic function *checkOrder* for testing whether or not a list is sorted according to an ordering passed as argument.

```
>>merge(O1,O2,ord/2)[]: O1==[]? O2[:]: O2==[]? O1[:]: ord(O1[.],O2[])? [O1[.] |
merge(O1[>],O2,ord)] : [O2[.] | merge(O1,O2[>],ord)];
>>lte(x,y) : x<=y;
>>gte(x,y) : x>=y;
>>L1= [1,3,5,7,9,11,13,15,17,19];
>>L2= [2,4,6,8,10,16];
>>^merge(L1,L2,lte);
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 15, 16, 17, 19]
>>^merge(reverse(L1),reverse(L2),gte);
[19, 17, 16, 15, 13, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>checkOrder(L[],o/2) : L==[]||L[>]==[]? true: o(L[0],L[1]) &&
checkOrder(L[>],o);
```

```
>>^checkOrder(L1,lte);
true
>>^checkOrder(reverse(L1),lte);
false
>>^checkOrder(reverse(L1),gte);
true
>>
```

Suppose now that we are given a list of jsons and we want to filter the jsons that satisfy some condition in one of the fields. We next write the filter as a generic function *jsFilter* whose filtering condition is expressed by a function *filterCond* that receives the current json and the field (key and value) to be used for the evaluation. In the example, we refer to the variable *emps* previously defined, and we introduce two implementations for *filterCond*.

```
>>jsFilter(LJ,filterCond/3,K,V) : LJ==[]? []: filterCond(LJ[.],K,V)?
[LJ[.]|jsFilter(LJ[>],filterCond,K,V)]: jsFilter(LJ[>],filterCond,K,V); /*
filter function for a list of jsons */
>>select1KV(J,K,V) = J[K]!=null && J[K]==V; /* the value of the field K must be
equal to V */
>>^jsFilter(emps,select1KV,"age",28);
[ { "firstName": "Luisa", "lastName": "Cosentino", "age": 28,
"projects": [ "p1", "p3" ] } ]
>>select1KinV(J,K,V) : J[K]!=null && member(V,J[K]); /* the value of K must be a
list containing the element V */
>>^jsFilter(emps,select1KinV,"projects","p1") %";
[    {
          "firstName": "Anna",
          "lastName": "Rossi",
          "age": 32,
          "projects": [
               "p1",
               "p2"
          ]
     },
     {
          "firstName": "Luisa",
          "lastName": "Cosentino",
          "age": 28,
          "projects": [
               "p1",
               "p3"
          ]
     }
]
>>
```

More elaborated implementations of *filterCond* can be done by storing in K and in V lists of keys and of values, respectively, corresponding to a conjunction or disjunction of conditions.

3.3. Functions with Side Effects

To extend its expressive power, CalcuList also supports the definition of a function with side effects, that is: (1) labeled global variables can be used inside the expression defining the function, in addition to parameters and function calls and (2) both labeled global variables and parameters can be updated inside a function by means of the so-called *global setting commands* (*GSC*), that are included between curly braces `{!  !}`. A GSC is an assignment of an expression value to a variable. Global setting commands can be included in the definition of a function both at the start and at the end of the expression in the right-hand side of the definition, e.g.:

$$\langle f\text{-}lhs \rangle = \{!\ GSC_1\}\ \text{EXPR}\ \{!\ GSC_2\}$$

where <*f-lhs*> is the left-hand side of the function definition, GSC_1 is executed before the function starts its execution and GSC_2 at the end of its execution. Note that GSC_1 and GSC_2 may indeed consist of a sequence of commands, each of then surrounded by curly braces. In case of conditional expression, the global setting commands may be also inserted before and after each sub-expression, e.g.,

<*f-lhs*> = {GSC_1} COND? {GSC_2} EXPR1 {GSC_3}: {GSC_4} EXPR2 {GSC_5}

In this case, GSC_1 is executed before the function starts its execution and, on the basis of the result of COND, either (1) GSC_2 and GSC_3 are executed before and after the execution of EXPR1 or (2) GSC_4 and GSC_5 are executed before and after the execution of EXPR2. If EXPR1 (or EXPR2) is in turn a conditional expression, additional global setting commands can be introduced using the same schema as above.

A function is declared to be with side effects by adding `*` next to its name. Note that the property of a function to be with or without side effects cannot be changed after its first definition. To be used inside a function definition, a global variable must be declared as a labeled variable and its label (say X), followed by `*`, must be included between brackets as `<X*>` at the beginning of the function definition. The labeled variables are then addressed inside the function without writing their label.

A simple example of a function with side effects follows:

```
>>MATH: zeroDivide; /* label MATH for to the variable zeroDivide*/
>>div*(x,y) : <MATH*> {! zeroDivide=false !} y==0? 0 {! zeroDivide=true !}: x/y;
>>^div(3,0);
0
>>^MATH.zeroDivide;
true
>>^div(2,3);
0.6666666666666666
>>^MATH.zeroDivide;
false
>>^div(3,0) %+ MATH.zeroDivide? " error: zero divide!": "";
0.0 error: zero divide!
>>^div(2,3) %+ MATH.zeroDivide? " error: zero divide!": "";
0.6666666666666666
<<
```

Note that the operator `%+` concatenates two queries: the first one asks for the result of a division, whereas the second one is a conditional expression that reports the error if *zeroDivide* is *true* or the empty string otherwise.

A sophisticated user may dislike to get the default result 0.0 when a division by zero arises. CalcuList allows her/him to define a function that avoids to print the default result using the advanced printing facilities illustrated in Section 4.3. Another solution to avoid such inconvenient is to launch an exception through the command *exc*("*name*") when the division by zero arises. The effect of throwing an exception is to immediately stop the computation and to report the name of the exception, that is a string, and the name of the function throwing it.

```
>>div_exc(x,y) : y==0? exc("zeroDivide"): x/y;
>>^div_exc(3,0);
**Computation stopped by exception "zeroDivide" thrown by div_exc
>>^div_exc(2,3);
0.6666666666666666
<<
```

CalcuList does not provide mechanisms for handling exceptions that, therefore, determine a brusque interruption of a computation. If the computation is wished to be continued after detecting an exceptional event, labeled variables may be suitably used to handle such an event.

As a more complex example, we next show how to recognize a string representing a (possibly signed) integer and to compute its integer value if the string is syntactically correct. The function *intRec* receives a string and outputs true if the string represents a possibly signed integer. In addition, the function stores the integer value of the string into the global variable *val* with label INT. In the example, we use the built-in function *_len* to compute the length of a string.

```
>>digitVal(x) : x-'0';
>>isDigit(x) : '0'<=x && x<='9';
>>INT: val, base10val; /* two labeled global variables */
>>intRec1*(S,l) : null; /* dummy definition */
>>intRec*(S) : intRec1(S,_len(S));
/* natRec recognizes an unsigned integer and possibly computes its value */
>>natRec*(S,l,i):<INT*> i>=l||!isDigit(S[i])?false: /* empty or wrong digit */
             i==l-1? true /* case of a single digit integer */
                    {! val=digitVal(S[i]) !} {! base10val=1 !}:
                    natRec(S,l,i+1) /* general case by recursion */
                    {! base10val*=10 !} {! val+=base10val*digitVal(S[i]) !};
/* actual definition of intRec1 */
>>intRec1*(S,l) : <INT*> l==0? false: /* empty string is wrong */
                  S[0]=='-'? natRec(S,l,1) {! val=-val !}: /*sign is changed */
                  S[0]=='+'? natRec(S,l,1): /* val remains unchanged */
                  natRec(S,l,0); /* recognition of unsigned integer */
>>^intRec("");
false
>>^intRec("+");
false
>>^intRec("+2384");
true
>>^INT.val;
2384
>>^intRec("2384");
true
>>^INT.val;
2384
>>^intRec("-2384");
true
>>^INT.val;
-2384
>>
```

We point out that, in order to compute the length of the string S only once, the function *intRec* calls *intRec1* passing the length *l*, instead of directly performing the involved commands – this can be avoided by using local variables as illustrated in the next session. Furthermore, observe that the order of the two conditions into the disjunction `i>=l||!isDigit(S[i])` is important as it avoids to get an error in case the index *i* is out of the range of the string S and, therefore, S[*i*] is undefined. Shortcuts are adopted while evaluating conditions in disjunctions: therefore, when the condition `i>=l` is satisfied, the evaluation of `!isDigit(S[i])` is skipped and, therefore, no error is detected. Shortcuts are adopted also to implement conjunctions: in this case, if the first term is evaluated to false then the next terms are skipped.

An alternative (and simpler) implementation of the recognition function is to return the integer value and to set the correctness of the string into a global labeled Boolean variable *err*.

```
>>ParseInt: err;
>>parseNat*(S,v,l,i) : <ParseInt*> i>=l ||!isDigit(S[i])? 0 {! err = true !}:
                  i==l-1? v*10+ digitVal(S[i]):
```

```
                  parseNat(S,v*10+ digitVal(S[i]),l,i+1);
>>parseInt1*(S,l) : <ParseInt*> {! err = false !} l==0? 0 {! err = true !}:
                    S[0]=='-'? -parseNat(S, 0,l,1):
                    S[0]=='+'? parseNat(S, 0,l,1): parseNat(S, 0,l,0);
>>parseInt*(S) : parseInt1(S,_len(S));
>>^parseInt("") %+ ParseInt.err? " wrong integer string!": "";
0 wrong integer string
>>^parseInt("-") %+ ParseInt.err? " wrong integer string!": "";
0 wrong integer string
>>^parseInt("-2384")%i %+ ParseInt.err? " wrong integer string!": "";
-2384
>>^parseInt("+2384")%i %+ ParseInt.err? " wrong integer string!": "";
2384
>>^parseInt("2384")%i %+ ParseInt.err? " wrong integer string!": "";
2384
>>
```

Note that the default value 0 is printed when the string is wrong. A shown next, we can use exceptions to avoid such a situation – another solution is to use the advanced printing facilities shown in Section 4.3.

```
>>parseNat_exc(S,v,l,i) : i>=l || !isDigit(S[i])? exc("Wrong Integer String") :
                    i==l-1? v*10+digitVal(S[i]):
                    parseNat_exc(S, v*10+digitVal(S[i]),l,i+1);
>>parseInt1_exc(S,l) : l==0? exc("Empty String") :
                  S[0]=='-'? -parseNat_exc(S, 0,l,1):
                  S[0]=='+'? parseNat_exc(S, 0,l,1): parseNat_exc(S, 0,l,0);
>>parseInt_exc(S) : parseInt1_exc(S,_len(S));
>>^parseInt_exc("");
**Computation stopped by exception "Empty String" thrown by parseInt1_exc
>>^parseInt_exc("-");
**Computation stopped by exception "Wrong Integer String" thrown by parseNat_exc
>>^parseInt_exc("-2384");
-2384
>>
```

Notice that the parsing functions with exceptions are without side effects. Another way to avoid side effects is to write a parsing function returning a pair: the integer parsing value and a Boolean value stating whether the string has been recognized or not – note that this time, a value *true* means that no error was detected. The new version of *parseInt* is described next.

```
>>parseNatA(S, v,l,i) :  i>=l || !isDigit(S[i])? [0, false] :
                    i==l-1? [v*10+digitVal(S[i]),true]:
                    parseNatA(S, v*10+digitVal(S[i]),l,i+1);
>>chs(L) : [-L[0],L[1]];
>>parseIntA1(S,l) : l==0? [0, false] :
                  S[0]=='-'? chs(parseNatA(S, 0,l,1)):
                  S[0]=='+'? parseNatA(S, 0,l,1): parseNatA(S, 0,l,0);
>>parseIntA(S) = parseIntA1(S,_len(S));
>>^parseIntA("");
[ 0, false ]
>>^parseIntA("-");
[ 0, false ]
>>^parseIntA("-2384");
[ -2384, true ]
>>
```

As a last example, we show how to modify the list of employees *emps*, defined in the previous section. We want to add the new project "p0" to the list of projects of every employee in *emps*.

```
>>^emps %*; /* remind of emps content  */
[    {
          "firstName": "Mimmo",
          "lastName": "Sacca",
```

```
            "age": 30
      },
      {
            "firstName": "Anna",
            "lastName": "Rossi",
            "age": 32,
            "projects": [
                  "p1",
                  "p2"
            ]
      },
      {
            "firstName": "Luisa",
            "lastName": "Cosentino",
            "age": 28,
            "projects": [
                  "p1",
                  "p3"
            ]
      }
]
>>addProj1Emp*(E,p) : E["projects"]==null? {! E["projects"]=[p] !} true:
 {! E["projects"]+=[p] !} true; /* add a project to the list of projects of an
employee E */
>>addProjEmps*(EL,p) : EL==[]? true: /* update all employees of a list EL */
                              {! &addProj1Emp(EL[.],p) !} addProjEmps(EL[>],p);
>>^addProjEmps(emps,"p0");
true
>>^emps %*;
[     {
            "firstName": "Mimmo",
            "lastName": "Sacca",
            "age": 30,
            "projects": [
                  "p0"
            ]
      },
      {
            "firstName": "Anna",
            "lastName": "Rossi",
            "age": 32,
            "projects": [
                  "p1",
                  "p2",
                  "p0"
            ]
      },
      {
            "firstName": "Luisa",
            "lastName": "Cosentino",
            "age": 28,
            "projects": [
                  "p1",
                  "p3",
                  "p0"
            ]
      }
]
>>
```

The function *add1Proj1Emp* adds a project *p* to the project list of the employee *E*. Actually, if *E* is not involved in any project (i.e., its field "projects" is *null*), a singleton project list containing the new project *p* is created. As this function updates its parameter, it must be declared with

side effects. It follows that also the function *addProjEmps*, scanning all the employees of a list and, for each of them, invoking the former function to update the list of projects, is with side effects.

3.4. Using Local Variables inside Functions

Inspired by the imperative paradigm, CalcuList enables the usage of local variables that in some case may simplify the writing of a function. Local variables of a function must be declared, at the beginning of the expression defining the function, between brackets as for global variables. As an example, we define a function for computing the real roots of a second grade equation $a\ x^2 + b\ x + c = 0$. The function returns a list that is empty if the are no real roots or otherwise it contains the two roots (or a unique root for degenerate cases). Observe that the function does not have side effects.

```
>>roots2(a,b,c): <disc,rdisc,b2a> /* 3 local variables of type double */
    a==0? b==0? []: [-c/b]: /* degenerate case */
    {! disc=b*b-4*a*c !} /* discriminant */ disc<0? []: /* no real roots */
    disc==0? {! b2a=-b/(2*a) !} [b2a,b2a]: /* two coincident real roots */
    {! rdisc=sqrt(disc) !} [(-b-rdisc)/(2*a),(-b+rdisc)/(2*a)]; /*general case*/
>>^roots2(0,0,1);
[]
>>^roots2(0,1,2);
[-2.0]
>>^roots2(1,-4,4);
[2.0, 2.0]
>>^roots2(1,0,-4);
[-2.0, 2.0]
>>^roots2(2,4,-4);
[-2.732050807568877, 0.7320508075688772]
>>
```

Local variables can be obviously used also by functions with side effects. As an example, consider a function that swaps two elements *i* and *j* of a list *L*. The function always returns *true* after swapping the two elements.

```
>>swap*(L,i,j) : <tmp> {!tmp=L[i]!} {!L[i]=L[j]!} {!L[j]=tmp!} true;
>>K=[1,2,3,4];
>>^swap(K,1,3);
true
>>^K;
[1, 4, 3, 2]
>>^swap(K,2,5);
** ERROR LIST_OUT_BOUND ** List Index Out of Range:
>>
```

The function *swap* has side effects for it alters the content of its formal parameter L and, therefore, of the global variable K, which is passed as argument (actual parameter).

The function is next used to implement the bubble sort for a list in the ascending order. This algorithm executes *n*-1 passes, where *n* is the length of the list: each pass consists in scanning a portion of the list and performing the swap of two consecutive elements if the first of them follows the other in the ascending order. Each pass is implemented by the function *bs_pass*. The first pass starts from the element with index 0 to the element with index *last* = *n*-2 so that the element with index *n*-1 in the list is eventually the greatest one; the second pass starts from the element with index 0 to the element with index *last* = *n*-3, which becomes the second greater element and so on. The iteration on the *n*-1 passes is performed by the function *bs_iter*. Finally, the function *bubbleSort* activates the overall procedure, using the function `_len(L)`" to compute the length of the list L.

```
>>bs_pass*(L,i,last)  :  i>last?  null:  L[i]>L[i+1]?  {!  &swap(L,  i,  i+1)  !}
bs_pass(L, i+1, last): bs_pass(L, i+1, last);
>>bs_iter*(L,last) :  last<0? L: {! &bs_pass(L, 0, last) !} bs_iter(L, last-1);
>>bubbleSort*(L) : L==[]?true: bs_iter(L,_len(L)-2);
>>L=[3,1,7,4,0,5,13,11,-1];
>>^bubbleSort(L);
[-1, 0, 1, 3, 4, 5, 7, 11, 13]
>>!clops;
43085
>>
```

The function *bs_pass* can be made more efficient by extracting the tail of the list at each step of possible swap so that the first two elements in the list are compared.

```
>>bs_pass*(L[],i,last)&  :  i>last?  null:  L[0]>L[1]?  {!  &  swap(L,  0,  1)  !}
bs_pass(L[>], i+1, last): bs_pass(L[>], i+1, last));
>>^bubbleSort(L);
true
>>!clops;
32529
>>
```

The fact that the latter version of *bs_pass* is more efficient is evidenced by the results returned by the "!clops" command: the number of clops reduces from 43,085 down to 32,529.

The general scheme for declaring variables inside a function is: <L_1*, ..., *locVar*$_1$, ..., L_p*, ..., *locVar*$_k$> , i.e., labels and local variables are declared in any order. As an example, we present a version of the *roots2* function that reveals some insights when the solution is the empty list – there are three possible cases: no roots (when $a = b = 0$ and $c \neq 0$), indeterminate roots (when $a = b = c = 0$) and complex roots (when $a \neq 0$ and the discriminant is negative).

```
>>ROOTS2: /* noRoots==true -> no solutions,
          indRoots==true -> indeterminate solutions,
          complexRoots -> complex roots */
          noRoots, indRoots, complRoots;
>>roots2nice*(a,b,c): <ROOTS2*,disc,rdisc,b2a>
       {! noRoots=false !}  {! indRoots=false !}  {! complRoots=false !}
       a==0? b==0? c==0? {! indRoots=true !} [] : {! noRoots=true !} []: [-c/b]:
       {! disc=b*b-4*a*c ! disc<0? {! complRoots=true !} []: disc==0?
       {! b2a=-b/(2*a) !} [b2a,b2a]:
       {! rdisc=sqrt(disc) !} [(-b-rdisc)/(2*a),(-b+rdisc)/(2*a)];
>>^roots2nice(0,0,1);
[]
>>!about ROOTS2;
------VARIABLES WITH LABEL ROOTS2-------
/* noRoots==true -> no solutions, indRoots==true -> indeterminate solutions,
complexRoots -> complex roots */
1: (GV13)@[26] noRoots: bool = true
2: (GV14)@[28] indRoots: bool = false
3: (GV15)@[30] complRoots: bool = false
---End of VARIABLES WITH LABEL ROOTS2---
>>?roots2nice(1,0,4);
[]
>>!about ROOTS2;
------VARIABLES WITH LABEL ROOTS2-------
/* noRoots==true -> no solutions, indRoots==true -> indeterminate solutions,
complexRoots -> complex roots */
1: (GV13)@[26] noRoots: bool = false
2: (GV14)@[28] indRoots: bool = false
3: (GV15)@[30] complRoots: bool = true
---End of VARIABLES WITH LABEL ROOTS2---
>>!labels;
------LABELS-------
```

```
(L1) MATH
(L2) INT
(L3) ParseInt
(L4) ROOTS2
---End of LABELS---
>>
```

As it will be explained in detail in the next section, the service command "!about ROOTS2" and "!label" print respectively the values of all the variables with label "ROOTS2" and all the labels. In addition, possible comment next to the colon will be printed as well.

As last example of the section, we define the function *listDiv* that, given a number *x* and a list L, returns the list of the divisions of *x* by every non-zero element of L using the function *div*, previously defined. The function *listDiv* uses two labels (the label MATH used by *div* and the new label MATH1) and the local variable *d*. As the both MATH and MATH1 define a labeled variable with the same name (namely, *zeroDivide*), the whole description (i.e., inclusive of the appropriate label) is used whenever it is necessary to remove possible ambiguities.

```
>>MATH1: zeroDivide, numZD;
>>listDiv1*(x,L) : <MATH*, MATH1*,d> L==[]? []:
         {! d=div(x,L[]) !} MATH.zeroDivide?
                  {! MATH1.zeroDivide = true !} {!numZD+=1!} listDiv1(x,L[>]):
                  [d | listDiv1(x,L[>])];
>>listDiv*(x,L) = <MATH1*> {! zeroDivide=false !} {!numZD=0!} listDiv1(x,L);
>>^listDiv(2,[2,0,-4,0,1,0,5]);
[ 1.0, -0.5, 2.0, 0.4 ]
>>^MATH1.zeroDivide;
true
>>^MATH1.numZD;
3
>>
```

3.5. Lambda Functions

We have seen that a parameter of a function can be a function with a given arity (i.e., number of parameters). For instance, given the definition *f*(*x*, *g*/2) = <*expr*>, the second formal parameter *g* is a function with arity 2. We point out that any function *h* can be passed to *f* as actual parameter (argument) – for instance as *f*(3,*h*) – provided that *h* has been previously defined with the following properties: (*i*) the arity of *h* is 2, (*ii*) no parameter of *h* is a function and (*iii*) *h* has no side effects. It is also possible to define on-the-fly a function being passed as argument, using a suitable lambda notation.

In computer programming, a *lambda function* (or *anonymous function*) is a function definition that is not bound to an identifier. If a function is only used once, or a limited number of times, an anonymous function may be syntactically lighter than using a named function.

A lambda function is defined in Calculist as "lambda" <parameter-list> ":" <expr>, thus the whole definition is precided by the key-word "lambda" and is divided in two parts separated by a colon: the first part consists of the list of parameters for the lambda function and the second part is its defining expression.

A lambda function has neither side effects nor local variables, its parameters are only of type variable and no further lambda functions can be defined inside its defining expression.

Some examples of lambda function follow. They refer to the function *checkOrder, filter, map* and *reduce* that have been introduced in Section 3.2.

```
>> ^checkOrder(L1, lambda x,y: x < y ); /* function lambda(x,y)= x < y */
true
>> /* query with three lambda functions */
```

```
>> ^reduce(map(filter(range(1,20), lambda x: x%5==0), lambda x:2*x), lambda x,y:
x+y, 0);
100
>> /* a function containing lambda functions in its definition */
>> mapReduce5(L): reduce(map(filter(L, lambda x:x%5==0), lambda x:2*x), lambda
x,y:x+y,0);
>> ^mapReduce5(range(1,20));
100
>> f(g/2,x): x+2+g(x,2*x);
>> a(x,y): x+y;
>> ^f(lambda x,y: x+f(a,y),3);
34
>> ^f(lambda x,y: x+f(lambda z,w: z+w,y),3);
---------------------------^
** ERROR WRONG_LAMBDA ** Wrong Lambda Function Definition:  - a lambda function
cannot be a parameter of another lambda function
-- skipping input until ';'
>>/* End of Examples of Section 3 */
```

As pointed out at the end of the above session, a present CalcuList limitation is that a lambda function cannot defined inside another lambda function.

4. Additional Features of CalcuList

4.1. Utilities

At the start CalcuList imports possible pre-defined commands from the text file “Utilities.cl”, that is stored into the current working directory. So far in the examples, we have used an empty file;

```
$java –jar CalcuList_R_4_0_5.jar
     ***********************************************
                    ***  CalcuList   ***
     ***********************************************
     ******** Release 4.0.5 of August 7, 2017 ******

** Importing file "/<working_directory_full_path>/Utilities.cl" **
** Import ended **
-- 0 imported commands
>>
```

The name of the utility file is preceded by the full path of the working directory.

If the file “Utilities.cl” does not exists, an error is reported; but the session is kept open and the user may start using CalcuList:

```
$java –jar CalcuList_R_4_0_5.jar
     ***********************************************
                    ***  CalcuList   ***
     ***********************************************
     ******** Release 4.0.3 of August 7, 2017 ********

** Import of Utilities File failed
** ERROR WRONG_FILE ** Wrong File: Utilities.cl (No such file or directory)
You may continue the session without using utilities.
>>
```

The CalcuList’s user may want to provide its own utilities files. An example of a set of useful definitions is the following:

```
/***** UTILITIES *****/

/**** some maths ****/
min(x,y) : /* minimum of two comparable (number, bool, string) elements */
x < y ? x: y;

max(x,y) : /* maximum of two comparable (number, bool, string) elements */
x > y ? x: y;

Log(b,x) : /* logarithm of x with base b */
log(x)/log(b);

sqrt(x) : /* square root of a number */
_pow(x,1/2);

divisible(n,m) : /* true if n is divisible by m */
n%m==0;

GCD(n,m) : /* greatest common divisor of two integers */
n==m || m == 0? n: n<m? GCD(m,n): GCD(m,mod(n,m));

LCM(n,m) : /* least common multiple of two integers */
```

```
n*m/ GCD(n,m);

abs(x) : /* absolute value of a number x */
x < 0? -x: x;

/* integer floor of a double */
floor_1(x,n) :  /* called by floor(x), not to be used stand-alone */
x==n || x>0 ? n:  n-1;
floor(x) : /* integer floor of a double */
floor_1(x,x@int) ;

/* integer ceil of a double */
ceil_1(x,n) :  /* called by ceil(x), not to be used stand-alone */
x==n || x<0 ? n:  n+1;
ceil(x) : /* integer ceil of a double */
ceil_1(x,x@int);

round(x) : /* round of x to the closest integer */
abs(x-x@int) < 0.5? floor(x): ceil(x);

/**** operations on characters ****/
isSpace(c) : /* returns true if c is space */
c == ' ';

isWhiteSpace(x) : /* returns true if x is a white space */
x==' ' || x=='\t' || x=='\n' || x=='\f' || x=='\r';

digitVal(x) : /* returns the value (from 0 to 9) of a decimal char */
x-'0';

isDigit(x) : /* returns true if x is a decimal char */
'0'<=x && x<='9';

isLetter(x) : /* returns true if x is a letter */
'a' <= x && x <= 'z' || 'A' <= x && x <= 'Z';

isLetterOrDigit(x) : /* returns true if x is a letter or a decimal digit */
isDigit(x) || isLetter(x);

isUpperCase(x) : /* returns true if x is an upper case letter */
'A' <= x && x <= 'Z';

isLowerCase(x) : /* returns true if x is a lower case letter */
'a' <= x && x <= 'z';

toUpperCase(x) : /* change case to a lower case letter */
isLowerCase(x)? (x-'a'+'A')@char: x;

toLowerCase(x) : /* change case to an upper case letter */
isUpperCase(x)? (x-'A'+'a')@char: x;

/*** basic list operations ****/

range(x1,x2) : /* construct a list of all integers in the range <x1, x2> */
x1 > x2? []: [x1|range(x1+1,x2)];

randL(minv, maxv, k) : /* generate a list of k random doubles in the range <minv,
maxv> */
k<=0? []: [_rand()*(maxv-minv)+minv|randL(minv, maxv, k-1)];

randLI(minv, maxv, k) : /* generate a list of k random integers in the range
<minv, maxv> */
k<=0? []: [round(_rand()*(maxv-minv)+minv)|randLI(minv, maxv, k-1)];
```

```
/* list membership test */
member(X,L) : /* returns true if X is an element X of the list L */
L!=[] && ( L[.]==X || member(X,L[>]);

/* deep append of the list L2 to the end of a list L1, that does not modify both
lists  */
/*  shallow  append,  coalescing  the  two  lists,  can  be  simply  implemented  as
"L1+L2" */
/* other semi-deep operators are: */
/* (1) L1+L2[:], that only modifies L1 and (2) L1[:]+L2 that only modifies L2 */
append(L1,L2) : /* deep append of two lists L1 and L2 - more efficient than
L1[:]+L2[:] */
L1==[]? L2[:]: [L1[.]|append(L1[.],L2)];

equalL(L1,L2) : /* return true if two lists have the same elements */
L1==[]||L2==[]? L1==[]&&L2==[]: L1[.]==L2[.] && equalL(L1[>],L2[>]);

compareL(L1,L2) : /* comparison of two lists L1 and L2: it returns \n0 if L1 and
L2  have  the  same  elements,  \n-1  (resp.  1)  if  L1  lexicographically  precedes
(resp., follows) L2 */
L1==[] && L2==[]? 0: L1==[]? -1: L2==[]? 1: L1[.]<L2[.]? -1: L1[.]>L2[.]? 1:
compareL(L1[>],L2[>]);

/* maximum of a list */
maxL1(L,m) : /* called by maxL – not to be used stand-alone */
L[>]==[]? max(m,L[.]): maxL1(L[>],max(m,L[.]));
maxL(L) = /* it computes the maximum of a list */
L==[]? null: maxL1(L[>],L[.]);

/* minimum of a list */
minL1(L,m) : /* called by minL – not to be used stand-alone */
L[>]==[]? min(m,L[.]): minL1(L[>],min(m,L[.]));
minL(L) : /* it computes the minimum of a list */
L==[]? null: minL1(L[>],L[.]);

/* reverse of a list */
reverse_1(L, M) : /* called by reverse(L), not be used stand-alone */
L==[]? M: reverse_1(L[>],[L[.]|M]);
reverse(L) : /* reverse of a list computed by tail recursion */
reverse_1(L,[]);

/***** END OF UTILITIES *****/
```

By storing such definitions into “Utilities.cl”, a CalcuList session will set up as follows:

```
$java –jar CalcuList_R_4_0_5.jar
     ***********************************************
                    ***  CalcuList   ***
     ***********************************************
     ******** Release 4.0.5 of August 7, 2017   ******

** Importing file "/<working_directory_full_path>/Utilities.cl" **
** Import ended **
-- to list the 36 imported commands, type the service command "!history;"
>>
```

The list of the definitions, cleaned by all comments, is displayed with the service command “!history;”:

```
>>!history;

----FULL HISTORY------
1: min(x, y) : x<y? x: y;
2: max(x, y) : x>y? x: y;
3: Log(b, x) : _log(x)/_log(b);
```

```
4: sqrt(x) : _pow(x, 1/2);
. . .
35: reverse_1(L, M) : L==[]? M: reverse_1(L[>], [L[.]|M]);
36: reverse(L) : reverse_1(L, []);
---End of HISTORY---

>>
```

The service command "!functions;" only lists the function prototypes:

```
>>!functions;

------FUNCTIONS------
Built-in: _exp ( P1 )
Built-in: _ind ( P1 )
Built-in: _ind ( P1, P2 )
Built-in: _isKey ( P1, P2 )
Built-in: _len ( P1 )
Built-in: _len ( P1 )
Built-in: _log ( P1 )
Built-in: _pow ( P1, P2 )
Built-in: _rand ( )
Built-in: _tuple ( P1 )
(U1) min ( P1, P2 )
(U2) max ( P1, P2 )
(U3) Log ( P1, P2 )
(U4) sqrt ( P1 )
. . .
(U35) reverse_1 ( P1, P2 )
(U36) reverse ( P1 )
---End of FUNCTIONS---

>>
```

The 36 utility functions are numbered with indices preceded by "U" to remind they are indeed defined in "Utility.cl". The five built-in functions *_exp, _ind/1, _ind/2, _len, _log*, *_pow, _rand* and *_tuple* are listed at the beginning (but not numbered) as well.

The definition of a utility function cannot be modified:

```
>>reverse(X):null;
-----------^
** ERROR WRONG_FUNCT_DEF ** Wrong Function Definition: - a utility function
cannot be redefined
-- skipping input until ';'
>>
```

More details on a function can be obtained by issuing the service command "!about <funct-name>;":

```
>>! about reverse;
(U36) reverse(L) = reverse_1(L, []);
/* reverse of a list computed by tail recursion */
>>
```

Observe that any comment inserted next to ":" in a function definition (*Documentation Comment*) is kept by CalcuList to be displayed together with the definition of a function – see the definition of *reverse* in the file *Utility.cl*.

If the utility file contains errors, it is convenient to rename it (say with the name *Wrong_Utility.cl*), to restart the session using an empty utility file, to import *Wrong_Utility.cl* using the service command "`!import`" and to correct the wrong definitions.

4.2. Service Commands

There are a number of service commands, preceded by "!", to inquiry on the state of definitions during the current session:

1. !*variables*: list all global variables (including the labeled ones) and their values – only the heap address of the content for a string or a list or a json;
2. !*functions*: list all function names (including built-in and utility ones) and their arguments;
3. !*labels*: lists all labels;
4. !*about* ID, where ID is the name of a
   - global variable (including labeled ones): if ID is a list or a string, the whole content is displayed in addition to the content address, or
   - function (including built-in and utilities ones): the whole definition of ID is printed together with the possible documentation comment (i.e., a comment written next to "=" in the definition of ID) or
   - label: all the variables with label ID are listed;
5. !*history k*: all (correct) definitions and queries typed in during the session (*statements*) are stored into an ordered list with index from 1 to *n*, where *n* is the last command;
   - if $k > 0$, the command prints the statements from *k* to *n*;
   - if $k \leq 0$, the printed statements are those with index from *n*-*k* to *n* (therefore the last statement if $n = 0$);
   - if *k* is missing, all statements are listed;
6. !*exec k*: the statement with index *p* is executed, where $p = k$ if $k > 0$ or $p = n-k$ if $k \leq 0$; if *k* is missing, it is assumed that $k = 0$ (i.e., the last statement is executed);
7. !*import <echo-option>*: a number of statements (definitions and queries), stored into a file, can be imported into the session – the name of the file (with relative path w.r.t the current working directory or full path) is to be given after the prompt; *<echo-option>* can be *echo* (to print the imported commands on the screen) or *noecho* or can be missing (and, in this case, the option *echo* will be assumed);
8. !*save*: all (correct) statements (definitions and queries) that have been issued during the entire session are saved into a text file – the name of the file (with relative path w.r.t the current working directory or full path) is to be given after the prompt;
9. !*clops*: the number of clops (micro-instructions of CalcuList Machine) executed by the last query or variable definition is displayed;
10. !*release*: it displays some details about the CalcuList release that is currently used;
11. !*memory:* the contents of the current session internal memory: in particular, the *stack* that stores the global variables and the *heap* that stores dynamic structures likes strings, lists and jsons.
12. !*debug on/off:* debugging will be enabled or disabled depending on whether the option *on* or *off* is typed – if the option is missing then the default is "*on*" .

All the above commands can be abbreviated by typing any non-empty prefix of them.

A short CalcuList working session is shown next, where all service commands are issued except *debug* that will be shown later in Section 6.2. In this session, we use a utility file different from the one presented in Section 4.1. The new utility file includes the function *swap* for swapping two elements in a list (the same as the one defined in the previous section), a new function *numeric*, which verifies that the input argument is numeric, and the function *div*, which first checks whether the two operands are numeric or the second operand is different from zero and then computes the division of the two number:

```
/***** UTILITIES *****/
```

```
swap*(L, i, j) = /* swap elements i and j of the list L */
<tmp> {! tmp=L[i] !} {! L[i]=L[j] !} {! L[j]=tmp !} true;

numeric(x) : x@type==int || x@type==double || x@type==char;

MATH: /* label MATH for the Boolean variable err*/
err;

div*(x, y) : /* it computes the division of x by y and set MATH.err to true if y
is zero or x and/or y are not numeric */
<MATH*> {! err=false !} !numeric(x) || !numeric(y) || y==0?0 {! err=true} : x/y;

/***** END OF UTILITIES *****/
```

The beginning of the session follows:

```
     ************************************************
                      *** CalcuList ***
     ************************************************
     ******** Release 4.0.5 of August 7, 2017 *****
** Importing file "Utilities.cl"  **
** Import ended **
-- to list the 4 imported commands, type the service command "!history;"
>>!history;
----FULL HISTORY------
1: swap*(L, i, j) : <tmp> {! tmp=L[i] }  {! L[i]=L[j] }  {! L[j]=tmp } true;
2: numeric(x) : x @type==int || x @type==double || x @type==char;
3: MATH: err;
4: div*(x, y) : <MATH*> {! err=false } !numeric(x) || !numeric(y) || y==0? 0 {!
err=true } : x/y;
---End of HISTORY---
>>!functions;
------FUNCTIONS------
Built-in: _exp ( P1 )
Built-in: _ind ( P1 )
Built-in: _ind ( P1, P2 )
Built-in: _isKey ( P1, P2 )
Built-in: _len ( P1 )
Built-in: _log ( P1 )
Built-in: _pow ( P1, P2 )
Built-in: _rand ( )
Built-in: _tuple ( P1 )
(U1) swap* ( P1, P2, P3 )
(U2) numeric ( P1 )
(U3) div* ( P1, P2 )
---End of FUNCTIONS---
>>!about swap;
(U1) swap*(L, i, j) : <tmp> {! tmp=L[i] }  {! L[i]=L[j] }  {! L[j]=tmp } true;
/* swap elements i and j of the list L */
>>!about numeric;
(U2) numeric(x) : x @type==int || x @type==double || x @type==char;
>>!about MATH;
------VARIABLES WITH LABEL MATH-------
/* label MATH for the Boolean variable err*/
1: (GV1)@[2] err: null
---End of VARIABLES WITH LABEL MATH---
>>!about div;
(U3) div*(x, y)  <MATH*> {! err=false } !numeric(x) || !numeric(y) || y==0? 0 {!
err=true } : x/y;
/* it computes the division of x by y and set MATH.err to true if y is zero or x
and/or y are not numeric */
>>!release;
Release 4.0.5 of August 7, 2017
>>
```

Next we introduce a global variable *list*_L and define a function *listDiv* that, given a number *x* and a list L of elements, returns the list of all divisions of x by every element in L if the division of x by that element is feasible (according to the criteria of the function div) or '*' otherwise.

```
>>list_L= [ 2, 0, -4, 0, 20 ];
>>MATH1: numErr;
>> listDiv1*(x,L) : /* called by listDiv - not to be used stand-alone */
    <MATH*, MATH1*,d> L==[]? []: {!d=div(x,L[.])!} err?
    {!numErr=numErr+1!} ['*' | listDiv1(x,L[>])]: [d | listDiv1(x,L[>])];
>>listDiv*(x,L) : /* divide each element of L by x by putting '*' if the
division is not feasible */
<MATH1*> {!numErr=0!} listDiv1(x,L);
>>!about listDiv;
(F5) listDiv*(x, L) : <MATH1*> {! numErr=0 !} listDiv1(x, L);
/*divide each element of L by x by putting '*' if the division is not feasible*/
>>^listDiv(4,list_L);
[ 2.0, '*', -1.0, '*', 0.2 ]
>>!about MATH1;
------VARIABLES WITH LABEL MATH1-------
1: (GV3)@[6] numErr: int = 2
---End of VARIABLES WITH LABEL MATH1---
>>
```

Let us now define a version of bubble sort that is generic w.r.t. the order, that is passed as a function parameter. Given a list whose elements are pairs, name and age, this version will enable two different sorting: by name and by age, both in ascending order. The ordering by name is defined by the function *ascName* whereas the one by age is given my means of a lambda function.

```
>>studs = [ {"name": "john", "age": 25}, {"name": "bill", "age": 26},
{"name": "tom", "age": 23}, {"name": "marc", "age": 27} ];
>>bs_iter*(L, last, ord/2) : null; /* dummy definition */;
>> bubbleSort*(L,ord/2) : /* bubble sort of a list L according to the ordering
function ord */
L==[]? true: bs_iter(L, _len(L)-2,ord);
>>bs_pass*(L, i, last, ord/2) : /*called by bs_iter - not to be used stand-
alone*/
i>last? true: !ord(L[0],L[1])? swap(L, 0, 1) && bs_pass(L[>], i+1, last, ord):
bs_pass(L[>], i+1, last, ord);
>>bs_iter*(L, last,ord/2) : /*called by bs_iter - not to be used stand-alone*/
last<0? true: bs_pass(L, 0, last, ord) && bs_iter(L, last-1, ord);
>>!about bubbleSort;
(F7) bubbleSort*(L, ord/2) : L==[]? true: bs_iter(L, len(L)-2, ord);
/** sort of a list L according to the ordering function ord */
>>ascName(X,Y) : X["name"] <= Y["name"];
>>^bubbleSort(studs,ascName);
true
>>!clops;
9738
>> ^studs;
[ { "name": "bill", "age": 26 }, { "name": "john", "age": 25 },
{ "name": "marc", "age": 27 }, { "name": "tom", "age": 23 } ]
>>^bubbleSort(studs, lambda X,Y:X["age"] <= Y["age"] ); /* sorting by age */
true
>>!clops;
10996
>> ^studs;
[ { "name": "tom", "age": 23 }, { "name": "john", "age": 25 },
{ "name": "bill", "age": 26 }, { "name": "marc", "age": 27 } ]
>>
```

The above dummy definition `bs_iter*(L, last, (ord)/2) : null;` can be rewritten in the anonymous format: `bs_iter*(_, _, _/2) : null;`.

Observe that sorting *class* by names requires less clops than sorting by age – actually, 9680 instead of 10938. Note that in the example, the ordering on strings is rather efficient as it only reduces to check the first characters.

We continue the session by defining a function that, given a number *x* and a list M, partitions M into two sub-lists: the first one containing all elements of M that are less than *x* and the second one those that are greater than or equal to *x*. We use two labeled global variables, PART.L and PART.R, to store the two sub-lists.

```
>>list_numbers = [0, 3, 15, -2, 4, -7, -9, 15];
>>PART: L, R;
>> partition*(M,x) :
/* it partitions the elements of list M into two parts w.r.t the number x:
<left-sublist, M[0], right-sublist>\n
- the left sublist consists of all elements < x\n
- the right sublist consists of all elements >= x  */
<PART*> M==[]? true {!L=[]!} {!R=[]!}: M[]<x? partition(M[>],x) {!L=[M[]|L]!}:
partition(M[>],x) {! R=[M[]|R] !};
>>^partition(list_numbers,0);
true
>>^PART.L;
[ -2, -7, -9 ]
>>^PART.R;
[ 0, 3, 15, 4, 15 ]
>>!about partition;
(F10) partition*(M, x) : <PART*>M==[]? true {! L=[] }   {! R=[] !} : M[]<x?
partition(M[>], x) {! L=[M[]|L] !} : partition(M[>], x) {! R=[M[]|R] !} ;
/* partitions the elements of list M into two parts w.r.t the number x: <left-
sublist, M[0], right-sublist>
    - the left sublist consists of all elements < x
    - the right sublist consists of all elements >= x  */
>>
```

We continue the session by first defining a version of the function *partition* without side effects and then issuing a number of service commands. Note that the two sub-lists are returned as the two elements of a list and their elements are listed in the opposite order w.r.t. the first version.

```
>>part1(M,x,L,R) : /* called by partitionA - not to be used stand-alone */
M==[]? [L,R]: M[]<x? part1(M[>],x,[M[.]|L],R): part1(M[>],x,L,[M[.]|R]);
>> partitionA(M,x) :
/* partition function without side effects\n  the result is the list [L,R] of
two sub-lists, where\n - L contains all elements of M < x \n - R contains all
elements of M >= x */
part1(M,x,[],[]);
>>^partitionA(list_numbers,0);
[ [ -9, -7, -2 ], [ 15, 4, 15, 3, 0 ] ]
>>!about partitionA;
(F12) partitionA(M, x) : part1(M, x, [], []);
/* partition function without side effects
  the result is the list [L,R] of two sub-lists, where
 - L contains all elements of M < x
 - R contains all elements of M >= x */
>>!functions;
------FUNCTIONS------
Built-in: _exp ( P1 )
Built-in: _ind ( P1, P2 )
Built-in: _ind ( P1, P2, P3 )
Built-in: _len ( P1 )
Built-in: _log ( P1 )
Built-in: _pow ( P1, P2 )
Built-in: _rand ( )
Built-in: _tuple ( P1 )
```

```
(U1) swap* ( P0, P1, P2 )
(U2) numeric ( P0 )
(U3) div* ( P0, P1 )
(F4) listDiv1* ( P0, P1 )
(F5) listDiv* ( P0, P1 )
(F6) bs_iter* ( P0, P1, P3/2 )
(F7) bubbleSort* ( P0, P2/2 )
(F8) bs_pass* ( P0, P1, P2, P4/2 )
(F9) ascName ( P0, P1 )
(F10) partition* ( P0, P1 )
(F11) part1 ( P0, P1, P2, P3 )
(F12) partitionA ( P0, P1 )
---End of FUNCTIONS---
>>!variables;
------GLOBAL VARIABLES------
(GV0)@[0] ans: list = 63791 (->Heap)
(GV1)@[2] MATH.err: bool = false
(GV2)@[4] list_L: list = 63999 (->Heap)
(GV3)@[6] MATH1.numErr: int = 2
(GV4)@[8] studs: list = 63937 (->Heap)
(GV5)@[10] list_numbers: list = 63863 (->Heap)
(GV6)@[12] PART.L: list = 63827 (->Heap)
(GV7)@[14] PART.R: list = 63818 (->Heap)
---End of GOLBAL VARIABLES---
>>!labels;
------LABELS-------
(L1) MATH
(L2) MATH1
(L3) PART
---End of LABELS---
>>!history -10;
------HISTORY FROM 18 ON ------
18: ^ bubbleSort(studs,lambda X,Y:X["age"]<=Y["age"]);
19: ^ studs;
20: list_numbers = [0, 3, 15, -2, 4, -7, -9, 15];
21: PART: L, R;
22: partition*(M, x) : <PART*>M==[]? true {! L=[] !}  {! R=[] !} : M[.]<x?
partition(M[>],x) {! L=[M[.]|L] !} : partition(M[>],x) {! R=[M[.]|R] !} ;
23: ^ partition(list_numbers,0);
24: ^ PART.L;
25: ^ PART.R;
26: part1(M, x, L, R) : M==[]? [L, R]: M[.]<x? part1(M[>],x,[M[.]|L],R):
part1(M[>],x,L,[M[.]|R]);
27: partitionA(M, x) : part1(M,x,[],[]);
28: ^ partitionA(list_numbers,0);
---End of HISTORY---
>>!history 0;
------HISTORY FROM 28 ON ------
28: ^ partitionA(list_numbers, 0);
---End of HISTORY---
>>!exec;
^ partitionA(list_numbers, 0);
[ [ -9, -7, -2 ], [ 15, 4, 15, 3, 0 ] ]
>>!exec 25;
^ PART.R;
[ 0, 3, 15, 4, 15 ]
>>!save;
Enter file name (path w.r.t the current working directory or full path):
>>Section_4_Save.cl
** Session saved **
>>halt
Bye
```

We now start a new session and import all the saved commands of the previous session – recall that service commands were not saved. The situation is illustrated next.

```
     ***********************************************
                      *** CalcuList ***
     ***********************************************
     ******** Release 4.0.5 of August 7, 2017*****
** Importing file "Utilities.cl"  **
** Import ended **
-- to list the 4 imported commands, type the service command "!history;"
>>!import;
Enter file name (path w.r.t the current working directory or full path):
>> Section_4_Save.cl
** Importing file "<full-path>/Section_4_Save.cl"  **
/* CalcuList Session of Thu Aug 17 08:43:30 CET 2017 */
list_L = [2, 0, -4, 0, 20];
MATH1: numErr;
listDiv1*(x, L) : <MATH*,MATH1*,d>L==[]? []:   {! d=div(x,L[.]) !} err?   {!
numErr=numErr+1 !} ['*'|listDiv1(x,L[>])]: [d|listDiv1(x,L[>])];
listDiv*(x, L) = <MATH1*> {! numErr=0 !} listDiv1(x,L);
^ listDiv(4,list_L);
[ 2.0, '*', -1.0, '*', 0.2 ]
studs = [ { "name": "john", "age": 25 } ,  { "name": "bill", "age": 26 } ,
{ "name": "tom", "age": 23 } ,  { "name": "marc", "age": 27 } ];
bs_iter*(_, _, _/2) = null;
bubbleSort*(L, ord/2) = L==[]? true: bs_iter(L,_len(L)-2,ord);
bs_pass*(L, i, last, ord/2) : i>last? true: !ord(L[.],L[1])? swap(L,0,1) &&
bs_pass(L[>],i+1,last,ord): bs_pass(L[>],i+1,last,ord);
bs_iter*(L,   last,   ord/2)   :   last<0?   true:   bs_pass(L,0,last,ord)   &&
bs_iter(L,last-1,ord);
ascName(X, Y) : X["name"]<=Y["name"];
^ bubbleSort(studs,ascName);
true
^ studs;
[ { "name": "bill", "age": 26 }, { "name": "john", "age": 25 }, { "name":
"marc", "age": 27 }, { "name": "tom", "age": 23 } ]
^ bubbleSort(studs,lambda X,Y:X["age"]<=Y["age"]);
true
^ studs;
[ { "name": "tom", "age": 23 }, { "name": "john", "age": 25 }, { "name": "bill",
"age": 26 }, { "name": "marc", "age": 27 } ]
list_numbers = [0, 3, 15, -2, 4, -7, -9, 15];
PART: L, R;
partition*(M,  x)  =  <PART*>M==[]?  true  {!  L=[]  }    {!  R=[]  }  :  M[.]<x?
partition(M[>],x) {! L=[M[.]|L] } : partition(M[>],x) {! R=[M[.]|R] } ;
^ partition(list_numbers,0);
true
^ PART.L;
[ -2, -7, -9 ]
^ PART.R;
[ 0, 3, 15, 4, 15 ]
part1(M,  x,  L,  R)  :  M==[]?  [L,  R]:  M[.]<x?  part1(M[>],x,[M[.]|L],R):
part1(M[>],x,L,[M[.]|R]);
partitionA(M, x) = part1(M,x,[],[]);
^ partitionA(list_numbers,0);
[ [ -9, -7, -2 ], [ 15, 4, 15, 3, 0 ] ]
^ partitionA(list_numbers,0);
[ [ -9, -7, -2 ], [ 15, 4, 15, 3, 0 ] ]
^ PART.R;
[ 0, 3, 15, 4, 15 ]
** Import ended **
>>
```

CalcuList is now in the same state as at the end of the previous session. Note that the last two commands were issued in the previous session using the service command `!exec`.

We conclude by illustrating the service command `!memory`. To this end we restart the session, define six variables and then we issue the service command to print the memory state: first the seven static variables (the variable "ans" in addition to the six explicitly defined ones), which are stored in the stack, and then the dynamic variables (a list, a json and the strings), which are stored in the heap. Every static variable is represented as a pair: the value and the type. A list is represented as a sequence of elements, each of them represented as a triple: the value of the element, its type and the link to the next element of the list. A string with n characters is represented with n+2 contiguous memory cells: the value n followed by the n characters and the ling to the next string. We point out that strings are stored into a pool so that no duplicates may arise – this can be done as strings are immutable in CalcuList. A json is represented by a list of fields, pointing to the corresponding pairs (key, value). Each field is stored into a triplet: the pointer to the pair (key, value), the type "field" and the ponter to the next field of the same json. There is an additional initial field that is dummy and is introduced for technical reasons only. Each pair (key, value) is stored into a triplet: value, value type and pointer to the string describing the key.

The stack is stored upward from address 0 to 15 and the heap backward from address 63999 (the last cell in the memory) down to 63963. We add red-colored some annotations to the printout of CalcuList to provide some useful explanations.

```
     ************************************************
                      *** CalcuList ***
     ************************************************
     ********* Release 4.0.5 of August 7, 2017 *****
** Importing file "Utilities.cl"  **
** Import ended **
-- 0 imported commands
>>x=3;
>>y=x*1.5;
>>s="house";
>>c='h';
>>L=[x,y,x+y,x*y];
>>^L;
[ 3, 4.5, 7.5 ]
>>Q= { "type": s, "house": c };
>>^Q;
{ "type": "house", "house": 'h' }
>>!memory;
======================
++ STACK
--------------- [GV 0]: "ans"
00000: 63983      *->HEAP     pointer to the json in the heap(also pointed by Q)
00001: 8          json        type of the variable
--------------- [GV 1]: "x"
00002: 3                      x = 3
00003: 2          int         type of the variable
--------------- [GV 2]: "y"
00004: 4.5                    y = 4.5
00005: 1          double      type of the variable
--------------- [GV 3]: "s"
00006: 63999      *->HEAP     pointer to the string "house" in the heap
00007: 6          string      type of the variable
--------------- [GV 4]: "c"
00008: 104        'h'         c='h'
00009: 3          char        type of the variable
--------------- [GV 5]: "L"
00010: 63992      *->HEAP     pointer to the list in the heap
```

```
00011: 7          list        type of the variable
--------------- [GV 7]: "Q"
00012: 63983      *->HEAP     pointer to the json in the heap
00013: 8          json        type of the variable
---------------
======================
======================
++ HEAP (1^ Garbage Element *-> null, 1^ string *-> 63999)
--------------- [String 0]          first string
63999: 5          length
63998: 104        'h'
63997: 111        'o'
63996: 117        'u'
63995: 115        's'
63994: 101        'e'
63993: 63980      *-> next string   pointer to the second string in the pool
--------------- [List Element 0]    first element of the list L
63992: 3
63991: 2          int
63990: 63989      *->next list element
--------------- [List Element 1]    second element of the list L
63989: 4.5
63988: 1          double
63987: 63986      *->next list element
--------------- [List Element 2]    third and last element of the list L
63986: 7.5
63985: 1          double
63984: 0          end list
--------------- [Json Element 0]    dummy field 0 of the json Q
63983: 0          null
63982: 15         field
63981: 63971      *->next json element
--------------- [String 1]    second and last string
63980: 4          length
63979: 116        't'
63978: 121        'y'
63977: 112        'p'
63976: 101        'e'
63975: 0          * last string
--------------- [Key-Value 0]
63974: 63999      *->HEAP           pointer to the string “house”
63973: 6          string
63972: 63980      *->HEAP (key string)    pointer to the string “type”
--------------- [Json Element 1]    field 1 of the json Q
63971: 63974      *->HEAP           pointer to Key-Value 0
63970: 15         field
63969: 63965      *->next json element
--------------- [Key-Value 1]
63968: 104        'h'
63967: 3          char
63966: 63999      *->HEAP (key string)    pointer to the string “house”
--------------- [Json Element 2]    field 2 of the json Q
63965: 63968      *->HEAP           pointer to Key-Value 1
63964: 15         field
63963: 0          end json
---------------
======================
>>
```

Note the the variable `ans` is an alias for the result of the last query, that is the variable Q in this case. At the beginning of the heap are reported the pointer to the first garbage storage area and ti the first string in the pool. In the example there the garbage collection is empty as its elements are collected only after issuing a command using deep *null* (“#null”).

4.3. Advanced Printing Facilities

We have shown that a function with side effects may include one ore more *GSC*s (i.e., *Global Setting Commands* enclosed between curly braces with the option "!") in the prologue and in the epilogues of its defining expressions. A GPS typically consists of an assignment of a value to a global labeled variable or to a local variable. There is a second type of commands that can be inserted in a function definition between curly braces (with the option "^"): *GPC* (i.e., *Global Printing Command*) that is a query involving global labeled variables as well as the arguments of the function. GPC can be very useful for printing a list using pretty formatting features.

As a simple example, we next define a function that provides a pretty printing of the result of the function *div* defined in Section 3.3 – for reader's convenience, we rewrite the definitions in a new session using an empty utility file.

```
>>MATH: zeroDivide;
>>div*(x,y) : <MATH*> {! zeroDivide=false !} y==0? 0 {! zeroDivide=true !}: x/y;
>>^div(3,0) %+ MATH.zeroDivide? " error: zero divide!": "";
0.0 error: zero divide!
>>^div(2,3) %+ MATH.zeroDivide? " error: zero divide!": "";
0.6666666666666666
<<
```

To avoid to print the default value 0.0 when a division by zero arises, we define a printing function with GPC, which is with side effects for it consults a labeled global variable.

```
>>printDiv*(x) : <MATH*> {^ zeroDivide? "error: zero divide!": x ^} null;
>>^printDiv(div(3,0));
error: zero divide!
>>^printDiv(div(2,3));
0.6666666666666666
<<
```

Observe that, without the option `%"`, the value *null* is not printed.

Also for the second example we refer to some definitions introduced in Section 3.3, in particular the function *parseInt* for recognizing whether a strings represents a (possible signed) integer.

```
>>digitVal(x) : x-'0';
>>isDigit(x) : '0'<=x && x<='9';
>>ParseInt: err;
>>parseNat*(S,v,l,i) : <ParseInt*> i>=l ||!isDigit(S[i])? 0 {! err = true !}:
                  i==l-1? v*10+ digitVal(S[i]):
                  parseNat(S,v*10+ digitVal(S[i]),l,i+1);
>>parseInt1*(S,l) : <ParseInt*> {! err = false !} l==0? 0 {! err = true !}:
                    S[0]=='-'? -parseNat(S, 0,l,1):
                    S[0]=='+'? parseNat(S, 0,l,1): parseNat(S, 0,l,0);
>>parseInt*(S) : parseInt1(S,_len(S));
>>^parseInt("-") %+ ParseInt.err? " wrong integer string!": "";
0 wrong integer string
>>^parseInt("-2384") %+ ParseInt.err? " wrong integer string!": "";
-2384
>>
```

Note that the default value 0 is printed when the string is wrong. To avoid such a situation, we next define a suitable printing function.

```
>>printParseInt*(P) : <ParseInt*> {^ err? "wrong integer string!": P ^} null;
>>^printParseInt(parseInt("-"));
wrong integer string
>>^printParseInt(parseInt("-2384"));
```

```
-2384
>>
```

As a further example of GPC usage, consider the problem of extracting all the words of a given string S – a word is a substring of S separated by one or more white spaces. The function *extractW* returns the list of all words, but we want to have the words printed one for line. The function *printW* provides a solution by means of a GPC command.

```
>>isWhiteSpace(x) : x==' ' || x=='\t' || x=='\n' || x=='\f' || x=='\r';
>>ew(_,_,_,_) : null; /* dummy definition */
>>ew1(S,W,W1,l,i): l==i? W+[W1@string]: isWhiteSpace(S[i])? ew(S, W+[W1@string],
l,i+1): ew1(S,W,W1+[S[i]],l,i+1);
>>ew(S,W,l,i):l==i? W: isWhiteSpace(S[i])?ew(S,W,l,i+1): ew1(S,W,[S[i]],l, i+1);
>>extractW(S) : ew(S,[],_len(S),0);
>>^extractW(" All the world likes CalcuList  ");
[ "All", "the", "world", "likes", "CalcuList" ]
>>printW(W) : W==[]? null: {^ '\n' %+ W[] ^} printW(W[>]);
>>?^printW(extractW(" All the world likes CalcuList  "));

All
the
world
likes
CalcuList
>>
```

Observe that a word is singled out as a list of characters and it is converted to a string at the end of its construction. Note also that the escape character \n is used to skip a line. The function *printW* is assumed to be without side effects even though it modifies the output file. For CalcuList, side effects arise when a function accesses global variables or modifies some of its arguments.

Next we show how to write two words per line.

```
>>print2WA(_) : null; /* dummy definition */
>>print2WB(W) : W==[]? null: {^ '\t' %+ W[] ^} print2WA(W[>]);
>>print2WA(W) : W==[]? null: {^ '\n' %+ W[] ^} print2WB(W[>]);
>>print2WL(W) : print2WA(W);
>>^print2WL(extractW(" All the world likes CalcuList  "));

All         the
world       likes
CalcuList
>>
```

As an additional example, suppose we are given a list of pairs, each of them storing the name, the age and the city of a person. We want to suitably print each triple into a new line – the three fields are aligned by the tab escape character \t.

```
>>persons= [ { "name": "john", "age": 25, "city": "NY"},
            { "name": "mark", "age": 30, "city": "SF"},
            { "name": "mimmo", "age": 66, "city": "CS"},
            { "name": "mike", "age": 18, "city": "LA"} ];
>> printPersons(P) : P == []? null:
            {^'\n' %+ P[.]["name"] %+ "\taged " %+ P[.]["age"]
            %+ "\tlives in " %+ P[.]["city"] ^}
            printPersons(P[>]);
<<^printPersons(persons);

john  aged 25     lives in NY
mark  aged 30     lives in SF
mimmo aged 45     lives in CS
mike  aged 18     lives in LA
```

```
>>invPrintPersons(P) : P == []? null: invPrintPersons(P[>])
       {^'\n' %+ P[.]["name"] %+ "\taged " %+ P[.]["age"]
        %+ "\tlives in " %+ P[.]["city"] ^};
>> ^invPrintPersons(persons);

mike  aged 18     lives in LA
mimmo aged 45     lives in CS
mark  aged 30     lives in SF
john  aged 25     lives in NY
>>
```

The list *persons* contains 4 jsons. At each stage of the recursion, the function *printPersons* prints a triple plus some formatting adornments; then, it recursively calls itself by skipping the current triple by means of the operation P[>]. If we postpone the printing after the recursive call as in the function *invPrintPersons*, we obtain the list of triples printed in the inverse order.

As a last example, we define two functions for printing respectively an array and a matrix of numbers. We recall that by using `%>`, a list or a json is printed with a first level indentation only, whereas the option `%*` enables full indentation. Observe that the option `%*` has the same effect as `%"` when applied to string or characters, while `%>` does not determine any effect on them.

```
>>M=[[-1, 2, 3, 0], [2, 0, 0, 1 ], [-1, 0, 1,  -4], [2, 1, 1, 0] ];
>>^M[2]; /* default print of a vector (row of the matrix) */
[ -1, 0, 1, -4 ]
>>^M[2] %*; /* pretty printing option */
[    -1,
     0,
     1,
     -4
]
>>^M %>; /* pretty printing option with first-level indentation only */
[    [ -1, 2, 3, 0 ],
     [ 2, 0, 0, 1 ],
     [ -1, 0, 1, -4 ],
     [ 2, 1, 1, 0 ]
]
>> printV(V) : /* formatted print of a vector V */
V==[]? null: {^"\t" %+ V[] ^}  printV(V[>]);
<<^printV(M[2]);
      2     0     0     1
>>printM(M) : /* formatted print of a matrix M */
M==[]? null:{^"\n" %+ printV(M[]) ^} printM(M[>]);;
>> ^printM(M);

      -1    2     3     0
      2     0     0     1
      -1    0     1     -4
      2     1     1     0
>>/* End of Examples of Section 4 */
```

Observe that the printing option %' is rather effective for bi-dimensional matrices.

The above examples have shown that CalcuList allows the user to format the output in a rather simple way. Indeed, despite some effort in defining ad hoc printing functions, the formatting capabilities of CalcuList remain rather limited: at this stage, there are no ambitions to provide powerful display environments.

5. Advanced Programming in CalcuList

5.1. Playing with Lists

Let us first recall the basic properties of a list, that is a sequence of elements of any type (*int, double, char, string, list, json, type, null*) – the elements are numbered with an index from 0 on. Given a list L and an index *i*,

- L[*i*] denotes the element *i* of L;
- L[.] is similar to L[0] (*head* of the list) but it cannot be used as left hand side of an assignment;
- L[*i*:] is the clone of the sublist of L from the element *i* to the end;
- L[:*i*] is the clone of the sublist of L from the element 0 to the element *i*-1;
- L[*i*:*j*] is the clone of the sublist of L from the element *i* to the element *j*-1, where *j* is second index;
- L[:] is the clone of whole list L;
- [E_1,..., E_n |L] extends the list L by adding the elements E_1,..., E_n on top of it;
- L[>] returns the *tail* of L, i.e., the sublist of L from the element 1 (i.e., the second element) up to the end without cloning the elements (i.e., the sublist shares such elements with L).
- L[>*i*] returns the *i*-th *subtail* of L, i.e., the sublist of L from the element *i*+1 up to the end without cloning the elements – it is equivalent to the *i*+1 operator sequence L[>][>] ... [>] and L[>] is equivalent to L[>0].

Two lists can be concatenated by the operator + and be compared only with the operators == and `!=`. A list L of characters can be converted into a string by issuing the command: `L@string`. The length of a list is computed by the built-in function `_len`.

An element of a list L, say with index *i*, can be removed by setting `L[i]=#null`. In addition, if `L[i]` is in turn a list then all elements of such a list are physically removed by inserting them into the heap garbage list. If L is a list, the assignment `L=#null` both sets L to *null* and physically removes all elements of L.

A list is displayed by enclosing its elements between square parentheses and insert a comma between two contiguous elements. The print option `%*` indents the list elements as well theirs sub-elements, whereas the print option `%>` indents the list elements only.

We now define a number of functions implementing classical operations on lists: membership, equality, compare and reverse. The *compare* function applied to two lists L1 and L2 returns 0 if the two lists have the same elements, -1 if L1 precedes L2 in the lexicographically ascending order of their elements (e.g., [1, 2, 1] precedes [1, 2, 2, 3]) and 1 if L1 follows L2 in the lexicographically ascending order.

In the example below, the reverse and equal functions are used to check whether a list is a palindrome, i.e., it reads the same backward or forward.

```
>> member(X,L) : /* it returns true if X is an element of the list L */
L==[]? false: L[.]==X? true: member(X,L[>]);
>> equalL(L1,L2) : /* it returns true if two lists have the same elements */
L1==[]||L2==[]? L1==[]&&L2==[]: L1[.]==L2[.] && equalL(L1[>],L2[>]);
>> compareL(L1,L2) : /* comparison of two lists L1 and L2: it returns\n0 if L1
and L2 have the same elements \n-1 (resp. 1) if L1 lexicographically precedes
(resp., follows) L2 */
L1==[] && L2==[]? 0: L1==[]? -1: L2==[]? 1: L1[.]<L2[.]? -1: L1[.]>L2[.]? 1:
compareL(L1[>],L2[>]);
```

```
>> reverseA(L) : /* naive implementation */
L==[]? []: reverseA(L[>])+[L[.]];
/* more efficient implementation of reverse using tail recursion */
/* elements are added at the top instead of at the end */
>> reverse_1(L, M) : /* called by reverse(L), not be used stand-alone */
L==[]? M: reverse_1(L[>],[L[.]|M]);
>> reverse(L) : /* reverse of a list computed by tail recursion */
reverse_1(L,[]);
>> isPalindrome(L) : /* returns true if L reads the same backward or forward */
L==[] || L[>]==[]? true: equalL(L,reverse(L));
>>
```

We next show how to repeat the element of a list a fixed number of times.

```
>> rep(L,i) : /* returns the list obtained by concatenating the list L for i
times */
i<=0? []: L[:] + rep(L,i-1);
>> ^rep([0,1],5);
[ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1 ]
>>
```

Note that the list L is cloned any time it is concatenated because otherwise L would be altered by the append operator +. Some more remarks on `append` are now in order.

Recall that, given two lists `L1` and `L2`, the command `L=L1+L2` appends the elements of `L2` to those of `L1` . As the last element of `L1` is attached to the first of `L2`, the append has the side effect of modifying `L1`, which eventually coincides with `L`. A deep implementation that clones the involved elements is given by the command: `L=L1[:]+L2[:]`. This command scans the elements of L1 twice: the first time to clone L1 (say that the clone of L1 is L1c) and the second time to reach the last element of L1c. A more efficient implementation of the deep append that scans these elements only once is given by the function shown next.

```
>> append(L1,L2) : /* deep append of two lists L1 and L2 - more efficient than
L1[:]+L2[:] */
L1[]==[]? L2[:]: [L1[.]|append(L1[>],L2)];
>> ^rep([0,1],5);
[ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1 ]
>>
```

The (*clockwise*) *rotation* of the *n* elements of a list by *k* positions consists of (*i*) shifting the elements to the right by k positions and (*ii*) moving the last *k* elements to the top of the list. As an example, consider the list L = [5, 1, 4, 20, 15, 13]. The rotation of L by *k* positions is [15, 13, 5, 1, 4, 20] if *k* = 2 or [4, 20, 15, 13, 5, 1] if k=4. If $k \geq n$, k is replaced by $k \% n$, i.e., the remainder of *k* by *n*. If *k*=0 the list remains unchanged. If *k*<0, the rotation is *anti-clockwise* for it is made by (*i*) shifting the elements to the left by k positions and (*ii*) moving the first *k* elements to the end of the list – this corresponds to a clockwise rotation by –*k* positions.

```
>> rotate(L,k) : /* list rotation by k positions */
<n,k1> {! n=_len(L) !}  k==0? []: {! k1=k%n !} k<0? L[-k1:] + L[:-k1]:
                                                 L[n-k1:] + L[:n-k1];
>> ^rotate([ 5, 1, 4, 20, 15, 13 ],3);
[ 20, 15, 3, 5, 1, 4 ]
>> ^rotate([ 5, 1, 4, 20, 15, 13 ],-4);
[ 15, 3, 5, 1, 4, 20 ]
>>
```

Observe that two local variables (*n* and *k*1) are used in the function *rotate* to simplify its implementation.

A more elaborated example of list manipulation is: providing a compressed representation of a list that replace replicated consecutive elements, compatible for == and !=, with a single

instance, followed by the number of its occurrences . For example, the compression for the list L = [1, 1, 2, 2, 3, 3, 3, 3, 4, 1, 1, 2, 2, 2, 2, 4, 4, 4, 4] is [[1, 2], [2, 2], [3, 4], [4, 1], [1, 2], [2, 4], [4, 4]].

```
>> count(x,L,n) : L==[]||L[.]!=x?[L,n]: count(x,L[>],n+1);
>> compr(_,_) : null; /* dummy definition */
>> compr1(x, M, T) : compr(T[0], M+[[x, T[1]]]);
>> compr(L,M) : L==[]? M: compr1(L[.],M,count(L[],S,0));
>> compress(L) : L==[]?[]: compr(L,[]);
>> L = [1, 1, 2, 2, 3, 3, 3, 3, 4, 1, 1, 2, 2, 2, 2, 4, 4, 4, 4];
>> ^compress(L);
[ [ 1, 2 ], [ 2, 2 ], [ 3, 4 ], [ 4, 1 ], [ 1, 2 ], [ 2, 4 ], [ 4, 4 ] ]
<<
```

The function *compress* receives a list L and, if the list is not empty, it calls the function *compr* by passing L and an empty list M that will collect the compressed representation of L. This function in turn calls *compr1* by passing the head element of L, M and the result returned by the function *count*, which is called with the following actual parameters: the head element (denoted by *x*), L and the number *n* of occurrences of *x* set to 0. The function *count* recursively computes the number *n* of consecutive occurrences of *x* and eventually returns the pair [SL, *n*], where SL is the sublist starting with an element different from *x*. The function *compr1* recursively calls *compr* by passing the sublist not yet scanned and the list M extended with the pair: element *x* and its number *n* of occurrences. The role of *compr1* is to introduce an intermediate step in the recursive calls of *compr* for extracting the two elements of the pair [SL, *n*] returned by *count*.

The call of *compr* to *compr1* can be avoided by using a local variable in *compr* for storing the result of the call to *count*. The rewriting of *compr* is shown next:

```
>> compr(L,M) : L==[]? M: {! T=count(L[.],L,0) !} compr(T[0],M+[[L[.],T[1]]]);
>> ^compress(L);
[ [ 1, 2 ], [ 2, 2 ], [ 3, 4 ], [ 4, 1 ], [ 1, 2 ], [ 2, 4 ], [ 4, 4 ] ]
<<
```

As a further example of list manipulation, consider this problem: given three list L, SL1 and SL2, replace all sub-lists SL1 in L with SL2. For example, given L as above (i.e., L = [1, 1, 2, 2, 3, 3, 3, 3, 4, 1, 1, 2, 2, 2, 2, 4, 4, 4, 4]), SL1 = [1, 2] and SL2 = [-1, -2, -4], the result is [1, -1, -2, -4, 2, 3, 3, 3, 3, 4, 1, -1, -2, -4, 2, 2, 2, 4, 4, 4, 4].

```
>> checkSL(L,SL) : SL==[]? [true,L]: L==[]||SL[.]!=L[.]? [false,L]:
                                                  checkSL(L[>],SL[>]);
>> replace(L,SL1,SL2) : /* replace sublists in a list */
<cRes> L==[]? []: {! cRes = checkSL(L,SL1) !} cRes[0]?
                                          SL2[:]+replace(cRes[1],SL1,SL2):
                                          [L[.]|replace(L[>],SL1,SL2)];
>> ^replace(L,[1,2],[-1,-2,-4]);
[ 1, -1, -2, -4, 2, 3, 3, 3, 3, 4, 1, -1, -2, -4, 2, 2, 2, 4, 4, 4, 4 ]
<<
```

To solve the problem, we first define the function *checkSL* that checks whether the sub-list SL is the prefix of the list L or not. This function returns a pair [*b*, R], where *b* is either *true* or *false* and R is the portion of L that has not been scanned by the function. In case of success, *b* = *true* and R is the sublist of L following the retrieved pattern. The function *replace* preliminary calls *checkSL*, passing the current list L and the sub-list SL1, and stores the result into the pair *cRes*. If the first element of *cRes* is *true*, replace SL1 with SL2 in the result, recursively computed by *replace* applied to the postfix of L, or otherwise concatenate the head of L with the result, recursively computed by *replace* applied to the tail of L.

The technique of recurvely calling a function from inside a Global Setting Command and, thereafter, storing the result into a local variable, may be used to simplify the writing of a function for finding all sub-sets of a given set, represented by a list.

```
>> add1E(L,E) : /* given a list L of sublists, add E to each sublist of L */
   L==[]? []:  [ [E|L[.]] |add1E(L[>],E)];
>> subSet(L) : /* construct all subsets of a set */
  <R> L==[]? [[]]: {! R = subSet(L[>]) !} R + add1E(R, L[.]);
>> L=[1,2,3];
>> ^subSeT(L)%>;
[     [  ],
      [ 3 ],
      [ 2 ],
      [ 2, 3 ],
      [ 1 ],
      [ 1, 3 ],
      [ 1, 2 ],
      [ 1, 2, 3 ]
]
<<
```

The printing option `%>` enables the display of a subset per line.

As last example of this section, we present a function for finding all permutations of the elements of a list.

```
>> ins1E(E,Q,i,n) : /* given a list Q and an element E, insert E into Q in all
possible positions and return the list of all the possible insertions */
   i >= n? []: [ Q[:i]+[E]+Q[i:] | ins1E(E,Q,i+1,n) ];
>> add1P(E,P,n) : /* given the list P of all permutations of n-1 elements,
construct all the permutations by adding the element E */
  P==[]? [] : ins1E(E,P[.],0,n)+add1P(E,P[>],n);
>> perm1(L,n) : /* construct all permutations of the last n elements of L */
  L[>]==[]? [[L[.]]]: add1P(L[.],perm1(L[>],n-1),n);
>> perm(L) : /* construct all permutations of a list */
  L==[]? [[]]: perm1(L,_len(L));
>> ^perm(L)%';
[     [ 1, 2, 3 ],
      [ 2, 1, 3 ],
      [ 2, 3, 1 ],
      [ 1, 3, 2 ],
      [ 3, 1, 2 ],
      [ 3, 2, 1 ]
]
<<
```

The function *perm* calls *perm1* by passing the list to be permuted and its length. In turn, given a list *L* with *n* element, *perm1* recursively computes all the permutations of the last *n*-1 elements of *L* and, then, calls *add1P* to add the first element of *L* to each of such (*n*-1)-permutations in order to get all possible *n*-permutations. Finally, *ins1E* receives a (*n*-1)-permutation *Q* and an element *E* and return the set of all *n*-permutations obtained by inserting *E* into *Q* in all possible positions, from 0 to (*n*-1).

5.2. Playing with Strings

Let us first recall the basic properties of a string, that is an immutable (possibly, empty) sequence of elements of type *char* – the elements are numbered with an index from 0 on. Given a string S and an index *i*,

- S[*i*] denotes the element *i* of S;
- S[*i*:] is the substring of S from the element *i* to the end;
- S[:*i*] is the substring of S from the element 0 to the element *i*-1;

- S[*i*:*j*] is the substring of S from the element *i* to the element *j*-1, where *j* is a second index;
- S[:] returns the whole string S (and, as a string is stored once, it is useless).

Observe that `S[.]` cannot be used to denote `S[0]` as it happens for lists. Furthermore, given a string *st*, the built-in function *_ind*(S,st) returns the index within the string S of the first occurrence of *st* or -1 if *st* does not occur in S. In addition, if a third argument *i* of type *int* is added, then *_ind*(S,*st*,*i*) returns the index within the string S of the first occurrence of the string *st*, starting at the index *i* or -1 if *st* does not occur in S, starting at *i*.

Given a list L of characters, the casting operator `L@string` returns the string consisting of all characters in L. The reverse casting operator is `S@list` that, given a string S, returns the list of all characters in S. The casting operator *@string* can be also applied to a character to produce a single-character string.

Two strings can be concatenated by the operator '+' and be compared under the lexicographic order with any of relational operators (i.e., "==", "!=", ">", ">=", "<", "<="). Membership of a character *c* to a string S can be simply tested by issuing the command:

```
_ind(S,c@string)>=0,
```

that searches within S for an occurrence of the string generated by casting the character c.

A general scheme for scanning the elements of a string S is to use an index *i* to address each character S[*i*] and to vary *i* from 0 to *n*-1, where *n* = *_len*(S). The same scheme can be used for a list L, but its performance is not satisfactory: addressing an element L[i] requires the scanning of all previous elements and computing *_len*(L) is done by a whole list scanning. For this reason, the "natural" scanning the elements of a list is recursively done in CalcuList by processing the head list element and then passing to the list tail. The difference is well evidenced by looking at the performance of a generic function *hasPropS,* checking whether there exists an element with a given property defined by a function parameter *f* – note that *hasPropS* is a generalization of the membership problem.

```
>> /* checking whether all elements of a string or a list have the property
defined by the generic function f */
>> hasPropS1(S,f/1,i,l) : i >= l? false: f(S[i])? true: hasPropS1(S,f,i+1,l);
>> hasPropS(S,f/1) : hasPropS1(S,f,0,_len(S));
>> L=rep(['0','1','0'],500); /* create a list by repeating a given char sublist
500 times */
>> S=L@string; /* convert the list L into the string S */
>> ^hasPropS(S, lambda x:x=='2'); /* checks whether the character '2' is in the
string S */
false
>> ^clops;
694870
>> ^hasPropS(L,lambda x:x=='2'); /* the same check in the list L */
false
>> !clops; /* the function hasPropS is not efficient for a list */
27754885
>> /* generic property check function specialized for a list */
>> hasPropL(L,f/1) : L==[]? false: f(L[.])? true: hasPropL(L[>],f);
>> ^hasPropL(L, lambda x:x=='2');
false
>> !clops;
577736
>>
```

In the above session, a list of characters is constructed by repeating the sublist "['0','1','0']" 500 times using the function *rep*, previously defined in this section. The example confirms that the scheme of *hasPropS* is not efficient for a list: given a list and a string with similar

contents, the number of clops grows from 694,870 for a string up to 27,754,885! Observe that the version *hasPropL* of *hasPropS* targeted for list is much more efficient: the number of clops is 577,736. Since the latter number is also less than the number of clops for the case of a string, the example also confirms that scanning a list of characters by recursively access to its sublist heads is more efficient than scanning a string.

A complementary function to *hasPropS* is *allPropS* checking whether all characters in a string enjoy a given property defined by a function parameter *f*, e.g., the string consists of all letters or of all decimal digits. Again the same function may be used for lists but, for efficiency reasons, it is convenient to define a specific function for them.

```
>> /* checking whether all elements of a string or of a list have the property
defined by the generic function f */
>> allPropS1(S,f/1,i,l) : i >= l? true: f(S[i]) && allPropS1(S,f,i+1,l);
>> allPropS(S,f/1) : allPropS1(S,f,0,_len(S));
>> isDigit(X) = '0' <= X && X <= '9';
>> ^allPropS(S,isDigit); /* checks whether all characters of the string S are
digits */
true
>> ^clops;
829870
>> ^allPropS(L,isDigit); /* the same check in the list L */
true
>> !clops; /* the function allPropS is not efficient for a list */
27889885
>> /* generic all property check function specialized for a list */
>> allPropL(L,f/1) : L==[]? true: !f(L[.])? false: allProp(L[>],f);
>> ^allPropL(L,isDigit);
true
>> !clops;
732236
>>
```

Recall that L is a list with the pattern `'0','1','0'` is repeated 500 times and S is the list obtained by casting L to string.

The above example points out that (*i*) scanning a list through the slice operator "[i]" is rather inefficient (27,889,885 vs 732,236) and (*ii*) scanning a list of characters by recursively access to its sublist heads is slightly more efficient than scanning a string (732,236 vs 829,870).

A further confirmation for the point (*i*) comes out from the next example, where the function *isPalindrome,* defined for list in Section 5.1, is now implemented for strings. The point (*ii*) is more controversial as we shall discuss later.

```
>> /* checking whether a string or a list is a palindrome */
isPal1(S,i,j) : i>j? true: S[i]==S[j] && isPal1(S,i+1,j-1);
>> isPalindromeS(S) : isPal1(S,0,_len(S)-1);
>> ^isPalindromeS(S); /* checks whether the string S is a palindrome */
true
>> ^clops;
346126
>> ^isPalindromeS(L); /* checks whether the list L is a palindrome */
true
>> !clops; /* the function isPalindromeS is not efficient for a list */
27406141
>> ^isPalindrome(L); /* specific palindrome check for a list */
true
>> !clops;
1142400
>>
```

Again scanning a list through the slice operator "[i]" is rather inefficient (27,406,141 vs 1,142,400). The performance comparison between strings and lists is now in favor of the strings (346,122 vs 1,142,400). This is caused by the backward scanning of the data structure with the index *j*, which cannot be efficiently implemented in a list as for the foreward case with the index *i*.

So far, we have only presented operations that perform checks on strings. When a string is to be modified, a new string must be constructed as strings are immutable. As constructing a string one character at a time would generate all possible prefix substrings, it is convenient to first construct a list and, then, transform the list into a string. We show how to proceed with two examples: (1) converting all of the characters of a string to upper case and (2) reversing a string. Observe that in the example below, the palyndrome check is performed on a rather long latin string.

```
>> toUpperCase(X) : 'a' <= X && X <= 'z'? ('A'+X-'a')@char: X;
>> toUpS1(S,i,l) : i>=l? []: [toUpperCase(S[i]) | toUpS1(S,i+1,l)];
>> toUpperCaseS(S) : /* convert all characters of a string to upper case */
    toUpS1(S,0,_len(S))@string;
>> ^toUpperCaseS("CalcuList");
CALCULIST
>> ^isPalindromeS(toUpperCaseS("InGirumImusNocteEtConsumimurIgni"));
true
>> revS1(S,L,i,l) : i>=l? L: revS1(S,[S[i]|L],i+1,l);
>> reverseS(S) : /* reversing a string */
   revS1(S,[],0,_len(S))@string;
>> ^reverseS(S);
010010 … 010010
>> !clops;
609420
>> ^reverse(S@list)@string; /* calling list reversing */
010010 … 010010
>> !clops;
471328
>>
```

The function *toUpperCaseS* calls *toUpS1* by passing a string S, the start index *i*=0 and the length *n* of S. Each character of S is scanned, converted to upper case and inserted into a list of characters, which will be eventually transformed into a string when returned to *toUpperCaseS*.

The function *reverseS* calls *revS1* by passing a string S, an empty list L, the start index *i*=0 and the length *n* of S. The characters of S are scanned and inserted into L in the reverse order, following the approach of the function *reverse*, defined in Section 5.1 for reversing a list. As shown next, the function *reverse* can be also used for a string after its conversion to a list and, once again, a function scanning a list in forward direction works faster than the one scanning a string with an index (471,328 vs 609,420).

Let us now consider a typical problem for a string: replacing a substring of a string. To this end, we may use the function replace, defined in Section 5.1 to replace a sublist of a list. As an example we show gow to remove spaces in a string and how to create a mail marge letter.

```
>> replaceS(S,S1,S2) : /* specific palindrome check for a list */
   replace(S@list, S1@list, S2@list)@string;
>>^replaceS("In Girum Imus Nocte Et Consumimur Igni"," ", "");
InGirumImusNocteEtConsumimurIgni
>>^replaceS("Dr. <name>\naddress\n\ndear <name> how are you?",
            "<name>", "Mark");
Dr. Mark
address
```

```
dear Mark how are you?
>>
```

In cryptography, a *Caesar cipher*, also known as *Caesar's code*, is one of the simplest and most widely known encryption techniques. The method is named after Julius Caesar, who used it in his private correspondence, and it is a type of substitution cipher in which, given an alphabet with n symbols (indexed from 0 to n-1) and a right shift k with $0 \le k \le n$-1 (typically denoted by the symbol with index k), each letter (say with index p) in a text is replaced by a letter k positions down the alphabet, by cycling to the start if $p+k > n$-1. For example, with the English alphabet consisting of 26 letters and a right shift of 3 (i.e., the key is the letter D), A would be replaced by D, B would become E, and so on until W, which becomes Z; thereafter, X is replaced by A, Y by B and Z by C. The decoding is done using the encoding key to make a left shift.

The encoding / decoding with the Caesar cipher can be easily implemented in CalcuList by making the following assumptions: the alphabet is the English one with 26 symbols from A to Z, spurious (i.e., non-letter) characters in the text are simply skipped while all letters are converted in the upper case format.

The function *mod26*(k) coincides with the classical module function k % 26 if $k \ge 0$; otherwise, it returns 26 + k % m, which is in the range (1, 25).

```
>>indL(X) : X-'A'; /* offset w.r.t. 'A': values from 0 to 25 */
>>isLetter(X)& : 'a' <= X && X <= 'z' || 'A' <= X && X <= 'Z';
>>mod26(k) : /* computes k % 26 or k % 26 + 26 if k is negative */
  <tmp> {! tmp=k%26 !} {! tmp=tmp<0? tmp+26: tmp !} ('A'+tmp)@char;
>>codeDecL(X, k, cd) : cd=='C'? mod26(indL(X)+k): mod26(indL(X)-k);
>>codeDecC(S,k,cd,i,n) : i>=n? []: isLetter(S[i])?
                  [codDecodL(toUpperCase(S[i]),k,cd) | codDecC(S,k,cd, i+1,n)]:
                  codDecC(S,k,cd,i+1,n);
>>codeC(S,K) : isLetter(K)?
                 codeDecC(S,indL(toUpperCase(K)),'C',0,_len(S))@string:
                 exc("wrong key '"+K+"'");
>>decodeC(S,K) : isLetter(K)?
                 codeDecC(S,indL(toUpperCase(K)),'D',0,_len(S))@string:
                 exc("wrong key '"+K+"'");
>>CODE=codC("attack the Gauls at dawn",'H'); /* coding */
>>^CODE
HAAHJRAOLNHBSZHAKHDU
>>^decodeC(CODE,'H'); /* decoding */
ATTACKTHEGAULSATDAWN
>>
```

Observe that if the single character key is not a letter, an exception is raised and the program terminates.

A *poly-alphabetic cipher* is any cipher based on substitution, using multiple substitution alphabets. The *Vigenère cipher* is probably the best-known (and definitely the most-simple) example of a poly-alphabetic cipher: it consists of several Caesar ciphers in sequence with different shift values. In other words, the key is not a single letter but a string whose letters become key in a circular fashion. For example, suppose that the "plaintext" to be encrypted is "ATTACKTHEGAULSATDAWN" and the key is "HUGE". The first letter 'A' is encrypted by the first letter of the key (i.e., 'H'), then 'T' by the subsequent key letter 'U', 'T' by 'G' and 'A' by the last key letter 'E'; after encoded the first block of letters of the same length of the key (in the example 4), the subsequent 4-letter blocks of the plaintext are encoded by the four letters of the key.

The implementation of the *Vigenère cipher* follows. The function *convertKey* transform the key from a string of characters to a list of letter indices – also in this case, if a character key is not a

letter, an exception is raised and the program terminates. The function *codDecV* must include among its parameters non only the current sublist of the key but also the whole key, sot that it can be circularly resumed. There are two distinct functions for encoding and for decoding, respectively *codV* and *decV*.

```
>>convertKey(K,i,n) : /* convert key chars to indices stored into a list */
             i >=n? []: isLetter(K[i])?
                        [indL(toUpperCase(K[i]))|convertKey(K,i+1,n)]:
                        exc("wrong key '"+K+"'");
>>isCorrectK(K[])& : K==[]? false: K[*]==[]? isUpperCase(K[^]):
                           isUpperCase(K[^])&&isCorrectK(K[*]);
>>codeDecV(S,K1,K,cd,i,n) : i >=n? []: isLetter(S[i])?
       [codeDecL(toUpperCase(S[i]),K1[.],cd) | codeDecV(S,K1[>]==[]? K: K1[>],
         K,cd,i+1,n)]: codeDecV(S,K1, K,cd,i+1,n);
>>codeV(S,K) : <LK> {! LK=convertKey(K,0,_len(K)) !}
                  codeDecV(S,LK, LK, 'C',0,_len(S))@string;
>>decodeV(S,K) : <LK> {! LK=convertKey(K,0,_len(K)) !}
                  codeDecV(S,LK, LK, 'D',0,_len(S)) @string;
>>CODE=codeV(text,"HUGE"); /* coding */
>>^CODE;
>>HNZEJEZLLAGYSMGXKUCR
<<^decodeV(CODE,"HUGE"); /* decoding */
>>ATTACKTHEGAULSATDAWN
<<
```

5.3. Playing with Jsons

Let us first recall the basic properties of a json (*JavaScript Object Notation*). In CalcuList, a json object is a (possibly empty) sequence of fields separated by comma and enclosed in curly braces. A field is a pair (*key*, *value*) separated by a colon: *key* is a string and *value* can be of any type: *double, int, char, bool, string, null, list* and (recursively) *json.* All fields of a json object have different keys.

Given a json J and a key *k*, J[*k*] denotes the value of the field of J with key equal to *k* – if there is no field with key equal to *k* then the value *null* is returned. Given a json J, a key *k* and a value *v*, `J[k]=v` modifies the value for the field if J includes the key *k* or otherwise, J is extended with a new field with key *k*. To clone a json J, it is sufficient to type `J[:]`.

Two jsons J1 and J2 can be compared only with the operators '==' and '!='. We have that `J1==J2` is *true* if and only if J1 and J2 points to the same data structure in the heap and `J1!=J2` is *true* if and only if `J1==J2` is *false*.

A json J = { key_1: val_1, ..., key_n: val_n } is converted into a list L = [[key_1, val_1], ..., [key_n, val_n]] by issuing the command `J@list`. On the other hand, J is obtained from L by issuing the command `L@json`. The length of a json J, i.e., the number *n* of its fields, can be obtained by invoking the built-in function `_len(J)`.

A field of a json J, say with key *k*, can be removed by setting `J[k]=#null`. In addition, if J[k] is in turn a list or a json then all elements of J[k] are physically removed by inserting them into the heap garbage list. Finally, the assignment "J=#null" both sets J to *null* and physically removes all fields of J.

A json is displayed by enclosing its fields between curly braces, inserting a comma between two contiguous fields and representing each pair key and value, say the string st and the value v, as `st : v`. The print option %* indents the json fields as well theirs sub-elements, whereas the print option %> indents the field elements at the first level only.

To illustrate advanced functions for handling jsons, we first introduce the notion of *Json Schema*, that is a specification for json based format for defining the structure of a json value

(indicated also as json data or *object*). A json schema has the same syntax of a jason value but the keys are predefined and they describe a data model for the interpretation of json values. A json value, interpreted according to this data model, is called an "instance" of it.

A crucial key for a json schema is “type”, that has one of the following seven primitive types:

1. *null*: an instance must be the null value;
2. *boolean*: an instance must have the "true" or "false" value
3. *object*: an instance is a json with a number of fields specified by the key “properties”
4. *array*: an instance is an ordered list of elements, whose structure is specified by the key “items”
5. *number*: an instance is an arbitrary-precision, base-10 decimal number value
6. *integer*: an instance is an integer;
7. *string*: an instance is a Unicode string,

As an example, we define the json schema for the json value stored into the variable *dept* described in Section 2.3:

```
{
     "deptName": "unical",
     "emps": [
          {
               "firstName": "Mimmo",
               "lastName": "Sacca",
               "age": 30
          },
          {
               "firstName": "Anna",
               "lastName": "Rossi",
               "age": 32,
               "projects": [
                    "p1",
                    "p2"
               ]
          },
          {
               "firstName": "Luisa",
               "lastName": "Cosentino",
               "age": 28,
               "projects": [
                    "p1",
                    "p3"
               ]
          }
     ]
}
```

A json schema for it is:

```
deptSchema = {
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "employee json schema",
  "description": "Schema of a dataset on a university's employees",
  "properties": {
    "deptName": {
      "type": "string",
```

```
      "description": "name of the university"
    },
    "emps": {
      "type": "array",
      "description": "list of employees",
      "items": {
        "type": "object",
        "description": "an employee",
        "properties": {
          "firstName": {
            "type": "string"
          },
          "lastName": {
            "type": "string"
          },
          "age": {
            "type": "integer"
          },
          "projects": {
            "type": "array",
            "description": "list of projects",
            "items": {
              "type": "string",
              "description": "project name"
            }
          }
        },
        "required": [
          "firstName",
          "lastName",
          "age"
        ]
      }
    }
  },
  "required": [
    "deptName",
    "emps"
  ]
}
```

The schema is a json with six fields:

1. "$schema": it states that this schema is written according to the draft v4 specification;
2. "type": an instance must by an object (json value);
3. "title": simply a name for the schema;
4. "description": a brief description of the schema;
5. "properties": it is a json stating that an instance must have the fields *deptName* and *emps* with given characteristics – their usage is not mandatory but, if used, they must have the prescribed structure;
6. "required": it is a list whose elements are the keys of the above fields that must be mandatorily defined – in the example both fields *deptName* and *emps* are mandatory in an instance.

The json associated to "properties" has two fields: *deptName,* whose value is a json stating that the instance value for this key must be a string, and *emps,* whose value is a json stating

that the instance value for this key must be an array (list), whose elements must have the characteristics defined by the field "items". The json associated to "items" states that each element in the list for *emps* is a json (denoting an employee) with two fields: "properties", stating that an employee instance must have the fields *firstName, lastName, age* and *projects,* and "required", stating that the first three fields must be mandatorily present in an instance, whereas *projects* is optional. The types for the four fields are described by suitable json stating that *firstName* and *lastName* are strings in an instance, *age* is an integer and *projects* is a list (array) of strings (denoting project names).

Next, we present a function that checks whether a json is actually an instance of a given schema. We stress that, for simplicity, the check only considers the schema properties discussed above and, therefore, do not analyze additional json schema fields such as "minimum" (minimum acceptable value for the instance), "exclusiveMinimum" (if true than the value must be strictly greater than the value of "minimum"), "maximum" (maximum acceptable value for the instance), "exclusiveMaximum" (if true than the value must be strictly less than the value of "maximum"), "multipleOf" (an acceptable value for an instance must be a multiple of this value), "minLength" (minimum number of characters for a string instance), "maxLength" (maximum number of characters for a string instance), and "pattern" (a string instance is considered valid if the regular expression matches the instance successfully).

```
>>checkJSobj(_,_) : null; /* defined next */
>>checkJSarray(_,_): null; /* defined next */
>>checkJSreq(LR,J) : /* all fields in the list must occur in the instance */
LR==[]? true: _isKey(J,LR[.]) && checkJSreq(LR[>],J);
>>checkJSall(S,J) : /* validate a type instance against the schema */
  <t> {! t=S["type"] !} t=="object"? checkJSobj(S,J):
   t=="array"? checkJSarray(S,J): t=="string"?
   J@type==string: t=="null"? J==null:
   t=="boolean"? J@type==bool: t=="integer"? J@type==int:
   t=="number"? J@type==int || J@type==double: false;
>>checkJSfield(K,V,J) : /* check whether a schema key has a type and the
instance has that key */
   !_isKey(V,"type") || !_isKey(J,K)? true: checkJSall(V,J[K]);
>>checkJSprop(SP,J):  /* validate the instance against all fields in
"properties"*/
   SP==[]? true: checkJSfield(SP[.][0],SP[.][1],J) && checkJSprop(SP[>],J);
>>checkJSobj(S,J): /* validate a json instance against an object schema */
     J@type==json &&
     ( !_isKey(S,"properties") || checkJSprop(S["properties"]@list,J) ) &&
     ( !_isKey(S,"required") || checkJSreq(S["required"],J) );
>>checkJSlist(S,LJ):/* validate an elements of a list against the item schema */
     LJ==[]? true: checkJSall(S,LJ[.]) &&  checkJSlist(S,LJ[>]);
>>checkJSarray(S,J):/* validate list element instances against an item schema */
     <o> J@type!=list? false: !_isKey(S,"items")? true: {! o=S["items"] !}
     o@type==json && ( !_isKey(o,"type") || checkJSlist(o,J));
>>checkJS(S,J) : /* validate a json instance against the schema */
     S@type==json && ( !_isKey(S,"type") || checkJSall(S,J) );
>>
```

The function *checkJS*(S,J) validates the json J against the schema S. If S has no type, then J is always true. Otherwise, *checkJSall*(S,J) performs the validation for all admissible values for type, using a number of other functions.

We next give an example of validation on the json that was written into the file "dept_unical.dat" in Section 2.3. To this end, we read the json now into a global variable, say `dept`.

```
>> dept = <<("dept_unical.dat"); /* input from a file */
- reading data from file 'dept_unical.dat'
- data are correct
>>^dept %*;
{
     "deptName": "unical",
     …

}
>> deptSchema = {
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "employee json schema",
  …
};
>>^checkJS(deptSchema,dept);
true
>>
```

In the following, we shall present a number of further functions for handling jsons, using the json datasets on nobel prizes published at the link:

https://github.com/jdorfman/awesome-json-datasets#nobel-prize

After clicking on the first dataset, named "Prize", the browser displays a dataset that is a json with the following schema:

```
prizeSchema = {
     "type": "object",
     "title": "nobel_prizes json schema",
     "description": " ",
     "properties": {
          "prizes": {
               "type": "array",
               "description": "list of all nobel prizes, ordered by year",
               "items": {
                    "type": "object",
                    "description": "a nobel prize given a year for a category",
                    "properties": {
                         "year": {
                              "type": "string"
                         },
                         "category": {
                              "type": "string",
                              "description": "6 categories: Physics, Chemistry,
Medicine, Literature, Peace, Economy"
                         },
                         "laureates": {
                              "type": "array",
                              "description": "list of the winners for that
category and that year - from 1 to 3 laureates sharing the same prize",
                              "items": {
                                   "type": "object",
                                   "description": "the winner or one of the co-
winners",
                                   "properties": {
                                        "id": {
                                             "type": "string"
                                        },
                                        "firstName": {
                                             "type": "string"
                                        },
                                        "surname": {
                                             "type": "string"
```

```
                                        },
                                        "motivation": {
                                             "type": "string"
                                        },
                                        "share": {
                                             "type": "string",
                                             "description": "from 1 to 4"
                                        }
                                   }
                              }
                         }
                    }
               }
          }
     }
}
```

The dataset consists of only one json with exactly one field, whose key is "*prizes*" and the value is a list of jsons, all sharing the following structure, described in informal format:

```
{ "year": <value_from_1901_to_2016>,
  "category": <Phisics_Chemistry_Medicine_Literature_Peace_Economy>,
  "laureates": <list_of_1_to_3_winners>
}
```

The above json represents the Nobel prize assignment for a year and for a category to a list of winners (called *laureates*). The laureates can be an integer from 1 to 4 and each laureate is a json with the following structure:

```
{ "id": <id_of_laureate>,
  "firstname": <firstname_string>,
  "surname": <surname_string>
  "motivation": <motivation_string>
  "share": <1_4>
}
```

Note that if the number of laureates is 1 then the prize share is 1 as well; if it is 2 then both laureates have the prize share of 2 (i.e., they will get 1/2 of the prize); if it is 3 then either the three shares are equal to 3 (i.e., they will get 1/3 of the prize) or one of the laureates has share 2 (i.e., s/he gets 1/2 of the prize) whereas the other two have share 4 (i.e., the two of them will get 1/4 of the prize).

We copy the whole file content and past it into a text file, named "nobel_prizes.dat". As this file contains a number of substrings "\/" that is not considered as a valid escape sequence, we modified such substrings into the sinple character "/" using an editor. Then we load the file into a CalcuList session, as shown next. Note that three motivations are longer than 256 charactters and are, therefore, truncated.

```
/* start a new session */
>> prizeSchema = {
     "type": "object",
     "title": "nobel_prizes json schema",
     …
};
>>T=<<("…/nobel_prizes.dat"); /* read nobel prize dataset from a text file */
- reading data from file '…/nobel_prizes.dat'
Warning: string truncated to max length 256
Warning: string truncated to max length 256
Warning: string truncated to max length 256
- data are correct
>>^T%*; /* select the list of all prizes */
{
```

```
     "prizes": [
          { /* FIRST PRICE ASSIGNED TO 3 LAUREATES */
               "year": "2016",
               "category": "physics",
               "laureates": [
                    {
                         "id": "928",
                         "firstname": "David J.",
                         "surname": "Thouless",
                         "motivation":   ""for   theoretical   discoveries   of
topological phase transitions and topological phases of matter"",
                         "share": "2"
                    },
                    {
                         "id": "929",
                         "firstname": "F. Duncan M.",
                         "surname": "Haldane",
                         "motivation":   ""for   theoretical   discoveries   of
topological phase transitions and topological phases of matter"",
                         "share": "4"
                    },
                    {
                         "id": "930",
                         "firstname": "J. Michael",
                         "surname": "Kosterlitz",
                         "motivation":   ""for   theoretical   discoveries   of
topological phase transitions and topological phases of matter"",
                         "share": "4"
                    }
               ]
          },
            /* OMISSIS */
          { /* LAST PRICE ASSIGNED TO 2 LAUREATES */
               "year": "1901",
               "category": "peace",
               "laureates": [
                    {
                         "id": "462",
                         "firstname": "Jean Henry",
                         "surname": "Dunant",
                         "share": "2"
                    },
                    {
                         "id": "463",
                         "firstname": "Frédéric",
                         "surname": "Passy",
                         "share": "2"
                    }
               ]
          }
     ]
}
>>prizes=T["prizes"]; /* extract the list of all prizes */
>>^_len(prizes); /* total number of prizes */
579
>>^checkJS(prizeSchema,T);
true
>>
```

A remark on the “run configuration” is now in the order. As the dataset is not anymore a toy (but, definitely, not a big one either!), we launched Calculist with the following setting: EXECMS 256000 EXECOS 128000, thus the sizes of the two memories MEM and OUTPUT are increased to respectively 256,000 and 128,000 words. On the other side, the default size of

CODE is not changed because the large code produced to compile the json is written on an external file, which represents a sort of external extension of the memory CODE.

We next define a generic map function that, given a list of jsons, returns the list of values for a field according to the prescription specified into a generic function *f* with arity 3. The function argument *f* is instanciated during the session with the function *proj* – note that this function has two different behaviors according to the values 0 or 1 for the index *i_proj* and launches an exception if the index has a value different from 0 and 1. With the 0 option, *proj* returns the value of the field K, whereas it returns the length of this value with the 1 option.

```
>>mapP1(P,f/3,i_f, K) : P==[]? []: [ f(i_f,P[.],K) | mapP1(P[>],f,i_f,K) ];
>>mapPrizes(P,f/3,i_f,K) =+; mapP1(P,f,i_f,K);
>>proj(i_proj,J,K) : i_proj==0? J[K]: i_proj==1? _len(J[K]): exc("wrong index
for function");
>>numLaureates= mapPrizes(prizes,proj,1,"laureates"); /* list of numbers of
laureates for every prize */
>>^numLaureates;
[ 3, 3, 1, . . ., 1, 1, 2 ]
>>^_len(numLaureates); /* it is equal to _len(prizes)=total number of prizes */
579
>>sumL(L) : L==[]? 0: L[.]+sumL(L[>]); /* sum of the elements of a list*/
>>totNumLaureates = sumL(numLaureates); /* total number of prize recipients */
>>^totNumLaureates;
911
>>^totNumLaureates/_len(prizes);/* number of laureates per prize */
1.5734024179620034
>>
```

Next, we define a function reducePrizes that removes duplicates from a list of values for fields of prizes, previously mapped. Duplication is checked by means of the member function. Nest step consists of retrieving prize categories and years by combining mapping and reduction.

```
>>member(x,L) : L!=[] && (L[.]==x || member(x,L[>])); /* list membership */
>>reduceP1(MP,V) :   MP==[]? V: member(MP[.], V)? reduceP1(MP[>],V):
reduceP1(MP[>],[MP[.]|V]);
>>reducePrizes(MP) : reduceP1(MP,[]);
>>categories= reducePrizes(mapPrizes(prizes,proj,0,"category")); /* list of
distinct prize categories */
>>^categories;
[ "economics", "peace", "literature", "medicine", "chemistry", "physics" ]
>>^_len(categories); /* number of distinct prize categories */
6
>>years= reducePrizes(mapPrizes(prizes,proj,0,"year")); /* list of distinct
years */
>>^years%>;
[    "1901",
     "1902",
     "1903",
      ...
     "2015",
     "2016"
]
>>^_len(years); /* number of years when prizes were assigned */
113
>>^_len(prizes)/_len(years); /* average number of prizes per year */
5.123893805309734
>>^totNumLaureates/_len(years); /* average number of laureates per year */
8.061946902654867
>>
```

While playing with data, we discovered that a same prize can be shared by up to three persons. Next, we want to know wether a same person was winner of more than one prize in the various years. To answer this question, we construct a list L1 of elements, one for each laureate. The element contains the "id" of the laureate, the name and the sublist of all her/his prizes, each indicated by year and category.

The function *add1L* takes a laureate (id, firstname, surname), who is a prize winner for the year "year" and the category "cat", and creates a new record in L1 for her/him or, if such a record exists, extends the record with the new piece of information – the list L is an alias for L1 that is used to scan all the elements.

The function *addL* takes a year and a category for a prize and scans the list of winners for that prize in order to include them into the list L1 by invoking *add1L* for each of them. The function *nPL* scans the list of all prizes and, for each prize, invokes addL to update the list L1. Finally, the function *nPrizesLaurea* sets L1 to the empy list and then invokes *nPL*.

```
>>add1L*(id,L,firstname,surname,year,cat,L1) : L==[]? [ [id,firstname,surname,1,
[year,cat]] | L1 ]: id==L[.][0]? {! L[0][3]+=1 !} {! L[0][4]+=[year,cat] !} L1:
add1L(id,L[>],firstname,surname,year,cat,L1);
>>addL*(LL,L,year,cat) : <L1> LL==[]? L: {! L1= add1L(LL[.]["id"],L,
LL[.]["firstname"],LL[.]["surname"],year,cat,L) !} addL(LL[>],L1,year,cat);
>>nPL*(P,L) : <L1> P==[]?L: {! L1= addL(P[.]["laureates"], L, P[.]["year"],
P[.]["category"]) !} nPL(P[>], L1);
>>nPrizesLaurea*(P) : nPL(P,[]);
>>nPL_Laureates=nPrizesLaurea(prizes);
>>nPrizesLaurea_GT1(nPL) : nPL==[]?[]: nPL[.][3]>1? [ nPL[.] |
nPrizesLaurea_GT1( nPL[>]) ]: nPrizesLaurea_GT1(nPL[>]);
>>pluriLaureates = nPrizesLaurea_GT1(nPL_Laureates);
>>^pluriLaureates%>;
[  ["6", "Marie", "Curie, née Sklodowska", 2, [ "1911", "chemistry", "1903",
     "physics" ] ],
   [ "217", "Linus Carl", "Pauling", 2, [ "1962", "peace", "1954",
     "chemistry" ] ],
   [ "482", "Comité international de la Croix Rouge (International Committee of
the Red Cross)", "", 3, [ "1963", "peace", "1944", "peace", "1917", "peace" ] ],
   [ "66", "John", "Bardeen", 2, [ "1972", "physics", "1956", "physics" ] ],
   [ "222", "Frederick", "Sanger", 2, [ "1980", "chemistry", "1958",
     "chemistry" ] ],
   [  "515",  "Office  of  the  United  Nations  High  Commissioner  for  Refugees
(UNHCR)", "", 2, [ "1981", "peace", "1954", "peace" ] ]
]
>>
```

The list L1 is eventually stored into the variable *nPL_Laureates*. To filter the winners of more than one nobel prize, we write down the function *nPrizesLaurea_GT1*. It turns out that they are six in total but only three are individuals and not organizations.

Next we store the whole list of prizes back into a text file but, this time, we remove the fields "firstname" and "surname" from each json element in the list so that they get this simplified structure:

```
{ "id": <id_of_laureate>,
  "motivation": <motivation_string>
  "share": <1_4>
}
```

The two fields are dropped out but the information will not be lost, as we shall in a while import another dataset containg a full description of every prize winner.

The writing of the revised version of prizes is done using three functions with advanced printing features: *printNobelPrizes* prints open and closed square parentheses and, in

between, launches *print1NP*, which prints all scalar fields about each prize while the list of laureates is printed by *printLaureates*, which drops out the two fields "firstname" and "surname" and also computes the number of laureates, that will be added as a final field by *print1NP*.

```
>>printLaureates(L) : L==[]? 0:
        {^ "{ \"id\":" %+ L[.]["id"] %+", " ^}
        {^ "\"motivation\":" %+ L[.]["motivation"]%" %+ ", " ^}
        {^ "\"share\":" %+ L[.]["share"]%" %+ "} " ^} {^ L[>]==[]? null: ", " ^}
         1+printLaureates(L[>]);
>>print1NP(L) : <n> L==[]? null:
         {^ "{\"year\":" %+ L[.]["year"] ^}
         {^ ", \"category\":" %+ L[.]["category"]%" ^}
         {^ ", " %+ "laureates"%" %+ ": [" ^}
         {! n=printLaureates(L[.]["laureates"]) !}
         {^ " ]" ^}
         {^ ", \"numLaureates\":" %+ n %+ " }" ^}
         {^ L[>]==[]? " ": ",\n " ^}
         print1NP(L[>]);
>>printNobelPrizes(P)={^ "[ " ^} print1NP(P) {^ " ]" ^};
>> ^>>(".../nobel_prizes_1.dat")printNobelPrizes(prizes); /* redirect the output
into a file */
- written output to file '.../nobel_prizes_1.dat'
>>prizes_1=<<(".../nobel_prizes_1.dat");
>>^prizes_1%*;
[
     {
          "year":2016,
          "category":"physics",
          "laureates": [
               {
                    "id":928,
                    "motivation":"\"for theoretical discoveries of topological
phase transitions and topological phases of matter\"",
                    "share":"2"
               },
               {
                    "id":929,
                    "motivation":"\"for theoretical discoveries of topological
phase transitions and topological phases of matter\"",
                    "share":"4"
               },
               {
                    "id":930,
                    "motivation":"\"for theoretical discoveries of topological
phase transitions and topological phases of matter\"",
                    "share":"4"
               }
          ],
          "numLaureates":3
     },
     ...
     {
          "year": 1901,
          "category": "peace",
          "laureates": [
               {
                    "id": 462,
                    "motivation": null,
                    "share": "2"
               },
               {
                    "id": 463,
```

```
                    "motivation": null,
                    "share": "2"
               }
          ],
          "numLaureates": 2
     }
]
>>
```

Note that we use the redirection of the input to load the revised version of prizes into the variable *prizes_1*. The content of this variable is displayed by the subsequent command `^prizes_1%*` – for obvious reasons of space, we just show the first and the last prize.

Let us now return to the link:

https://github.com/jdorfman/awesome-json-datasets#nobel-prize

This time we click on the second dataset, named "Laureate" and the browser displays a json with exactly one field, whose key is "*laureates*" and the value is a list of jsons. The schema is:

```
laureateSchema = {
     "type": "object",
     "title": "nobel_laureates json schema",
     "description": "dataset \"http://api.nobelprize.org/v1/laureate.json\"",
     "properties": {
          "laureates": {
               "type": "array",
               "description": "list of all nobel prizes winners, ordered by id",
               "items": {
                    "type": "object",
                    "description": "a laureate, i.e., nobel prize winner",
                    "properties": {
                         "id": {
                              "type": "string",
                              "description": "number from 1 to 937"
                         },
                         "firstname": {
                              "type": "string"
                         },
                         "surname": {
                              "type": "string"
                         },
                         "born": {
                              "type": "string",
                              "description": "yyyy-mm-dd"
                         },
                         "died": {
                              "type": "string",
                              "description": "yyyy-mm-dd"
                         },
                         "bornCountry": {
                              "type": "string"
                         },
                         "bornCountryCode": {
                              "type": "string",
                              "description": "country codes stored into the
dataset \"country\""
                         },
                         "bornCity": {
                              "type": "string"
                         },
                         "diedCountry": {
                              "type": "string"
                         },
```

```
                        "diedCountryCode": {
                             "type": "string",
                             "description": "country codes stored into the
dataset \"country\""
                        },
                        "diedCity": {
                             "type": "string"
                        },
                        "gender": {
                             "type": "string",
                             "description": "at the moment, male or female"
                        },
                        "prizes": {
                             "type": "array",
                             "description": "list of all prizes winned by a
laureate",
                             "items": {
                                  "type": "object",
                                  "description": "a prize winned by a laureate",
                                  "properties": {
                                       "year": {
                                            "type": "string",
                                            "description": "from 1901 on"
                                       },
                                       "category": {
                                            "type": "string"
                                       },
                                       "share": {
                                            "type": "string"
                                       },
                                       "motivation": {
                                            "type": "string"
                                       },
                                       "affiliations": {
                                            "type": "array",
                                            "description": "list of
affiliations while working for the prize results",
                                            "items": {
                                                 "name": {
                                                      "type": "string"
                                                 },
                                                 "city": {
                                                      "type": "string"
                                                 },
                                                 "country": {
                                                      "type": "string"
                                                 }
                                            }
                                       }
                                  }
                             }
                        }
                   }
              }
         }
    }
}
```

The list of prizes for each laureate typically consists of one prize, while in 5 cases, it contains two prizes[5] and in one case, three prizes. The structure of the list elements is rather

[5] I was tempted to write "two prizes only" but eventually I refused to use the term "only" for a nobel prize!

elaborated but we omit to describe it as we shall remove the whole field when loading the file into a CalcuList session.

We proceed as for the previous file “nobel_prizes.dat”. We copy the whole web file content and past it into a text file, named “nobel_laureates.dat”. Also, this file contains a number of substrings “\/” that must be cleaned out using an editor. In addition, the fields “affiliates” often have value [[]], we replace this value with [] in order to simplify the schema. Then we load the file into a CalcuList session, by giving some keys for the fields we want to remove. We already mentioned the field with key “prizes”; we shall actually discard also “bornCountry”, “bornCity”, “diedCountry” e “diedCity”. Note that three motivations are longer than 256 charactters and are, therefore, truncated as it happened for the previous file “nobel_prizes.dat”. But this time truncation has no effect at all as motivations are within the scope of the discarded field “prizes”.

```
/* start a new session */
>>nobel_laureates=<<("…/nobel_laureates.dat","bornCountry","bornCity","diedCou
ntry", "diedCity", "prizes"); /* read nobel laureate dataset from a text file */
- reading data from file '…/nobel_laureates.dat'
Warning: string truncated to max length 256
Warning: string truncated to max length 256
Warning: string truncated to max length 256
- data are correct
>> laureateSchema = {
     "type": "object",
     "title": "nobel_laureates json schema",
     …
};
>>^checkJS(laureateSchema, nobel_laureates);
true
>>
```

We read again the file nobel_laureates, but this time with the option of discarding all fields whose keys are listed in the <<(*fileName*) command: "bornCountry", "bornCity", "diedCountry", "diedCity", "prizes".

```
/* start a new session */
>>nobel_laureates=<<("…/nobel_laureates.dat","bornCountry","bornCity","diedCou
ntry", "diedCity", "prizes"); /* read nobel laureate dataset from a text file
with the option of discarding a number of fields */
- reading data from file '…/nobel_laureates.dat'
Warning: string truncated to max length 256
Warning: string truncated to max length 256
Warning: string truncated to max length 256
- data are correct
>>laureates=nobel_laureates["laureates"]; /* extract the list of laureates */
>>^laureates[0]%*; /* display the first laureate in the list */
{
     "id": "1",
     "firstname": "Wilhelm Conrad",
     "surname": "Röntgen",
     "born": "1845-03-27",
     "died": "1923-02-10",
     "bornCountryCode": "DE",
     "diedCountryCode": "DE",
     "gender": "male"
}
>>^_len(laureates); /* total number of laureates */
910
>>^laureates[_len(laureates)-1]%*; /* display the last laureate in the list */
{
     "id": "937",
     "firstname": "Bob",
```

```
     "surname": "Dylan",
     "born": "1941-05-24",
     "died": "0000-00-00",
     "bornCountryCode": "US",
     "gender": "male"
}
>>
```

As the number of laureates is 910 whereas the code of the last t is “937”, it seems that some ids are missing. We wanted to investigate this matter by executing the session below. So, we first trasformed id’d values from strings to integers inside the list of laureates in order to get a type more flexible to do computations. Then we saved a copy of the updated list in the text file “.../nobel_laureates_1.dat”. At this point we were ready to compute the minimum and maximum id value and to identifies 27 missing ids – but we were not able to provide any explanation for the missing identifiers (some nobel prize aspirant?).

```
>>parseId1(s,v,i,n) : n==i? v: parseId1(s,v*10+s[i]-'0',i+1,n);
>>parseId(s) : parseId1(s,0,0,_len(s)); /* parse a string into an integer */
>>modId*(L) : L==[]? null: {! L[0]["id"]=parseId(L[.]["id"]) !} modId(L[>]); /*
function with the side effect of modifying laureates */
>>^modId(laureates); /* launch the update on laureates */

>>^>>(".../nobel_laureates_1.dat") laureates %*; /* copy the new version of
laureates into a new file */
- written output to file '/.../nobel_laureates_1.dat'
>>^laureates[_len(laureates)-1]%*; /* display the last laureate in the list */
{
     "id": 937,
     "firstname": "Bob",
     "surname": "Dylan",
     "born": "1941-05-24",
     "died": "0000-00-00",
     "bornCountryCode": "US",
     "gender": "male"
}
>>minMax1*(L,mM) : L==[]? mM: {! mM[0]=L[.]["id"] < mM[0]? L[.]["id"]: mM[0] !}
{! mM[1]=L[.]["id"]>mM[1]? L[.]["id"]: mM[1] !} minMax1(L[>],mM);
>>minMax*(L) : L==[]? []: minMax1(L[>],[L[.]["id"],L[.]["id"]]);
>>mML=minMax*(laureates);
>>^mML;
[ 1, 937 ]
>>range(m,M) : m > M? []: [m|range(m+1,M)];
>>memberL(x,L) : L!=[] && (L[.]["id"]==x || memberL(x,L[>]));
>>missing1(R,L) : R==[]? []: memberL(R[.], L)? missing1(R[>],L):
                                                [R[.]|missing1(R[>],L)] ;
>>missing(L,mM) : missing1(range(mM[0],mM[1]), L);
>>^missing(laureates,mML);
[ 7, 94, 170, 171, 256, 469, 504, 521, 522, 542, 570, 591, 595, 598, 599, 611,
612, 616, 632, 636, 646, 656, 784, 785, 788, 859, 860 ]
>>^_len(ans);
27
>>
```

The function *minMax* computes the minimum and the maximum identifier used in laureates, stored as the two elements of the result list. To this end, it calls *minMax1* that is defined with side effects so that the result list may preserve the structure over all invocations and only the data are updated. The two functions state that ids are in the range from 1 to 937.

In the above session, we also define the function *range*, which constructs the list of all integers into a given range, and the function *memberL*, which tests id membership directly inside a list of laureates. The two functions *missing* and *missing1* are used to compute the ids in the range

from 1 to 937 that are not included in laureates. The are 27 missing ids, as answered by the query `^_len(ans)` – recall that *ans* is a built-in variable storing the last answer to a query (in our case, the list of missing ids).

We now start a new session to merge the data of the list *prizes* stored into "…/nobel_prizes_1.dat" with those of the list *laureates* stored into "…/nobel_laureates_1.dat". The function *findLid* searches for a laureate in *laureates* with a given id. Given a prize, the function *extLaureates* constructs a json for each element in the laureate list for that prize; the json merges data from the two source datasets and is inserted into a list. The function *newPrizeLaureates* scans all prizes and passes each of them to *extLaureates*. The effect of launching *newPrizeLaureates* on the actual arguments *prizes* and *laureates* is to returning a new list of json, each of them collecting a number of meaningful data about the corresponding laureate.

```
/* start a new session */
>>prizes=<<("…/nobel_prizes_1.dat");
 - reading data from file '…/nobel_prizes_1.dat'
 - data are correct
>>laureates=<<("…/nobel_laureates_1.dat");
 - reading data from file '…/nobel_laureates_1.dat'
 - data are correct
>>findLid(id,L) : L==[]? exc("missing key"): id==L[.]["id"]? L[.]:
          findLid(id,L[>]); /* return the json in laureates with a given id */
>>extLaureates(year,cat,PL,L) : /* for each prize, constructs a json for each
element in the laureate list for that prize and include it into a new unique
list */
   <lo,ln> PL==[]? true:
   {! ln={ "id":PL[0]["id"], "year": year, "category": cat,
     "share": PL[0]["share"], "firstname": lo["firstname"],
     "surname": lo["surname"], "born": lo["born"],
     "died": lo["died"], "bornCountryCode": lo["bornCountryCode"],
     "diedCountryCode": lo["diedCountryCode"], "gender": lo["gender"] }
   !}
   [ln | extLaureates(year,cat,PL[>],L)];
>>newPrizeLaureates(P,L) : /*scans all prizes and passes them to extLaureates */
   P==[]? []:
   extLaureates(P[.]["year"], P[.]["category"], P[.]["laureates"],L) +
   newPrizeLaureates(P[>],L);
>>eLaureates = newPrizeLaureates(prizes,laureates); /* new laureate list is
created */
>>^eLaureates[0]%*; /* display the first prize with enriched data */
{
     "id": 928,
     "year": 2016,
     "category": "physics",
     "share": "2",
     "firstname": "David J.",
     "surname": "Thouless",
     "born": "1934-09-21",
     "died": "0000-00-00",
     "bornCountryCode": "GB",
     "diedCountryCode": null,
     "gender": "male"
}

>>^>>("…/nobel_elaureates_1.dat") prizes %*; /* write  enriched laureates into
a text file */
- written output to file '/…/nobel_elaureates_1.dat'
>>
```

The new version of *laureates* is written into a text file in order to preserve it after the end of the session.

In the next session, we first read the file written at the end of the previous session and, then, present some generic functions for asking questions about laureates, in particular for the ones who were born in Europe. To this end, we first list the European country codes in *EuropeCodes* – such data are available in the third file, named "Country", at the link: https://github.com/jdorfman/awesome-json-datasets#nobel-prize. We use as filter condition a list Q of the format [[keyF1, [val$_1$F1, ..., val$_{n_1}$F1]], ... , [keyF*m*, [val$_1$F*m*, ..., val$_{n_m}$F*m*]]]: a laureate L1 is filtered if for each [keyF*i*, [val$_1$F*i*, ..., val$_{n_i}$F*i*]] in Q, P1[keyF*i*] is equal to one of the values in [val$_1$F*i*, ..., val$_{n_i}$F*i*].

```
/* start a new session */
>>elaureates=<<("../nobel_elaureates_1.dat");
 - reading data from file '../nobel_elaureates_1.dat'
 - data are correct
>>EuropeCodes = [  "DE", "AT", "BE", "CZ", "DK", "FI", "FR", "UK", "GR", "HU",
"IS", "IE", "IT", "NO", "PL", "PT", "ES", "SE","CH", "NL", "RS", "LU", "GB",
"BA", "LT", "RO", "BG", "SI", "HR", "BA" ];
>>lFilter1(L1,key,vL) = vL!=[] && ( L1[key]==vL[.] || lFilter1 (L1,key,vL[>]));
/* true if the field "key" for the laureate L1 is equal to one of the values in
vL */
>>laureateFilter(L1,Q) : Q==[] || lFilter1(L1,Q[.][0],Q[.][1]) &&
  laureateFilter(L1,Q[>]);  /*  true  if  the  laureate  L1  satisfies  all  field
conditions listed in Q */
>>laureateGroup1(L,Q) : L==[]? 0: laureateGroup1(L[.],Q)?
 1+laureateGroup1(L[>],Q): laureateGroup1(L[>],Q); /* scan all laureates in L
and count the ones satisfying the conditions in Q */
>>^laureateGroup1(elaureates,[["bornCountryCode",EuropeCodes]);  /*  display  the
number of nobel laureates born in Europe */
441
>>^laureateGroup1(elaureates,[["bornCountryCode",EuropeCodes],
   ["category", ["physics", "chemistry", "medicine"]] ]); /* display the number
of nobel laureates, both born in Europe and with a science prize */
298
>>
```

The query for computing the number of nobel prize winners who were born in Europe is done by passing the filter [["bornCountryCode", EuropeCodes]] while the query that restricts the answer only to the winners of prices in scientific disciplines (physics, chemistry and medicine) uses the filter [["bornCountryCode",EuropeCodes], ["category", ["physics", "chemistry", "medicine"]]].

We next add further conditions on range value that are espressed as a list R of the format [[keyF1, val$_1$F1, val$_2$F1]], ... , [keyF*m*, val$_1$F*m*, val$_2$F*m*]]]: a laureate L1 is filtered if for each [keyF*i*, val$_1$F*i*, val$_2$F*i*] in R, val$_1$F*i* ≤ P1[keyF*i*] ≤ val$_2$F1. For intance, [["year", 2000, 2016]] states that the year of a prize is included in ther range from 2000 to 2016. The range condition is checked by the function *rangeFilter*.

```
/* continue the session */
>>rangeFilter(L1,R) : R==[] || R[.][1] <= L1[R[.][0]] && L1[R[.][0]] <= R[.][2]
       && rangeFilter(L1,R[>]); /* true if the field "R[.][0]" for the laureate
L1 is in the range from R[.][1] to R[.][2] */
>>laureateGroup2(L,Q,R) : L==[]? 0: laureateFilter(L[.],Q) &&
  rangeFilter(L[.],R)?1+laureateGroup2(L[>],Q,R): laureateGroup2(L[>],Q,R);
 /* scan all laureates in L and count the ones satisfying the conditions in both
Q and R */
>>^laureateGroup2(elaureates,[["bornCountryCode",EuropeCodes],
   ["category", ["physics", "chemistry", "medicine"]] ],
```

```
   [ ["year", 2000, 2016] ] ); /* display the number of  nobel laureates, both
born in Europe and with a science prize achieved in the present millenium */
42
>>
```

A further extension is to return a count result on the filtered laureates that is itemized by the values of a given field (e.g., the gender). The field to be used for itemization is to be given in the format [key, [val_1, ..., val_n]] and no null values are expected for that field. The change in the structure of the new query only concerns the counting commands: (1) the initial setting of the summation to zero is to be duplicated for each value of the itemization field by using a json S whose fields correspond to such values and (2) the increment at each step is to applied to the field of S that corresponds to the value hold by the current laureate. In the session below, we use "gender" as itemization field with value "male" and "female".

```
/* continue the session */
>>initM*(S,M1) : M1==[]? S: {! S[M1[.]]=0 !} initM(S,M1[>]); /* initial setting
to zero for the two value summations */
>>laureateGroup3*(L,Q,R,M) : <S> L==[]? initM({}, M[1]):
  laureateFilter(L[.],Q) && rangeFilter(L[.],R)?
  L[.][M[0]]==null? exc("null values not allowed"):
                {! S = laureateGroup3(L[>],Q,R,M! } {! S[L[.][M[0]]]+=1 !} S :
                laureateGroup3(L[>],Q,R,M);
 /* scan all laureates in L and count the ones satisfying the conditions in both
Q and R, itemized by the field described in M */
>>^laureateGroup3(eLaureates,[["bornCountryCode",EuropeCodes],
 ["category", ["physics", "chemistry", "medicine"]] ], [ ["year", 2000, 2016] ],
 ["gender",["male","female"]] ); /* number of  nobel laureates, both born in
Europe and with a science prize achieved in the present millenium, itemized by
gender */
{ "male": 40, "female": 2 }
>>^laureateGroup3(eLaureates,[["bornCountryCode",EuropeCodes],
 ["category", ["physics", "chemistry", "medicine"]] ], [ ],
 ["gender",["male","female"]] ); /* number of  nobel laureates, both born in
Europe and with a science prize achieved, itemized by gender */
{ "male": 289, "female": 9 }
>>^laureateGroup3(eLaureates,[["bornCountryCode",EuropeCodes]], [ ],
 ["gender",["male","female"]] ); /* number of  nobel laureates born in Europe,
itemized by gender */
{ "male": 421, "female": 20 }
>>
```

Before concluding this section, a remark is in order. As the careful reader has probably already noted, the query defined by the function *laureateGroup1* implements a scheme "*filter+map*", where the filtered objects can be actually mapped to a same simple object, followed by a *reduce* phase based on counting the mapped simpled object instances. The function *laureateGroup2* follows the same scheme with the difference that a range condition filter is added. Finally, *laureateGroup3* uses a more complex mapping: to implement itemization, the filtered objects must be mapped to several simple objects, one for each possible value of the itemization field. The reduce phase is more complicated as well: the counting must be separately applied to the diverse types of simple objects.

5.4. Sorting Algorithms

A large number of sort algorithms for object collections are known in the literature. Some of them have a quadratic complexity, i.e., in the average they require a number of operations quadratic in the length of the collection (in our case, list), say n: in the average, these algorithms run in time $\Theta(n^2)$. Quadratic sorting algorithms preserve a historical interest but they are replaced by more efficient algorithms with time complexity $\Theta(n \log n)$. In this section,

we shall present some implementations of both types of algorithms and provide an empirical comparison of their time complexity using *clops* (CalcuList micro OPerationS) as time measure. We test various sorting algorithms on five list of 500 integers, randomly generated in the range from 10 to 10000 as follows:

```
>>randLI(minv, maxv, k) : /* generate a list of k random integers in the range
<minv, maxv> */
     k<=0? []: [round(_rand()*(maxv-minv)+minv)|randLI(minv, maxv, k-1)];
>>RL=randLI(10,10000,500); /* list with 500 random integers in <10, 10000> */
>>^RL;
[ 1664, 1513, 3997, 2053, 2189, 9647, 8123, 7779, 5444, 3051, 7066, 5338, ..., ]
>>
```

A first classical quadratic sort algorithm dealt with is *Bubble Sort*, whose implementation in CalcuList has been already presented in Section 3.4. We next present an implementation of the algorithm that works in-place, i.e., to order a list, it does not construct a new one at each step, but it performs swaps inside the input list itself. As the swap function has side effect, all other functions involved with the bubble sorts will be with side effects as well. The ordering function is generic and passed as an argument of type function. In the query, the ordering will be given by the operator "<=" and defined as a lambda function

```
>>swap*(L,i,j) : <tmp> {! tmp=L[i] !} {! L[i]=L[j] !} {! L[j]=tmp !} true; /*
swap two list elements */
>>bs_pass*(L,o/2, i, last,lsInd) : i>last? lsInd:
         !o(L[0],L[1])? {! & swap(L, 0, 1) !} bs_pass(L[>], o, i+1, last,i):
         bs_pass(L[>], o, i+1, last,lsInd); /* scan the list and each two
consecutive elements that do not satisfy the ordering o are swapped and the
index lsIndex of the last swapped element is updated */
>>bs_iter*(L, o/2, last) : <lsInd> last<0? L:
     {! lsInd=bs_pass(L, o, 0, last,0) !} bs_iter(L, o, lsInd-1); /* scan the
list and each two consecutive elements that do not satisfy the ordering o are
swapped */
>>bubbleSort*(L,o/2) : L==[]? L: bs_iter(L, o,_len(L)-2);
>>L= RL[:]; /* clone the input for a subsequent test */
>>L=bubbleSort(L,lambda x,y: x<=y); /* run bubble sort */
>>!clops;
120186484
>>
```

The bubble sorting is performed by executing n-1 passes, where n is the length of the list L: each pass consists of scanning a portion of the list and performing the swap of two consecutive elements if the first of them follows the other in the order defined by *o*. Each pass is implemented by the function *bs_pass*. The first pass starts from the element with index 0 to the element with index *last* = n-2 so that the element with index n-1 in the list is eventually the greatest one; the second pass starts from the element with index 0 to the element with index *last* = n-3, which becomes the second greater element and so on. The function *bs_pass* is made more efficient by extracting the tail of the list at each step of possible swap so that the first two list elements in the list are compared. The iteration on the n-1 passes is performed by the function *bs_iter*. Finally, the function *bubbleSort* activates the overall procedure, using the function `_len(L)`" to compute the length of the list L. As we have already pointed out in Section 3.4, our CalcuList implementation does not narrowly follow the typical scheme that is particularly targeted for data structures whose element can be accessed in constant time. As a list element in a list with index i requires CalcuList to scan the previous i-1 elements, our functions try to reduce as much as possible offsets while passing a list.

Another classic quadratic sorting algoritm is *Selection Sort*, which divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list.

Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on the order function o) element in the unsorted sublist, swapping it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

We first provide a CalcuList implementation of Selection Sort that strictly follows the algorithm scheme for direct access data structures. We add a postfix "1" to such an implementation to distinguish it from a more refined one, presented next.

```
>>SELSORT: jmin; /* labeled global variable used by selection sort */
>>sel_min1*(L,o/2,vmin,j,n) : /* scan the list to search an element smaller than
the one with index jmin and with value vmin */
    <SELSORT*> j==n? true:
    vmin <= L[j]? sel_min1(L,o,vmin,j+1,n):
    {! jmin= j !} sel_min1(L,o,L[j],j+1,n);
>>sel_Sort1*(L,o/2,i,n) : /* for i = 0 to n-2, find the index jmin of the
minimum element from the position i on and locate it in the ordered position  */
   <SELSORT*> i==n-1? L:
   {! jmin= i !} {! &sel_min1(L,o,L[i],i+1,n) !}
   {! &jmin!=i? swap(L,i,jmin):true} sel_Sort1(L,o,i+1,n);
>>selectionSort1*(L,o/2) : L==[]?[]: sel_Sort1(L,o,0,_len(L));
>>L= RL[:]; /* clone the input for a subsequent test */
>>L=selectionSort1(L,lambda x,y: x<=y); /* run preliminary version of selection
sort */
>>!clops;
1099575659
>>
```

The sorting is performed by invoking *n*-1 times the function *sel_Sort1*, where *n* is the length of the list L. At each pass *i* (from 0 to *n*-1), the portion of the list L from *i* to *n* is scanned by the function *sel_min1* to find the *i*-th smaller (or larger, depending on the the ordering o) element and put it in the sorted position.

The number of clops for this version of Selection Sort is very high in comparison to the ones for Bubble Sort. A version of Selection Sort that is better suited to CalcuList's architecture is defined next. The main differences are in defining indices as offsets w.r.t. the begin of a current sublist rather than the the whole list.

```
>>sel_min*(o/2,L1,vmin,j) = /* index j of an element E is now used only to store
its position in case it is the smallest and not to access E */
    <SELSORT*> L1==[]? true: o(vmin, L1[.])?
     sel_min(o,L1[>], vmin,j+1):
     {! jmin= j} sel_min(o,L1[>],L1[.],j+1);
>>sel_Sort*(L,o/2,L1) = /* an index now starts from the begin of the sublist to
reduce offset sizes */
   <SELSORT*> L1[>]==[]? L:
   {! jmin= 0} {! &sel_min(o,L1[>],L1[.],1)} {! &jmin!=0? swap(L1,0,jmin):true}
   sel_Sort(L,o,L1[>]);
>>selectionSort*(L,o/2)= L==[]?[]: sel_Sort(L,o,L);
>>L= RL[:];
>>L=selectionSort(L,lambda x,y: x<=y); /* run improved version of selection sort
*/
>>!clops;
67215077
>>
```

The performance of Selection Sort very much improves and overtakes the one of Bubble Sort.

A third classical quadratic sort algorithm is *Insertion Sort* that scans the input list L and includes each element of it into the correct position of the ordered list O, that is eventually delivered. Each iteration, insertion sort removes one element from L, finds the location it

belongs within O, and inserts it there. It repeats until no input elements remain. Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

An efficient in-place insertion requires the scan to proceed from the bottom to the top, so that position shifting can be immediately done. This backward approach cannot be efficiently implemented in CalcuList. Therefore, it is convenient to construct the ordered list into a new structure, following the classical schemes of functional programming.

```
>>add(O1,o/2,O,E) = /* the element E is inserted into the correct position of
the output ordered list O */
   O==[]? O1+[E]: o(E,O[.])? O1+[E]+O: add(O1+[O[.]],o, O[>],E);
>>ins(L,o/2,O) = /* each element E is taken from the input list L and inserted
in the output list O  */
   L==[]? O: ins(L[>],o,add([],o,O,L[.]));
>>insertionSort(L,o/2) = L==[]? [] : ins(L[>],o,[L[.]]);
>>L= RL[:];
>>L=insertionSort(L,lambda x,y: x<=y); /* run of insertion sort */
>>!clops;
74622546
>>
```

The number of clops executed confirms that our implementation of Insertion Sort is not far from the improved version of Selection Sort in terms of efficiency.

Let us now move to the implementation of optimal sorting algorithms, which have an average complexity of $\Theta(n \log n)$, where n is the length of the list. A well-known optimal sorting algorithm is *Merge Sort*, that divides a list into two halves, that in turn are recursively ordered and then merged, i.e., combined into a unique list by preserving the overall ordering. To implement the algorithm we use the function *listMerge* that has been already defined in Section 3.1 for merging two ordered lists. For the sake of readability, we rewrite the function by renaming it *merge*.

```
>>merge(O1,O2,o/2) = /* merge of two ordered lists */
     O1==[]? O2[:]: O2==[]? O1[:]:
     o(O1[.],O2[.])? [O1[.] | merge(O1[>],O2,o)] : [O2[.] | merge(O1,O2[>],o)];
>>ms(L,o/2,n) = /* divide the list elements into two sublists of equalę size,
recursively order them and then merge the results */
      <n2> {! n2=n//2} n<=1? L[:]:
           merge(ms(L[:n2],n2,o), ms(L[n2:], n-n2,o));
>>mergeSort(L,o/2) = L==[]||L[>]==[]? L[:]: ms(L,_len(L));
>>L= RL[:];
>>L=mergeSort(L,lambda x,y: x<=y); /* run of merge sort */
>>!clops;
2345265
>>
```

The number of clops is much lower now, on coherence with the complexity theory results.

We conclude the section by implementing *Quick Sort*, which is another well-known optimal sorting algorithm. Let x be the list head and T be the list tail. Then we partition T into the list T0 of the elements y of T such that $y \leq x$ (or larger, depending on the the ordering o) and the list T1 of all other elements; then, recursively sort T0 and T1 and finally return the concatenation of the two resulting ordered lists, separated by the element x.

```
>>partition(x,T,T0,T1,o/2) = /* partition T in the two sublists T0, with
elements smaller than x, and T1, with all other elements of T */
```

```
     L==[]? [T0,T1] : o(L[.],x)?
     partition(x,L[>],[L[.]|T0],T1,o): partition(x,L[>],T0, [L[.]|T1],o);
>>quickSort(L,o/2)= /* partition the tail T of L into two sublists T0 and T1 and
recursively perform quicksort on them, leaving the head of L between the two
ordered sublists */
    <T01> L==[]||L[>]==[]?L[:]:
    {! T01= partition(L[.],L[>],[],[],o) }
    quickSort(T01[0],o)+[L[.] | quickSort(T01[1],o)];
>>L= RL[:];
>>L=quickSort(L,lambda x,y: x<=y); /* run of quick sort */
>>!clops;
3493111
>>checkOrder(L,o/2) = /* check whether the list L is ordered according to o */
>>^checkOrder(L,lambda x,y: x<=y);
true
>>
```

The number of clops is here slightly lower than for merge sort. The function *checkOrder* at the end of the above session check whether or not a list is ordered – before looking for an efficient sorting algorithm implementation, it may be useful to have a tool to verificy the semantic correctness.

To have a better understanting of the actual performance of CalcuList implementation of the sorting algorithms, we re-run computations for a second list of 500 random elements and for additional two lists of 250 elements. The number of clops executed by the six algorithms on the 4 lists are shown in the table below:

	SIZE 500			SIZE 250			
	500/1	500/2	AV (500)	250/1	250/2	AV (250)	AV(500) / AV(250)
BS	120,186,484	124,230,272	122,208,378	30,300,161	30,470,763	30,385,462	4.021935819
SS1	1,099,575,659	1,101,931,910	1,100,753,785	149,930,354	149,796,050	149,863,202	7.345057158
SS	67,174,140	67,444,254	67,309,197	16,992,716	16,968,419	16,980,568	3.963895612
IS	74,622,546	69,280,412	71,951,479	14,084,970	14,398,651	14,241,811	5.052130065
MS	2,345,265	3,392,201	2,868,733	1,525,591	1,518,045	1,521,818	1.8850697
QS	3,493,111	3,031,530	3,262,321	1,515,918	1,495,095	1,505,507	2.16692555

As the input size is doubled, the number of clops grows of around 4 times for algorithms BS (Bubble Sort) and SS (Selection Sort): thus, our CalcuList implementations compensate possible inefficiencies caused by the lack of a direct access to a list element and preserve the quadratic nature of the algorithms. On the other side, as expected, the performance of SS1 (simplified version of Selection Sort) is rather poor as it fully pays the limitations of CalcuList architecture. Also the performance of IS (Insertion Sort) is poor w.r.t. BS and SS, but it is not as bad as for SS1. Finally, both MS (Merge Sort) and QS (Quick Sort) show an actual run time behavior coherent with their complexity of $\Theta(n \log n)$.

5.5. Vectors and Matrices

A vector V of size (length) m in CalcuList is typically represented by a list with n elements, which may be addressed as V[0], ..., V[m-1] – recall that direct access to a list element is not supported by CalcuList and, therefore, the cost of accessing an element is in the average linear in m.

A matrix M of size $n \times m$ is represented by a list of n lists (*rows*) of m-sized lists (*columns*). The elements of M are addressed as M[0][0], ..., M[n-1][n-1] and the rows by M[0], ..., M[n-1]. The cost of accessing an element is in the average quadratic ($n \times m$).

In the following, we show how to construct V and M in two ways: (1) by assigning a same value *x* to all elements and (2) by generating random values for them.

```
>> newVx_1(m,j,x) : j >=m? []: [x | newVx_1(m,j+1,x)];
>> newVx(m,x) : /* create a vector V with m elements equal to x */
   m <= 0? []: newVx_1(m,0,x);
>> newVr_1(m,j) : j >=m? []: [_rand() | newVr_1(m,j+1)];
>> newVr(m) : /* create a vector V with m elements randomly generated */
   m <= 0? []: newVr_1(m,0);
>> newMx_1(n,i,m,x) : i>= n? []: [newVx(m,x) | newMx_1(n,i+1,m,x)];
>> newMx(n,m,x) : /* create a matrix M with n x m elements equal to x */
   n<=0? []: newMx_1(n,0,m,x);
>> newMr_1(n,i,m) : i>= n? []: [newVr(m) | newMr_1(n,i+1,m)];
>> newMr(n,m) : /* create a vector V with m elements randomly generated */
   n<=0? []: newMr_1(n,0,m);
>> eye_2(n,i,j) : j >=n? []: i==j? [1 | eye_2(n,i,j+1)]: [0 | eyex_2(n,i,j+1)];
>> eye_1(n,i) : i>= n? []: [eye_2(n,i,0) | eye_1(n,i+1)];
>> eye(n) : /* create identity matrix with 1 in the diagonal and 0 elsewhere */
   n<=0? []: eye_1(n,0);
>>
```

Given *n*, the function *eye,* defined above, constructs the *identity matrix* of size *n*, i.e., the $n \times n$ square matrix with ones on the main diagonal and zeros elsewhere.

The function *swap,* defined below, exchanges the values of two list (or vector) elements. It can be also used to swap two rows (lists) of a matrix, as a matrix is a list of rows. However, swapping two matrix columns needs an ad-hoc function *swapCol*. A function similar to *swapCol* is *colM* that returns a deep copy of a matrix column – note that, because of matrix data structure, a shallow copy cannot be obtained. The function *colM* is used to define the function *transpose*, which computes the transpose M^T of a given matrix M, i.e., every column of M becomes a row of M^T.

```
>> swap*(V,i,j) : /* swap two vector (list) elements and return true */
   <tmp> {! tmp=V[i] !} {! V[i]=V[j] !} {! V[j]=tmp !} true;
>> swapCol*(M,j1,j2) : /* exchange  columns j1 and j2 of the matrix M and return
true */
   M==[]? true: swap(M[0],j1,j2) && swapCol(M[>],j1,j2);
>> colM(M,j) : /* return a deep copy of the column j of the matrix  M */
   M==[]? []: [M[0][j] | colM(M[>],j)];
>> transpose_1(M,C,j) : C==[]? []: [colM(M,j) | transpose_1(M,C[>],j+1)];
>> transpose(M) : /* return the transpose of a matrix M */
   M==[]? []: transpose_1(M,M[0],0);
>> C= [ [1, -4], [2,-8], [-11,12] ];
>> ^C %>; /* "%>" print option for matrices */
[    [ 1, -4 ],
     [ 2, -8 ],
     [ -11, 12 ]
]
>> ^colM(C,1); /* display column 1 of C */
[ -4, -8, 12 ]
>> CT=transpose(C);
>> ^CT %>;
[    [ 1, 2, -11 ],
     [ -4, -8, 12 ]
]
>> ^swapCol(C,0,1);
true
>> ^C %>;
[    [ -4, 1 ],
     [ -8, 2 ],
     [ 12, -11 ]
]
```

```
>>
```

Important operations for both vectors and matrices are multiplications:

- given two vectors V1 and V2 that are conform (i.e., they have the same length, say m), the product V1 × V2 is equal to the scalar $\Sigma_{i=0,m-1}$V1[i]×V2[i], and
- given two matrices M1, say with size $n \times p_1$, and M2, say with size $p_2 \times m$, that are conform (i.e., $p = p_1 = p_2$), the product M1 × M2 is equal to the $n \times m$ matrix M such that, for each i = 0, …, n-1 and each j = 0, …, m-1, M[i][j] = $\Sigma_{k=0,p-1}$M1[i][p]×V2][p][j],.

The two operations are defined next.

```
>> vectorMult(V1,V2) : /* multiplication of two vectors V1 and V2 conform */
   V1==[] && V2==[]? V1==[] || V2==[]? 0:
   exc("two vectors non conform for sum"):
   V1[0]*V2[0]+vectorMult(V1[>],V2[>]);
>> mMult_2*(A,BT,CR) : CR==[]? null:
   {! CR[0]=vectorMult(A[0],BT[0]) !} mMult_2(A,BT[>],CR[>]);
>> mMult_1*(A,BT,C,CC) :  C==[]? CC:
   {! &mMult_2(A,BT,C[.]) !} mMult_1(A[>],BT,C[>],CC);
>> matrixMult*(A,B) : /* multiplication of two matrices A (nxk) and B (kxm) */
   <C>{! C=newMx(_len(A),_len(B[0]),0) !}
  _len(A[0])!=_len(B)? exc("two vectors non conformant to sum"):
  mMult_1(A,transpose(B),C,C);
>> ^CT %>;
[     [ 1, 2, -11 ],
      [ -4, -8, 12 ]
]
>> ^vectorMult(CT[0],CT[1]); /* vector multiplication =
   1*(-4)+2*(-8)+(-11)*12 = -4-16-132 = -152 */
-152
>> ^C %>; /* recall C has been updated */
[     [ -4, 1 ],
      [ -8, 2 ],
      [ 12, -11 ]
]
>> ^matrixMult(C,CT) %>; /* matrix multiplication – C ha size 3x2, CT has size
2x3, the result has size 3x3 */
[     [ -8, -16, 56 ],
      [ -16, -32, 112 ],
      [ 56, 112, -264 ]
]
>>
```

A square matrix is called *lower triangular* if all the entries above the main diagonal are zero. Similarly, a square matrix is called *upper triangular* if all the entries below the main diagonal are zero. A matrix that is both upper and lower triangular is called a *diagonal matrix*. As examples, considers the matrices defined next: A is upper triangular, AT is lower triangular and D is diagonal.

```
>> A=[ [ 2, 3, -1 ], [ 0, 1, -1 ], [ 0, 0, 2 ]; /* upper matrix */
>> ^A %>;
[     [ 2, 3, -1 ],
      [ 0, 1, -1 ],
      [ 0, 0, 2 ]
]
>> AT=transpose(A); /* lower matrix */
>> ^AT %>;
[     [ 2, 0, 0 ],
      [ 3, 1, 0 ],
      [ -1, -1, 2 ]
]
```

```
>> D=[ [ -1, 0, 0 ], [0, 2, 0], [0, 0, 3] ]; /* diagonal matrix */
>> ^D %>;
[    [ -1, 0, 0 ],
     [ 0, 2, 0 ],
     [ 0, 0, 3 ]
]
>> isUpperTriangM_2(V,j) : j<0? true: V[.]==0 && isUpperTriangM_2(V[>],j-1);
>> isUpperTriangM_1(M,i) : M==[] ||
                   isUpperTriangM_2(M[.],i-1) && isUpperTriangM_1(M[>],i+1);
>> isUpperTriangM(M) : /* true if all elements of M below the diagonal are 0*/
                       isUpperTriangM_1(M,0);
>> ^isUpperTriangM(A);
true
>> ^isUpperTriangM(AT);
false
>> ^isUpperTriangM(D);
true
>> isLowerTriangM_2(V) : V==[]? true: V[.]==0 && isLowerTriangM_2(V[>]);
>> isLowerTriangM_1(M,i) : M==[] ||
            isLowerTriangM_2(M[.][>i]) && isLowerTriangM_1(M[>],i+1);
>> isLowerTriangM(M) : /* true if all elements of M above the diagonal are 0*/
                       isLowerTriangM_1(M,0);
>> ^isLowerTriangM(A);
false
>> ^isLowerTriangM(AT);
true
>> ^isLowerTriangM(D);
true
>> isDiagonalM(M) = /* true if it is both a upper and a lower matrix*/
                       isUpperTriangM(M) && isLowerTriangM(M);
>> ^isDiagonalM(A);
false
>> ^isDiagonalM(AT);
false
>> ^isDiagonalM(D);
true
>>
```

Observe that L[>n] corresponds to apply n+1 times the suffix operator [>]to the list L.

Many operations on matrices are simpler in case they are triangular. For instance, the determinant is simply the product of all elements in the main diagonal. Also an important problem such as solving a linear equation system A × X = B, where A is a triangular matrix of size $n \times n$, B is a vector of size n and X is a vector of n unknowns, can be done in simple and efficient way (quadratic time). We recall that solving a linear equation systems consists in finding n values for X such that, for each i = 0, ..., n-1:

$$\Sigma_{0 <= j < n} A[i][j] \times X[j] = B[j]$$

In the example below, we present a function for solving a linear equation systems for the case A is a upper triangular matrix. Note that a necessary and sufficient condition to have finite solutions is that the diagonal does not contain zeros.

To define the function *triangLinearSystem* on a triangular matrix of size n, we start computing X[n-1] = B[n-1]/A[n-1][n-1]. Then we continue by decreasing the index i down to 0 so that, at a generic step i, X[i] = (B[i] – $\Sigma_{i<j<n}$ A[i][j] × X[j]) / A[i][i]. Note that if the function call *noZeroInDiagonal*(A) returns *true* then no division by zero will occur. The complexity of this algorithm is $\Theta(n^2)$, i.e., quadratic in the size of the matrix.

```
>> triangDet_1(M,i) : M==[]? 1: M[.][i]*triangDet_1(M[>],i+1);
>> triangDeterminant(M) : /* product of all diagonal elements of a Matrix M */
```

```
                          triangDet_1(M,0);
>> ^triangDeterminant(A);
4
>> ^triangDeterminant(AT);
4
>> ^triangDeterminant(D);
-6
>> noZeroInDiagonal_1(M,i) = M==[] ||
                                  M[.][i]!=0 && noZeroInDiagonal_1(M[>],i+1);
>> noZeroInDiagonal(M) = /* true if no diagonal element of a matrix M is 0 */
                        noZeroInDiagonal_1(M,0);
>> ^noZeroInDiagonal(A);
true
>> ^noZeroInDiagonal(AT);
true
>> ^noZeroInDiagonal(D);
true
>> triangLinearSystem_sumA(AT,X,i,i1,n1) : i1>n1? 0:
                     AT[i1][i]*X[i1]+triangLinearSystem_sumA(AT,X,i,i1+1,n1);
>> triangLinearSystem_1*(AT,B,X,i,n1) : i<0? X:
             {! X[i]=(B[i]-triangLinearSystem_sumA(AT,X,i,i+1,n1))/AT[i][i] !}
             triangLinearSystem_1(AT,B,X,i-1,n1);
>> triangLinearSystem*(A,B) : /* solve a linear equation system A*X = B
                                where A is an upper triangular matrix */
        <n>{! n=_len(A) !} n!=_len(A[0]) || n!= _len(B)?
        exc("matrix A and vector B are not conformant"):
        !noZeroInDiagonal(A)? exc("matrix A singular"):
        triangLinearSystem_1(transpose(A),B,newVx(n,0),n-1,n-1);
>> B= [2,1,-2];
>> ^triangLinearSystem(A,B);
[ 0.5, 0.0, -1.0 ]
>> ^triangLinearSystem(D,B);
[ -2.0, 0.5, -0.6666666666666666 ]
>> diagonalLinearSystem_1(A,B,i) : A==[]? []:
                 [ B[.]/A[.][i] | diagonalLinearSystem_1(A[>],B[>],i+1) ];
>> diagonalLinearSystem(A,B) : /* solve a linear equation system A*X = B
                where A is diagonal matrix */
       <n>{! n=_len(A)} n!=_len(A[0]) || n!= _len(B)?
        exc("matrix A and vector B are not conformant"):
        !noZeroInDiagonal(A)? exc("matrix A singular"):
        diagonalLinearSystem_1(A,B,0);
>> ^diagonalLinearSystem(D,B);
[ -2.0, 0.5, -0.6666666666666666 ]
>>
```

The resolution for the case of a diagonal matrix is even simpler: for each index i, $0 \le j < n$, it is sufficient to compute X[i] = B[i] / A[i][i]. This algorithm is linear in n.

Let us now solve a linear equation system for a general case of matrix. We implement *Gaussian elimination* (also known as *row reduction*), which perform a sequence of three elementary row operations to modify the matrix A together with the vector B (*augmented matrix*) until the lower left-hand corner of the matrix is filled with zeros, as much as possible: 1) swapping two rows, 2) multiplying a row by a non-zero number, 3) adding a multiple of one row to another row.

Using row operations, a matrix can always be transformed into an upper triangular matrix in row echelon form, i.e., (*i*) all nonzero rows (rows with at least one nonzero element) are above any rows of all zeroes (all zero rows, if any, belong at the bottom of the matrix), and (*ii*) the *leading coefficient* (the first nonzero number from the left, also called the *pivot*) of a nonzero row is always strictly to the right of the leading coefficient of the row above it. Some variants of the row reduction algorithms prescribe to continue the trasformations until a

reduced format is achieved where all pivots are equal to one. We instead stop as soon as we get an upper triangular matrix: at this point we may invoke the function *triangLinearSystem* to compute the unknowns. Note that our early stop has as consequence that the row operation of multiplying a row by a non-zero number is never performed so that only two operators are used: swapping two rows and adding a multiple of one row to another row.

The approach used in the implementation of *linearSystem* proceeds as follows: at each step i, before performing the row operations induced by the row i, it searches for the row i_1 below i with maximum absolute value in the column i and eventually swappes the row i and i_1. This search mitigazes the possibility of introducing approximation errors due to a division by a value close to zero.

As an example of row operations are shown in the table below:

	System of Linear equations					Row Operations	Augmented Matrix			
R1.1	2 x	+ y	- z	=	8		2	1	-1	8
R1.2	-3 x	- y	+ 2z	=	-11		-3	-1	2	-11
R1.3	-2 x	+ y	+ 2z	=	-3		-2	1	2	-3
R2.1	-3 x	- y	+ 2z	=	-11	R2.1 = R1.2	-3	-1	2	-11
R2.2	2 x	+ y	- z	=	8	R2.2 = R1.1	2	1	-1	8
R2.3	-2 x	+ y	+ 2z	=	-3		-2	1	2	-3
R3.1	-3 x	- y	+ 2z	=	-11		-3	-1	2	-11
R3.2		1/3 y	+ 1/3 z	=	2/3	R3.2 = R2.2 + 2/3 R2.1	0	1/3	1/3	2/3
R3.3		5/3 y	2/3 z	=	13/3	R3.3 = R2.3 - 2/3 R2.1	0	5/3	2/3	13/3
R4.1	-3 x	- y	+ 2z	=	-11		-3	-1	2	-11
R4.2		5/3 y	2/3 z	=	13/3	R4.2 = R3.3	0	5/3	2/3	13/3
R4.3		1/3 y	+ 1/3 z	=	2/3	R4.3 = R3.2	0	1/3	1/3	2/3
R5.1	-3 x	- y	+ 2z	=	-11		-3	-1	2	-11
R5.2		5/3 y	2/3 z	=	13/3		0	5/3	2/3	13/3
R5.3			1/5 z	=	-1/5	R5.3 = R4.3 - 1/5 R4.2	0	0	1/5	-1/5

The overall complexity of performing the row operations is $\Theta(n^3)$, i.e., cubic in the size of the matrix. Once we have got the upper triangular augmented matrix (at the bottom left corner), we call the function *triangLinearSystem* to complete the resolution. We have shown that he latter phase has a quadratic complexity; hence the overall complexity is $\Theta(n^3)$, i.e., cubic in the size of the matrix. It turns out that achieving triangulization is the hardest part of the overall algoritm.

The implementation of Gauss elimination is presented next. Preliminarly, we introduce two technical functions: the first one return the absolute value of a variable and the second one returns the index of an element in a column with the highest absolute value.

```
>> abs(x) : /* return the absolute value */
            x>=0? x: -x;
>> jMaxV_1(xmax,V,i,imax) : V==[]? imax:
                            abs(V[0])>abs(xmax)? jMaxV_1(V[0],V[>],i+1,i):
                                                 jMaxV_1(xmax,V[>],i+1,imax);4
```

```
>> jMaxV(V) : /* return the index of the element of a vector V with maximal
absolute value */
              V==[]? -1: jMaxV_1(V[0],V[>],1,0);
>> linearSystem_triang1*(vi_aii,ATCi,V) : V==[]?true:
       {! V[0]=V[.]-ATCi[.]*vi_aii !} linearSystem_triang1(vi_aii,ATCi[>],V[>]);
>> linearSystem_triang*(aii,ATCi,ATC,i) : ATC==[]?true:
                       linearSystem_triang1(ATC[0][i]/aii,ATCi,ATC[.][>i]) &&
                       linearSystem_triang(aii,ATCi,ATC[>],i);
>> linearSystem1*(AT,B,ATC,BC,i,n) : <imax> ATC[>]==[]?
      ATC[.][i]==0? exc("singular matrix"):
      triangLinearSystem_1(AT,B,newVx(n,0),n-1,n-1):
      {! imax=jMaxV_1(ATC[.][i],ATC[.][>i],i+1,i) !}
      {! &imax==i? null: swap(B,i,imax) && swapCol(ATC,i,imax) !}
      ATC[.][i]==0? exc("singular matrix"):
      {! &linearSystem_triang1(BC[.]/ATC[.][i],ATC[.][>i],BC[>]) &&
      linearSystem_triang(ATC[.][i],ATC[.][>i],ATC[>],i) !}
      linearSystem1(AT,B,ATC[>],BC[>],i+1,n);pHeads(ATC[.],i),ATC[>],i) !}
      linearSystem1(AT,B,ATC[>],BC[>],i+1,n);
>> linearSystem*(A,B) : /* solve a linear equation system */
    <n,AT,BB> {! n=_len(A) !}
        n==0 || n != _len(A[0]) || n != _len(B)?
        exc("matrix A and vector B are not conformant"):
        {! AT=transpose(A) !} {! BB=B[:] !} linearSystem1(AT,BB,AT,BB,0,n);
>> A= [ [2, 1,-1], [-3, -1, 2], [-2, 1, 2] ];
>> B= [8, -11,-3];
>> X=linearSystem(A,B);
>> ^X;
[ 2.0, 3.0000000000000004, -0.9999999999999999 ]
>> ^transpose(matrixMult(A,transpose([X])));
[ [ 8.0, -11.0, -2.999999999999999 ] ]
>>
```

The results present some minor approximation errors: less than 10^{-16}. To verify that the algorithm is indeed correct, we may compute the product A × X: the result should be equal to B. We have performed this test but paying some attention to the fact that the function *matrixMult* assumes that also the second operand is a matrix and not a vector. We then constructed a matrix with X as unique row, transposed this new matrix so that it gets size $n \times 1$ and becomes conform to the first operand A, invoked *matrixMult* obtaining a matrix $n \times 1$ and transposed the result to eventually get (except for the approximation errors) the vector B.

Gauss elimination can be easily extended to compute the determinant of any matrix. We inizialize the variable *signDet* (sign of determinant) to 1 and proceed to apply two row operators (swapping two rows and adding a multiple of one row to another row) to achieve the upper triangular format. Any time swapping is performed, the determinant sign is changed, i.e., *signDet* = –*signDet*. Once we get the upper triangular format, the matrix determinant is obtained by multiplying the upper triangular matrix determinant by *signDet*. As the row operation of multiplying a row by a non-zero scalar is never performed in our approach, we do not need to store intermediate scalars.

```
>> det1*(AT,ATC,i,n,signDet) : <imax> ATC[>]==[]?
           ATC[.][i]==0? 0:
           signDet*triangDeterminant(AT):
           {! imax=jMaxV_1(ATC[.][i],ATC[.][>i],i+1,i) !}
           {! & imax==i? null: swapCol(ATC,i,imax) !}
           {! signDet= imax==i? signDet: -signDet !}
           ATC[.][i]==0? 0:
           {! &solveLinSys_triang(ATC[.][i],ATC[.][>i],ATC[>],i) !}
           det1(AT,ATC[>],i+1,n,signDet);
>> det*(A) : /* compute the determinant of a square matrix n x n */
           <n,AT>  {! n=_len(A) !}
           n==0 || n != _len(A[0])? exc("matrix non conform"):
```

```
            {! AT=transpose(A) !} det1(AT,AT,0,n,1);
>> ^A %>;
[    [ 2, 1, -1 ],
     [ -3, -1, 2 ],
     [ -2, 1, 2 ]
]
>> ^det(A);
-0.9999999999999993
>> M= [ [1, 2, -3], [2, 4, -6], [-2,-4,6] ];
>> ^det(M);
0
>>
```

The actual value of the determinant is -1 but some approximation error must be taken into account. Notice that, as we have shown step by step, the matrix A is triangulized after two row swaps: hence, *signDet* is equal to 1 and the sign of the determinant depends on the value of the triangularized matrix of A; as the diagonal of this triangularized matrix is [-3, 5/3, 1/5], the determinant is $-3 \times 5/3 \times 1/5 = -1$. Note that the upper triangular matrix obtained for M after applying Gauss elimination is [[1, 2, 3], [0, 0, 0], [0, 0, 0]] and, therefore, the determinant of M is 0. The complexity of the *det* algoritm is $\Theta(n^3)$, i.e., cubic in the size of the matrix, and triangulization continues to be the hardest part of the overall algoritm.

Another important matrix operation is the *rank* defined as follows. Given a matrix A, let T be the apper triangular matrix obtained from A by applying row operators. The rank of A is the number of non-zero rows in T. If A is not singular then its rank is equal to its size *n*; otherwise, it is equal to some value in the range from 0 to *n*-1.

```
>> rank_1*(ATC,i,n,rankMat) : <imax>
                 ATC[>]==[]? ATC[.][i]==0? rankMat-1: rankMat:
                 {! imax=jMaxV_1(ATC[.][i], ATC[.][>i],i+1,i) !}
                 {! &imax==i? null: swapCol(ATC,i,imax) !}
                 ATC[.][i]==0? rank_1(ATC[>],i+1,n,rankMat-1):
                 {! &solveLinSys_triang(ATC[.][i],ATC[.][>i],ATC[>],i) !}
                 rank_1(ATC[>],i+1,n,rankMat);
>> rank*(A) = /* return the rank of a square matrix A */
            <n> {! n=_len(A)} n==0 || n != _len(A[0])? exc("non square matrix"):
            rank_1(transpose(A),0,n,n);
>> ^rank(A);
3
>> ^rank(M);
1
>>
```

Note that, after applying Gauss elimination, the upper triangular matrix for M is [[1, 2, 3], [0, 0, 0], [0, 0, 0]] and, therefore, the rank of M is 1. As also this algorithm is based on triangularization, its complexity is $\Theta(n^3)$, i.e., cubic in the size of the matrix.

We conclude the section by describing a function to compute the inverse of a generic square matrix M, say of size *n*. Again, we perform Gauss elimination as for the resolution of a linear system, but this time the augmented matrix is obtained by adding to M a block with the identity matrix I_n rather than with the known vector B as for the resolution of a linear equation system. The procedure is illustrate with our running example:

Row Operations	**Augmented Matrix**						
	M			**I_3**			
	2	1	-1	1	0	0	R1.1
	-3	-1	2	0	1	0	R1.2
	-2	1	2	0	0	1	R1.3

R2.1 = R1.2	-3	-1	2	0	1	0	R2.1
R2.2 = R1.1	2	1	-1	1	0	0	R2.2
	-2	1	2	0	0	1	R2.3
	-3	-1	2	0	1	0	R3.1
R3.2 = R2.2 + 2/3 R2.1	0	1/3	1/3	1	2/3	0	R3.2
R3.3 = R2.3 - 2/3 R2.1	0	5/3	2/3	0	-2/3	1	R3.3
	-3	-1	2	0	1	0	R4.1
R4.2 = R3.3	0	5/3	2/3	0	-2/3	1	R4.2
R4.3 = R3.2	0	1/3	1/3	1	2/3	0	R4.3
	-3	-1	2	1	q	0	R5.1
	0	5/3	2/3	0	-2/3	1	R5.2
R5.3 = R4.3 - 1/5 R4.2	0	0	1/5	1	4/5	-1/5	R5.3

In a sense, we solve *n* systems at once, one for each column of I_n – in the example the three known B vectors are: [1, 0, 1], [[0, -2/3, 4/5] and [0, 1, -1/5]. The results of the *n* systems are the columns of the matrix inverse so that the inverse can be obtained by transposition. The algorithm is shown next.

```
>> inv_triangLS_1*(AT,AIT,n1) : AIT==[]? []:
   [ triangLinearSystem_1(AT,AIT[.],newVx(n1+1,0),n1,n1) |
     inv_triangLS_1(AT,AIT[>],n1) ];
>> inv_LS_triang*(aii,ATCi,AITC,i) : AITC==[]?true:
           linearSystem_triang1(AITC[0][i]/aii,ATCi,AITC[.][>i]) &&
           linearSystem_triang(aii,ATCi,AITC[>],i);
>> inv1*(AT,ATC,AIT,AITC,i,n) : <imax>
           ATC[>]==[]?
           ATC[.][i]==0? exc("singular matrix"):
           inv_triangLS_1(AT,AIT,n-1):
           {! imax=jMaxV_1(ATC[.][i],ATC[.][>i],i+1,i) !}
           {! &imax==i? null: swapCol(AIT,i,imax) && swapCol(ATC,i,imax) !}
           ATC[.][i]==0? exc("singular matrix"):
           {! &inv_LS_triang(ATC[.][i],ATC[.][>i],AITC,i) &&
           linearSystem_triang(ATC[.][i],ATC[.][>i],ATC[>],i) !}
           inv1(AT,ATC[>],AIT,AITC,i+1,n);
>> inv*(A) : /* return the inverse of a square, non-singular matrix A */
  <n,AT,I>  {! n=_len(A)}
           n==0 || n != _len(A[0])? exc("non square matrix"):
           {! AT=transpose(A)} {! I = eye(n) !} transpose(inv1(AT,AT,I,I,0,n));
>> AI=inv(A);
>> ^AI %>;
[    [ 4.000000000000003, 3.000000000000002, -1.0000000000000007 ],
     [ -2.0000000000000018, -2.0000000000000013, 1.0000000000000004 ],
     [ 5.0000000000000036, 4.000000000000003, -1.0000000000000009 ]
]
>> ^matrixMult(A,AI)%>;
[    [ 1.0, 8.881784197001252E-16, 0.0 ],
     [ 1.7763568394002505E-15, 1.0, -4.440892098500626E-16 ],
     [ 0.0, -8.881784197001252E-16, 1.0 ]
]
>> ^inv(M);
**Computation stopped by exception "singular matrix" thrown by inv1
>>
```

To verify the correctness of the algoritm, we have multiplied A by the computed inverse AI. The result is indeed the identity matrix I_3 in conformity of the inverse definition – note that the displayed matrix contains some approximation errors (4 instances of zeros are represented by values with approximation smaller than 10^{-15}).

Let us now discuss the complexity of the function *det.* The step of triangulization now involves n columns of known terms instead of one is based on triangularization, i.e., the matrix has size $n \times 2n$ instead of $n \times (n+1)$. As constants can be dropped off, the complexity remains $\Theta(n^3)$. Also the subsequent step of solving upper triangular systems becomes more elaborated: this time the number of systems to be solved is n instead of 1. Hence the complexity of thi step passes from $\Theta(n^2)$ to $\Theta(n^3)$. In the whole, the complexity remains $\Theta(n^3)$.

5.6. Finite State Automata

Given an *alphabet* Σ, a *string* σ on Σ is any sequence of (possibly repeated) symbols on Σ and $|\sigma|$ denotes the length of σ – if $|\sigma| = 0$, σ is called the *empty string* and is denoted by ε. The concatenation operator is "$\cdot$" and may be omitted whenever no confusion arises. Given the set Σ^* of all strings on Σ, a language is any subset of Σ^* – note that Σ^* is a language as well and is called the *universal language*.

A *regular expression* on Σ is a sequence of symbols in Σ and of additional symbols (*meta-symbols*): ε ( ) + * | (the latter also denoted by $\cup$) with this format:

- any symbol in Σ is a regular expression;
- the empty string ε is a regular expression;
- if e is a regular expression, then e^+, e^*, (e) are regular expressions;
- if e_1 and e_2 are two regular expressions, then $e_1 \cdot e_2$ (or, simply $e_1\, e_2$) and $e_1 \mid e_2$ are regular expressions.

A regular expression e on Σ defines a language $L(e)$ on Σ consisting of all strings that match e according to the following rules:

- if e is any symbol in Σ or is ε then $L(e) = \{e\}$;
- if $e = e_1^+$ then $L(e) = L(e_1)^+$, i.e., a string in $L(e_1^+)$ is equal to the concatenation of 1 or more strings in $L(e)$;
- if $e = e_1^*$ then $L(e) = L(e_1)^*$, i.e., a string in $L(e_1^*)$ is equal to the concatenation of 0, 1 or more strings in $L(e)$;
- if $e = (e_1)$ then $L(e) = L(e_1)$;
- if $e = e_1\, e_2$ then $L(e\ ) = L(e_1) \cdot L(e_2)$, i.e., a string in $L(e)$ is equal to the concatenation of a string in $L(e_1)$ and of a string in $L(e_2)$;
- if $e = e_1 \mid e_2$ then $L(e\ ) = L(e_1) \cup L(e_2)$.

All the languages defined by a regular expression are called *regular*. For example, the regular expression $e = (ab)^+(b \mid d)^*c^+$ is matched by all strings starting with a pair *ab*, which can be repeated many times, ending with one or more *c* and possibly having in-between a symbol *b* or *c*, that can be repeated many times. The strings *ababbdbbcc* and *abdc* are in $L(e)$ but *abadc* and *abbdb* are not. A language defined by a regular expression is called *regular*.

A regular language is recognized by a *Deterministic Finite State Automaton* (*DFSA*), that is a simple abstract machine having an input tape, where an input string α on a given alphabet is stored, and a finite number of *states* (represented by nodes, i.e., circles, and labeled by a unique identifier), which include the *start state* (denoted by an incoming arrow to the circle),

one or more *final states* (denoted by double concentric circles) and possible intermediate states. The automaton reads a symbol on the input tape and moves from one state to another according to some predefined rules, called transitions. A *transition* (represented by directed edges, i.e., arrows) is a partial function from pairs (state, symbol) to states – an edge from q to p with label a defines the transition from the state q to the state p when the current read is a. The automaton is *deterministic* if the are no two edges with the same label leaving the same node.

Given a string α as input, the machine starts the computation by first entering into the start state and then reading the symbols of α from left to right and executing the applicable transitions. The computation ends as soon as either the whole input string is read or no further transition is applicable (i.e., the automaton hangs). The input α is recognized if both α is entirely read and the last state is final. The set of all strings recognized by a DFSA A is a language denoted by $L(A)$. Given any regular expression e, there always exists a DFSA A such that $L(e) = L(A)$. For example, a DFSA recognizing the language defined by $(ab)^+(b \mid d)^*c^+$ is shown in Figure 1:

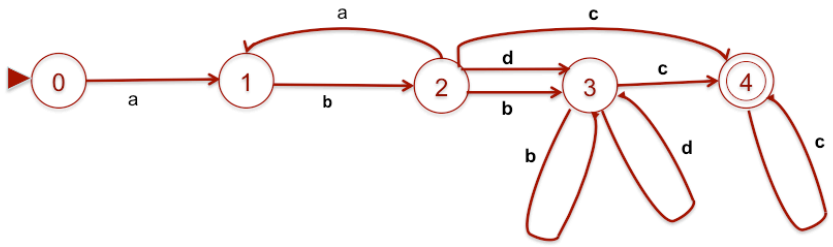


Fig. 1 *Deterministic Automaton for* $(ab)^+(b \mid d)^*c^+$

The automaton can be implemented in CalcuList as follows:

```
>> q4(S,i,n) : i>=n? true: S[i]=='c' && q4(S,i+1,n);
>> q3(S,i,n) : i < n && (  ( (S[i]=='b' || S[i]=='d') && q3(S,i+1,n) ) ||
                           ( S[i]=='c' && q4(S,i+1,n) ) );
>> q1(_,_,_) : null; /* dummy definition to handle mutual recursion with q2 */
>> q2(S,i,n) : i < n && (  ( S[i]=='a' && q1(S,i+1,n) ) ||
                           ( (S[i]=='b' || S[i]=='d') && q3(S,i+1,n) ) ||
                           ( S[i]=='c' && q4(S,i+1,n) ) );
>> q1(S,i,n) : i < n && S[i] =='b' && q2(S,i+1,n);
>> q0(S,i,n) : i < n && S[i]=='a' && q1(S,i+1,n);
>> r(S) : q0(S,0,_len(S));
>> ^ r("ababbdbbcc");
true
>> ^ r("abdc");
true
>> ^ r("abadc");
false
>> ^ r("abbdb");
false
>>
```

An alternative solution is to cast the input string to a list and to perform the membership checks on this list:

```
>> q4L(L) = L==[] || ( L[.]=='c' && q4L(L[>]) );
>> q3L(L) = L!=[] && ( ( (L[.]=='b' || L[.]=='d') && q3L(L[>]) ) ||
                       ( (L[.]=='c' && q4L(L[>])) ) );
>> q1L(_) = null;
>> q2L(L) = L!=[] && ( ( L[.]=='a' && q1L(L[>]) ) ||
                      ( (L[.]=='b' || L[.]=='d') && q3L(L[>]) ) ||
                      ( (L[.]=='c' || q4L(L[>])) ) );
```

```
>> q1L(L) = L!=[] && L[.]=='b' && q2L(L[>]);
>> q0(S,i,n) = i < n && S[i]=='a' && q1(S,i+1,n);
>> rL(S) = q0L(S@list);
>> ^ rL("ababbdbbcc");
true
>> ^ r("abadc");
false
>>
```

As a second example, consider the automaton in Figure 2 that recognizes strings representing numbers in the floating-point format.

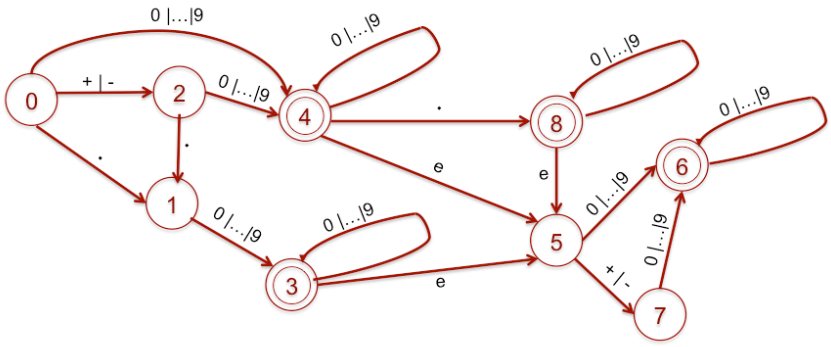


Fig. 2 *Deterministic Automaton for floating-point numbers*

The implementation in CalcuList follows:

```
>> isDigit(C) = '0' <= C && C <= '9';
>> q6F(L) : L==[] || ( isDigit(L[.]) && q6F(L[>]) );
>> q7F(L) : L!=[] && isDigit(L[.]) && q6F(L[>]);
>> q5F(L) : L!=[] && ( (isDigit(L[.]) && q6F(L[>])) ||
                       ( (L[.]== '+'|| L[.]== '-') && q7F(L[>]) ) );
>> q8F(L) : L==[] || ( (isDigit(L[.]) && q8F(L[>])) ||
                       (L[.]== 'e' && q5F(L[>])) );
>> q4F(L) : L==[] || ( (isDigit(L[.]) && q4F(L[>]) ) ||
                       (L[.]== 'e' && q5F(L[>])) ||
                       (L[.]== '.' && q8F(L[>])) );
>> q3F(L) : L==[] || ( (isDigit(L[.]) && q3F(L[>])) ||
                       (L[.]== 'e' && q5F(L[>])) );
>> q1F(L) : L!=[] && isDigit(L[.]) && q3F(L[>]);
>> q2F(L) : L!=[] && ( (isDigit(L[.]) && q4F(L[>])) ||
                       (L[.]=='.' && q1F(L[>])) );
>> q0F(L) : L!=[] && ( (isDigit(L[.]) && q4F(L[>])) ||
                       ( (L[.]== '+'|| L[.]== '-') && q2F(L[>] ) ) ||
                       (L[.]== '.' && q1F(L[>])) );
>> rF(S) : q0(S@list);
>> ^ rF("+2.232e-5");
true
>> ^ rF(".024");
true
>> ^ rF("+.024");
true
>> ^ rF("e+2");
false
>> ^ rF(".");
false
>>
```

A deterministic finite state automaton can be made *universal* by adding an additional tape that is used to represent a specific automaton. Giving the description of a DSFA *A* and a string α as

input, a *universal* DFSA produces the same result of *A* on α. We next illustrate how to define a universal deterministic finite state automaton in CalcuList.

We store the final states into the list F and represent the transitions as triplets (in the order, symbol, source state and target state) into a list G to represent the transitions – we assume that the states be labeled by non-negative integers and the start state be 0. For the automaton of Figure 1, the list of final states is F=[4] and the lists of transitions is:

G = [['a',0,1], ['a',2,1], ['b',1,2], ['b',2,3], ['b',3,3], ['c',2,4], ['c',3,4], ['c',4,4],
['d',2,3], ['d',3,3]];

The implementation of the universal DFSA is done by the following function:

uq(F, G, L, q) = q <0? false: L==[]? member(q,F): uq(F,G,L[>], next(G,L[.],q));

A state with negative label states that a transition has hanged in some state. If the whole string α has been read, α is recognized if and only if the last state is in the list F – the check is done by the function *member*. If the string is not yet fully read then the list G is consulted to find the next state for the transition from the current state and the current read – the operation is implemented by the function *next*:

next(G,x,q) = G==[]? -1: G[.][0]==x && G[0][1]==q? G[0][2]: next(G[>],x,q);

A session for the definition of the universal automaton and its implementation for the automaton in Figure 1 is shown next:

```
>> next(G,x,q) : G==[]? -1: G[.][0]==x && G[0][1]==q? G[0][2]: next(G[>],x,q);
>> member(x, L) : L==[]? false: L[.]==x? true: member(x,L[>]);
>> uq(F, G, L, q) : q <0? false:
                         L==[]? member(q,F):  uq(F,G,L[>], next(G,L[.],q));
>> ur(F,G,S) : uq(F,G,S@list);
>> F=[4];
>> G=[ ['a',0,1], ['a',2,1], ['b',1,2], ['b',2,3], ['b',3,3], ['c',2,4],
       ['c',3,4], ['c',4,4], ['d',2,3], ['d',3,3] ];
>> ^ur(F,G,"ababbdbbcc");
true
>> ^ur(F,G,"abdc");
true
>> ^ur(F,G,"abadc");
false
>> ^ur(F,G,"abbdb");
false
>>
```

Let us now show how to use the universal DFSA for implementing the automaton of Figure 2. The list of final states is F=[3, 4, 6, 8]. To simplify the definition of the transitions in G, we select one representative symbol for the edges with multiple labels, in particular '9' for the labels "0 | ...| 9" and '+' for "+ | -". Then the definition G simplifies to:

G =[['9',0,4], ['9',1,3], ['9',2,4], ['9',3,3], ['9',4,4], ['9',5,6], ['9',6,6], ['9',7,6], ['9',8,8],
['+',0,2], ['+',5,7], ['.',0,1], ['.',2,1], ['.',4,8], ['e',3,5], ['e',4,5], ['e',8,5]];

Because of the simplified definition of G, the input string α must be preliminary modified by the function *modInpFP*, which replaces a symbol with its representative. The overall implementation of the automaton in Figure 2 is snown next:

```
>> modInpFP(L) = L==[]?[]: isDigit(L[.])? ['9' | modInpFP(L[>]) ]:
                  L[.]=='-'? ['+' | modInpFP(L[>]) ]: [L[.] | modInpFP(L[>]) ];
>> ur1(F,G,S) = uq(F,G,modInpFP(S@list),0);
>> F=[3, 4, 6, 8];
>> G =[ ['9',0,4], ['9',1,3], ['9',2,4], ['9',3,3], ['9',4,4], ['9',5,6],
        ['9',6,6], ['9',7,6], ['9',8,8], ['+',0,2], ['+',5,7], ['.',0,1],
```

```
        ['.',2,1], ['.',4,8], ['e',3,5], ['e',4,5], ['e',8,5] ];
>> ^ur1(F,G,"+2.232e-5");
true
>> ^ur1(F,G,".024");
true
>> ^ur1(F,G,"+.024");
true
>> ^ur(F,G,"e+2");
false
>> ^ur(F,G,".");
false
>>
```

A regular language may contain some syntactic properties that are hard (or even impossible) to express in the standard formalism of regular expressions. In such cases, the properties can be checked using syntactic controls, that constitute the *static semantics* of the language – despite the name, this part does not deal with semantic issues of the language but it simply adds further syntactic rules. Static semantics can be implemented by using suitable functions that are called by the automaton while executing a transition. As an example, consider the language of the dates expressed in the format: *day* (digits), *month* (letters), *year* (digits).

We divide the syntactic analysis into two parts: (1) verify whether an input string is composed by three substrings (the first and the third representing an integer and the second consisting of a sequence of letters only) and (2) check whether the three substrings correctly represent a date. The first analysis is directly made by a DFSA whiles the second one is performed with suitable static semantics actions. Note that, as the language is regular, such actions could be avoided by implementing a much more elaborated automaton[6].

The DFSA implementing the first part of the analysis is depicted in Figure 3.

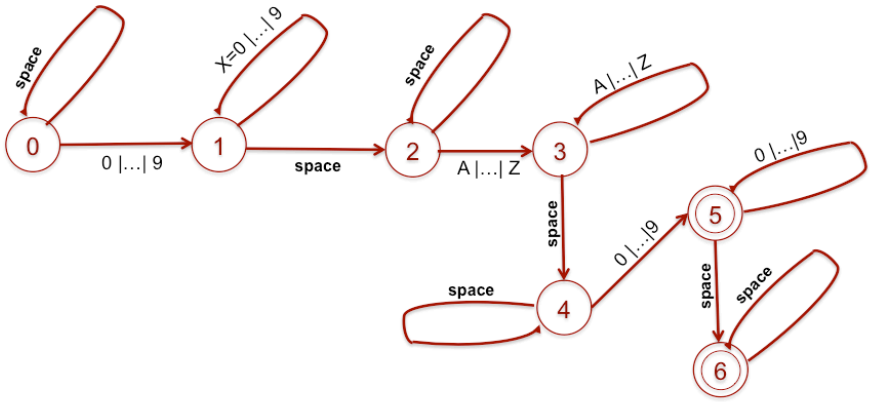


Fig. 3 *Basic DFSA for recognizing dates*

The DFSA implementation in CalcuList is presented below. Most of the actions issued by the DFSA are calls to simple functions that have been already used in other examples – in particular, *isSpace*, *isLetter*, *isDigit* and *toUpperCase*.

The function *isYear*(*y*) simply checks whether the integer *y* is in a prefixed range, which we arbitrary choosed as [1900, 2030]. To validate the number of admissible days per month, we define a function *isMonthDays*(MD,M,d) that uses a json MD with fields (*month*, *days*) to verify whether the number of days d is admissible for the month M – for simplicity we assume that February has always 28 days.

[6] A formalism is not for all seasons: it must be used in its strong points and replaced or extended in the weak ones.

```
>> MonthDays = { "JANUARY": 31, "FEBRUARY": 28, "MARCH": 31, "APRIL": 30,
        "MAY": 31, "JUNE": 30, "JULY": 31, "AUGUST": 31, "SEPTEMBER": 30,
        "OCTOBER": 31, "NOVEMBER": 30, "DECEMBER": 31};
>> isYear(x) : 1900 <= x && x <= 2030;
>> toUpperCase(x) : x >= 'a' && x<='z'? (x-'a'+'A')@char: x;
>> isSpace(x) : x==' ';
>> isLetter(x) : ( x >='a' && x <='z' ) || ( x >='A' && x <='Z');
>> isDigit(x) : x >= '0' && x <= '9';
>> isMonthDay(MD, M, d) : _isKey(MD,M) && MD[M]>=d && d>0;
>> q6D(L) : L==[] || isSpace(L[.]) && q6D(L[>]);
>> q5D(L,y) :     L==[]?  isYear(y): isDigit(L[.])? q5D(L[>],y*10+L[.]-'0'):
                  isSpace(L[.]) && isYear(y) && q6D(L[>]);
>> q4D(L) :       L==[]? false: isDigit(L[.])?  q5D(L[>],L[.]-'0'):
                  isSpace(L[.]) && q4D(L[>]);
>> q3D(L,d,M,MD) :      L==[]? false: isLetter(L[.])?
                        q3D(L[>],d, M+toUpperCase(L[.]),MD):
                        isSpace(L[.]) && isMonthDay(MD,M,d) && q4D(L[>]);
>> q2D(L,d,MD) :  L==[]? false:
                  isLetter(L[.])? q3D(L[>],d, toUpperCase(L[.])@string,MD):
                  isSpace(L[.]) && q2D(L[>],d,MD);
>> q1D(L,d,MD) :  L==[]? false: isDigit(L[.])? q1D(L[>], d*10+L[.]-'0',MD):
                  isSpace(L[.]) && q2D(L[>], d,MD);
>> q0D(L,MD) :    L==[]? false: isSpace(L[.])? q0D(L[>],MD):
                  isDigit(L[.]) && q1D(L[>],L[.]-'0',MD);
>> rD(S,MD) : q0D(S@list,MD);
>> ^rD(" 31 june 2017",MonthDay);
false
>> ^rD(" 30 june 2017 ",MonthDay);
true
>>
```

The automaton first extracts the substring denoting the day and convert it into an integer, then extract the characters denoting the month and convert them into a string, afterwards it checks the correctness of the pair (month day) and finally extracts the substring denoting the year and convert it into an integer that is checked to be a correct year.

5.7. Finite State Transducers

A *finite state transducer* (FST) is a deterministic finite state machine with two tapes: an input tape and an output tape, whereas an ordinary finite state automaton has the input tape only. A FST is not only capable to recognize a regular language as an ordinary finite state automaton but it can perform semantic actions on the input string whose results are written on the output tape. We have already described static semantic actions that simply perform additional syntactic controls for which there is no need for an output tape. Instead truly semantic actions for a language must use an output tape to return the result of the actions, which can typically be of two types: (1) *interpretation,* if the output is the result of the computation describer by the input string, and (3) *translation,* if the output is the description of the computation into another language, e.g., a machine language.

As an example of interpreter, consider the language representing floating point numbers as strings and the recognizing automaton in Figure 2. To transform the automaton into a transducer that writes the double value of the floating point string on the output tape, we introduce the following global variables to be used inside global settings:

- *v*: the final double value of the floating point number;
- *vI*: the integer part of the number;
- *vD*: the decimal part of the number;
- *vE*: the exponent

- *pI*: a power of 10 used to compute the value of the integer part of the mantix;
- *pE*: a power of 10 used to compute the value of the exponent
- *LT*: a list of spurious characters in the string, following the substring representing a double – we shall require that this list be empty if the whole string must exactly match a double.

The attributes are defined as labeled variables, with label DOUBLE, so that their values can be inizialized and modified inside star functions (i.e., with side effects). Actually, the values are inizialized at the start of state q0Db, in particular *v, vI* and *vD* are set to 0, LT is set to the empty list and *pI* and *pE* are set to 1 by GSCs (Global Setting Commands), to be executed as actions at the start of q0Db. The other semantic actions are performed in the various pertinent states by means of GSCs, to be executed as actions at the end of the functions, as it happens for synthesized attributes within grammar rules. The implementation is done by the functions defined in the session below. They use the function *isDigit*, that has been defined in the previous session.

```
>> DOUBLE: v, vI, vD, vE, pI, pE; /* Labeled Global Variables */
>> q6Db*(L) : <DOUBLE*> L==[]? true: isDigit(L[.])?
              q6Db(L[>]) {! vE=vE+(L[.]-'0')*pE !} {! pE=pE*10 !}:
              true {! LT=L !};
>> q7Db*(L) : <DOUBLE*> L==[]? false: isDigit(L[.])?
             q6Db(L[>]) {! vE=vE+(L[.]-'0')*pE !}:  false;
>> q5Db*(L) : <DOUBLE*> L==[]? false: isDigit(L[.])?
             q6Db(L[>]) {! vE=vE+(L[.]-'0')*pE !}:
             L[.]=='+'? q7Db(L[>]): L[.]=='-'? q7Db(L[>]) {! vE=-vE !}: false;
>> q8Db*(L) : <DOUBLE*> L==[]? true: isDigit(L[.])?
             q8Db(L[>]) {! vD=(vD+(L[.]-'0'))/10 !}:
             L[.]=='e'||L[.]=='E'? q5Db(L[>]): true {! LT=L !};
>> q4Db*(L) : <DOUBLE*> L==[]? true: isDigit(L[.])?
             q4Db(L[>]) {! vI=vI+(L[.]-'0')*pI !} {! pI=pI*10 !}:
             L[.]=='e'||L[.]=='E'? q5Db(L[>]):
             L[.]=='.'? q8Db(L[>]): true {! LT=L !};
>> q3Db*(L) : <DOUBLE*> L==[]? true: isDigit(L[.])?
             q3Db(L[>]) {! vD=(vD+(L[.]-'0'))/10 !}:
             L[.]=='e'||L[.]=='E'? q5Db(L[>]): true {! LT=L !};
>> q1Db*(L) : <DOUBLE*> L==[]? false: isDigit(L[.])?
             q3Db(L[>]) {! vD=(vD+(L[.]-'0'))/10 !}: false;
>> q2Db*(L) : <DOUBLE*> L==[]? false: isDigit(L[.])?
             q4Db(L[>]) {! vI=vI +(L[.]-'0') *pI !}:
             L[.]=='.' && q1Db(L[>]);
>> q0Db*(L) : <DOUBLE*>
             {! vI=0 !} {! vD=0 !} {! vE=0 !} {! pI=1 !} {! pE=1 !} {! LT=[] !}
             L==[]? false:
             L[.]=='+'? q2Db(L[>]) {! v=(vI+vD)*_pow(10,vE) !}:
             L[.]=='-'? q2Db(L[>]) {! v=-(vI+vD)* _pow(10,vE) !}:
             isDigit(L[.])? q4Db(L[>])
             {! vI=vI+(L[.]-'0')*pI !} {! v=(vI+vD)*_pow(10,vE) !}:
             L[.]=='.'? q1Db(L[>]) {! v=vD*_pow(10,vE) !}: false;
>> rDb*(S) : <DOUBLE*> q0Db(S@list) && LT==[]? V :
                       exc("error: wrong double definition");;
>> ^rDb("-221.34e+02");
-22134.0
>> ^DOUBLE.v;
-22134.0
>> !about DOUBLE;
------VARIABLES WITH LABEL DOUBLE-------
1: (GV1)@[2] v: double = -22134.0
2: (GV2)@[4] vI: int = 221
3: (GV3)@[6] vD: double = 0.33999999999999997
4: (GV4)@[8] vE: int = 2
```

```
5: (GV5)@[10] pI: int = 100
6: (GV6)@[12] pE: int = 10
7: (GV7)@[14] LT: list = 0 (->Heap)
---End of VARIABLES WITH LABEL DOUBLE---
>> ^rDb("22e-2+3");
**Computation stopped by exception "error: wrong double definition" thrown by
rDb
>> !about DOUBLE;
1: (GV1)@[2] v: double = 0.22
2: (GV2)@[4] vI: int = 22
3: (GV3)@[6] vD: int = 0
4: (GV4)@[8] vE: int = -2
5: (GV5)@[10] pI: int = 10
6: (GV6)@[12] pE: int = 1
7: (GV7)@[14] LT: list = 523929 (->Heap)
---End of VARIABLES WITH LABEL DOUBLE---
>> ^q0Db("22e-2+3"@list);
true
>>
```

Concerning the query `^rDb("22e-2+3")`, observe that the the string is not recognized as a double since it consists of a prefix substring representing the double `22e-2`, followed by the substring `"+3"`, that is stored into the variable `DOUBLE.LT`. The query `^q0Db("22e-2+3"@list)` returns *true* as a prefix substring represents a double and can be therefore used to continue the analysis of the suffix substring stored into `DOUBLE.LT`.

Let us now present a more elaborated example of interpretation using transducers to compute the value of an expression consisting of a sequence of sums and subtractions, without parentheses to keep the language regular and, therefore, recognizable by a finite state automaton. A regular expression that generates this language is: `DOUBLE((+|-)DOUBLE)*`, where `DOUBLE` is considered as a token returned by `q0Db`. Examples of expressions of this language are: "+23+37-11", "-2+4", "-4", "2-18".

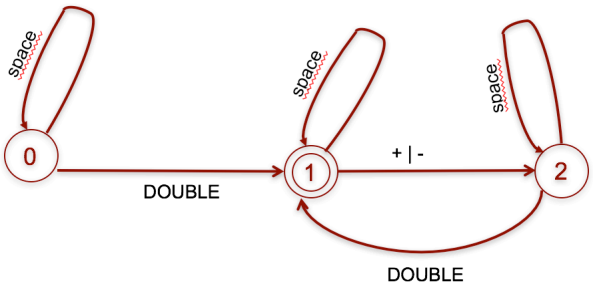


Fig. 4 *Recognizing automaton for a sequence of sums and subtractions*

The recognizing automaton is shown in Figure 4 and the functions implementing an interpreter of the language is presented in the session below. The value of the expression is stored into the labeled variable INTER.v.

```
>> isSpace(x) : x==' ' || x=='\t'; /* space and horizontal tabulator */
>> INTER: v; /* labeled variable storing the value of the expression */
>> q1I*(_) : null; /* dummy definition to solve mutual recursion */
>> q2I*(L,s) : <INTER*,DOUBLE*> L==[]? false: isSpace(L[.])? q2I(L[>],s):
      q0Db(L)? {! v=s=='+'?v+DOUBLE.v: v-DOUBLE.v !} q1I(LT): false;
>> q1I*(L) : L==[]? true: isSpace(L[.])? q1I(L[>]):
             L[.]=='+' || L[.]=='-'? q2I(L[>],L[.]): false;
>> q0I*(L) : <INTER*,DOUBLE*> {! v=0 !} L==[]? false:
             isSpace(L[.])? q0I(L[>]):
             q0Db(L)? {! v=DOUBLE.v !} q1I(LT): false;
>> rI*(S) : <INTER*> q0I(S@list)? v:exc("error: wrong expression");
```

```
>> ^rI("22e-2+3");
3.22
>> ^INTER.v;
3.22
>> ^rI("+23 + 37 - 1");
49
>>
```

Observe that also DOUBLE contains a variable named *v*, that is the value of the recognized token of type *double*. To distinguish the two homonym variables inside a function using them, we include also their label names. However, the label that occurs first in the list of global variables used in a function can be omitted – for instance, in the functions above, the variable *v* alone refers to the label INTER, that is listed before the label DOUBLE.

Let us now show and example of a translator for the language of sequence of sums and subtractions without parentheses. We refer to a simple stack-based abstract machine with the following instructions, each composed by a string (representing an operation code) and a value of type double:

- PUSH *val*: add the double *val* to the top of the stack;
- ADD 0: sum the two values at the top of the stack and replace them with the result;
- SUB 0: subtract the value at the top of the stack from the previous one in the stack and replace them with the result;
- PRINT 0: print the value at the top of the stack, remove it from the stack and stop the computation;
- DUPL 0: duplicate the the value at the top of the stack.

An implementation of this stack machine by means of the function *simpleM* follows. To test the implementation, we call *simpleM* by passing a program as a list C of instructions, where each instruction is a pair of operator – operand represented by a binary list. The program pushes the values 23 and 27 into the stack, sums the two values, duplicates the result 60 and prints it, push the value 11 and subtracts it from the partial result 60, thus obtaining 49. This value is eventually printed.

```
>> SIMPLEM: S; /* labeled variable storing the machine stack */
>> execI*(I) : <SIMPLEM*>
      I[0]=="PUSH"? {! S=[I[1]|S] !} true:
      I[0]=="SUM"? S==[]||S[>]==[]? exc("error: missing operand(s) for SUM"):
                        {! S[1]=S[0]+S[1] !} {! S=S[>] !} true:
      I[0]=="SUB"? S==[]||S[>]==[]? exc("error: missing operand(s) for SUB"):
                        {! S[1]=S[1]-S[0] !} {! S=S[>] !} true:
      I[0]=="PRINT"? S==[]? exc("error: missing operand for PRINT"):
                        {^ S[0] %+ '\n' ^} {! S=S[>] !} true:
      I[0]=="DUP"? S==[]? exc("error: missing operand for DUP"):
                        {! S=[S[0]|S] !} true:
      exc("error: undefined instruction");
>> simpleM1*(C) : C==[]? null: execI(C[.]) && simpleM1(C[>]);
>> simpleM*(C) : <SIMPLEM*> {! S=[] !} simpleM1(C);
>> C=[ ["PUSH",23], ["PUSH",37], ["SUM",0], ["DUP",0], ["PRINT",0],["PUSH",11],
       ["SUB",0],   ["PRINT",0] ];
>> ^simpleM(C);
60
49

>>
```

Observe that the printing of the values 60 and 49 are done by a CPS (*Global Printing Command*) inside the function *execI*. The function *simpleM* eventually returns null, which is as an empty string printed by default.

We are now ready to define the function *rT* implementing a translator of the language of sequences of summations and subtractions into a program in the above simple machine. The translator calls the functions corresponding to the states of the automaton depicted in Figure 4 and the function *q0Db* for getting a double operand as a token.

```
>> TRANSL: C;/* labeled variable storing the target machine code */
>> q1T*(_) : null;  /* dummy definition to solve mutual recursion */
>> q2T*(L,s) : <TRANSL*,DOUBLE*> L==[]? false: isSpace(L[.])? q2T(L[>],s):
               q0Db(L)? {! C += [ ["PUSH",v] ] !}
               {! C += s=='+'? [ ["SUM",0] ]: [ ["SUB",0] ] !}
               q1T(LT): false;
>> q1T*(L) : <TRANSL*> L==[]? true {! C += [ ["PRINT",0] ] !}:
             isSpace(L[.])? q1T(L[>]): L[.]=='+' || L[.]=='-'? q2T(L[>],L[.]):
             false;
>> q0T*(L) : <TRANSL*,DOUBLE*> {! C=[] !} L==[]? false:
             isSpace(L[.])? q0T(L[>]):
             q0Db(L)? {! C += [ ["PUSH",v] ] !} q1T(LT): false;
>> rT*(S) : <TRANSL*> q0T(S@list)? C: exc("error: wrong expression");
>> ^rT("+23+37-11");
[ [ "PUSH", 23 ], [ "PUSH", 37 ], [ "SUM", 0 ], [ "PUSH", 11 ], [ "SUB", 0 ],
[ "PRINT", 0 ] ]

>> rTsM*(S) : <TRANSL*> q0T(S@list)? simpleM(C): exc("error: wrong expression");
>> ^rTsM("+23+37-11");
49

>>
```

The function *rTsM* first calls the function *q0T* to translate an arithmetic expression into a stack virtual machine program *C* and, then, calls the function exmpleM to execute C on the virtual machine.

5.8. Pushdown Automa

<<To be extracted from Parser_Discendente>>

A grammar is in Greibach normal form if (1) the right-hand sides of all production rules start with a terminal symbol, possibly followed by other terminal or non-terminal symbols and (2) there exist no two rules $X \rightarrow a\ \alpha$ e $X \rightarrow b\ \beta$ with $a \neq b$. The empty production is allowed only for the axiom.

As example, we consider the following grammar of well-parenthesized arithmetic expression, i.e., the two operands of an arithmetic operation must be enclosed by parentheses such as (a+3) or ((a+2)*3):

E → (E O E) | L | C
L → a | b | ... | z | A | B | ... | Z |
C → 0 | 1 | ... | 9
O → + | *

The above grammar can be put into Greibach normal form as follows:

E → (E O E) | a | b | ... | z | A | B | ... | Z | 0 | 1 | ... | 9
O → + | *

The language recognizer in CalcuList is described next that use the global setting to handle the global variable *error*.

```
isLetter(X)& = X >= 'a' && X <= 'z' || X >= 'A' && X <= 'Z';
isDigit(X)& = X >= '0' && X <= '9';
```

```
error&=false;
accept*(S[],C)[]= error || S==[]? S {error=true}: S[^]==C? S[*]: S {error=true};
O*(S[])[] = error || S==[]? S {error=true}: S[^]=='+' || S[^]=='*'? S[*]: S {error=true};
E*(S[])[] =     error || S==[]? S {error=true}:
                S[^]=='(' ? accept( E( O( E( accept(S,'(' ) ) ) ), ')'):
                isLetter(S[^]) || isDigit(S[^])? S[*]: S {error=true};
recognize*(S[])& = {error=false}  E(S)==[]&& ! error ? true: false {error=true};
```

La funzione *riconosci* risponde true se non è stato rilevato alcun errore e la lista della stringa è stata letta tutta. Le pre-impostazione imposta *errore* a falso prima di passare l'analisi della stringa *E* e la post-impostazione lo imposta a vero prima di restituire *false* in quanto l'errore potrebbe non essere stato rilevato dalla funzione *E* in quanto determinato dalla presenza di ulteriori caratteri a fine stringa. Le varie funzioni implementano in maniera quasi immediata le regole grammaticali. Come esempio di utilizzo con esito positivo, si scriva ?recognize("(3+(((A*(A+5))+B)*4))");

Come secondo esempio consideriamo una grammatica per espressioni ben parentesizzate un po' più estesa che permetta di avere come operandi variabili con più di un carattere o numeri con più di una cifra. Inoltre rendiamo la grammatica astratta cioè consideriamo che i simboli terminali siano token (cioè simboli astratti) restituiti dall'analizzatore lessicale - ad esempio un numero è visto come un unico simbolo piuttosto che un sequenza di cifre.

E → PARTA E OPER E PARTC | ID | NUM

I token sono definiti da espressioni regolari - per evitare confusioni inseriamo i simboli terminali tra parentesi

OPER → "+" | "*"
PARTA → "("
PARTC → ")"
ID → ("a" | ... | "z" | "A" | ... | "Z") ("a" | ... | "z" | "A" | ... | "Z" | "0" | "1" | ... | "9")*
NUM → ("0" | "1" | ... | "9")+

Due stringhe corrispondenti a due token possono essere separati da un numero qualsiasi di spazi bianchi.

Anche in questo caso è possibile riconoscere il linguaggio con CalcuList. Anche in questo caso dividiamo il riconoscimento in due parti: analisi lessicale e parser. L'analisi lessicale consiste nel trasformare la stringa da riconoscere in una lista di token. Al contrario dell'implementazione in Java, conviene costruire preliminarmente la lista di tutti i token e non passarli uno alla volta su richiesta del parser. Di seguito l'implementazione dell'analizzatore lessicale attraverso la funzione *tokenize*:

```
OPER=0;
PARTA=1;
PARTC=2;
ID=3;
NUM=4;
tokens[]=["OPER "@d, " PARTA "@d, "PARTC "@d, "ID "@d, "NUM "@d]; ;
isSpace(X)& = X == ' ';
isLetter(X)& = X >= 'a' && X <= 'z' || X >= 'A'&& X <= 'Z';
isDigit(X)& = X >= '0' && X <= '9';
error&=false;
errLex&=false;
```

```
skipLetterOrDigit(L[])[] = L==[]? []: isLetter(L[^]) || isDigit(L[^])?
skipLetterOrDigit(L[*]): L;
skipDigit(L[])[] = L==[]? []: isDigit(L[^])? skipDigit(L[*]): L;
tokenize1*(L[])[]=    error || L==[]? []:
                      L[^]=='+' || L[^]=='*' ? [OPER | tokenize1(L[*])]:
                      L[^]=='(' ? [PARTA | tokenize1(L[*])]:
                      L[^]==')' ? [PARTC | tokenize1(L[*])]:
                      isLetter(L[^]) ? [ID | tokenize1(skipLetterOrDigit(L[*]))]:
                      isDigit(L[^]) ? [NUM | tokenize1(skipDigit(L[*]))]:
                      isSpace(L[^]) ? tokenize1(L[*]): [] {error=true}{errLex=true};
tokenize*(L[])[] = {error=false} {errLex=false} tokenize1(L);
printTokens*(L[])& = L==[]? {?'\n'%c} true: {?tokens[L[^]]@l%s}printTokens(L[*]);
```

Preliminarmente i token sono codificati come numeri assegnati a variabili mnemoniche per agevolare la lettura. La funzione *tokenize1* riceve in input una stringa e restituisce una lista di token, impostando a falso (valore 0) *errLex*. L'impostazione di errore è effettuata dal parser; tuttavia, nel caso di utilizzo autonomo dell'analizzatore lessicale, è messa a disposizione la funzione tokenize che imposta a a falso (valore 0) *errore*. Gli spazi vengono scartati da *tokenize1*; per il resto la funzione implementa in maniera immediata le espressioni regolari. Come esercizio si potrebbe riconoscere un numero anche in formato decimale o floating point.

Il parser è realizzato nel seguente modo:

```
errPars&=false;
accept*(S[],C)[]=     error || S==[]? S {error=true}{errPars=true}:
                      S[^]==C? S[*]: S {error=true}{errPars=true};
O*(S[])[] =     error || S==[]? S {error=true}{errPars=true}:
                S[^]==OPER? S[*]: S {error=true}{errPars=true};
E*(S[])[] =     error || S==[]? S {error=true}{errPars=true}:
                S[^]==PARTA ? accept( E( O( E( accept(S,PARTA ) ) ) ), PARTC):
                S[^]==ID || S[^]==NUM? S[*]: S {error=true}{errPars=true};
recognize*(S[])& = {error=false}{errPars=false} E(tokenize(S))==[]&& !error ?
                                    true: false {errPars=error? errPars: true} {error=false};
```

La funzione *riconosci* imposta preliminarmente gli errori a falso e poi passa alla funzione *E* la lista di token prodotta da *tokenize*. In caso di mancato riconoscimento dovuto al fatto che non tutti i token sono stati "consumati", la post-impostazione in *riconosci* imposta a vero *errPars*. Le altre funzioni implementano in maniera immediata le regole grammaticali. Come esempio di utilizzo con esito positivo, si scriva: ?recognize("(35+(((AB*(A27+5555))+pippo)*49))");. Come esempio di utilizzo con esito negativo da parte dell'analizzatore lessicale, si scriva ?recognize("(35^(((AB*(A27+5555))+pippo)*49))");. Come esempio di utilizzo con esito negativo da parte del parser, si scriva ?recognize("(35+(((AB*(A27+5555))+pippo)*49)"); - manca una parentesi finale. Come esempio di utilizzo con esito negativo da parte del parser per mancato svuotamento della lista di token, si scriva ?recognize("(35+(((AB*(A27+5555))+pippo)*49)))"); - c'è una parentesi in più alla fine.

Possiamo effettivamente concludere che l'analisi sintattica in presenza di una grammatica in forma normale di Greibach deterministica è abbastanza semplice e immediata. Esistono due problemi che riducano l'applicabilità del metodo: (i) se il linguaggio non è deterministico non

è proprio possibile e (ii) nel caso sia possibile la grammatica è spesso estremamente lunga e disagevole da trattare.

Nel primo caso non c'è nulla da fare se non cambiare linguaggio. Nel secondo caso conviene scrivere una grammatica in un formato più agevole per costruire un algoritmo di riconoscimento. Di seguito mostreremo un metodo per l'analisi sintattica di tipo discendente (detto anche ricorsivo o top-down) che estende quello descritto per una grammatica in forma normale di Greibach deterministica, che costituisce il modello più semplice di riconoscimento discendente.

Recognition of 1^n 0 1^ n (Exam of 20/6/2014)

Recognition of Dyck language (Exam of 16/7/2014)

5.9. Parsers with Semantic Actions

<<To be extracted from Grammatiche con Attributi>>

6. Error and Exception Handling and Debugging Features

6.1. Errors and Exceptions

CalcuList captures possible syntactic and execution errors and, as consequence, it raises predefined exceptions. The exceptions are listed below according to four categories.

(a) *Exceptions while passing arguments to invoke CalcuList*

- "*Odd Number of Arguments Passed to the Program*": as explained next, the arguments must be give as pairs, each consisting of parameter name and value assigned to the parameter;
- "*Wrong Name for an Argument Passed to the Program*" – the valid parameter names are listed below;
- "*Integer Format Required for an Argument Passed to the Program*" – the value assigned to a parameter must be an integer;
- "*Non-Positive Value for an Argument Passed to the Program*" – the integer assigned to a parameter must be positive.

The above exceptions may be thrown only when arguments are passed to the java program. The arguments are used to increase the sizes of some data structures, that are internally used and have been dimensioned for a typical usage of CalcuList. The arguments must be given as pairs, each consisting of a parameter (denoting an internal data structure) and a positive integer (providing a new value for the parameter overriding the default one). The available parameters are:

- *EXECCS*: it refers to the size of the memory used by the virtual machine of Calculist to store an executable program – the default value is 32,000
- *EXECMS*: it refers to the size of the memory used by the virtual machine of Calculist to store data – the default value is 64,000
- *EXECOS*: it refers to the size of the memory used by the virtual machine of Calculist to store the output – the default value is 16,000

As an example, assume that we want to increase the output memory size to 20,000 and to extends the data memory to 80,000 elements. Then, we write:

java –jar CalcuList EXECOS 20000 EXECMS 80000

A new value for a parameter that is less than the default one does not have effect, i.e., the default value is preserved.

(b) *Exceptions during the execution of a CalcuList computation*

- "*Stack Heap Overflow*": the size of the data memory is insufficient but, before re-launching CalcuList with an increase of the memory by means of the parameter *EXECMS*, it is convenient to check the semantic correctness of the last computation (e.g., possible loops) – if the overflow has been caused by the growth of the stack then it is possible to continue the session, otherwise it is wise to restart the session;
- "*Zero Divide*": check the program to detect a division by zero;
- "*Output Overflow*": the size of the output memory is insufficient but again, before increasing it by means of the parameter *EXECOS*, it is convenient to check the semantic correctness of the commands used for the computation;
- "*Wrong Machine Operator*": check the commands used for the computation, particularly those for the manipulation of list, or eventually, please report a possible CalcuList bug;

- "*Wrong Memory Address*": as the preceding exception;
- "*List Index Out of Range*": as above;
- "*Exec Failure*": as above;
- "*Negative List Index*": check the usage of list indices that cannot be negative in CalcuList.

(c) *Exceptions aborting the session*

- "*Fatal Error*": the session is abnormally closed, so carefully check the commands used for the last computation, particularly those for the manipulation of lists or eventually report a possible CalcuList bug;
- "*Wrong Printing Format*": it should be a very unlikely exception – please record the whole session and report a CalcuList bug.

(d) *Exceptions for syntactic errors*

- "*Wrong Double Number*": the current string is expected to represent a double but it is not conform to the floating point format;
- "*Wrong Escape Sequence*": a wrong escape sequence has been written as a character or inside a string – the valid ones are \b or \t or \n or \f or \r or \" or \' or \\;
- "*Missing quote*": the closing quote for a character constant is missing;
- "*Wrong Option*": valid options are 'variables' or 'functions' or 'debug' or 'history' or any prefix of them as well as 'on' or 'off' after the option 'debug';
- "*Invalid Symbol*": an unexpected symbol is encountered;
- "*Wrong Token*": an unexpected token is encountered instead of other ones displayed in the error message;
- "*Type Mismatch*": an operand has a type that is not conform to its usage;
- "*Undefined Identifier*": a variable or a function is used without having been previously defined;
- "*Actual Parameter Type Mismatch*": the actual parameter of a function has a type different from the formal parameter given in the function definition;
- "*Actual Parameter Number Mismatch*": the number of actual parameters in the call of a function is different from the number of formal parameter in the function definition;
- "*Missing* '?' *at the beginning*": a query must start with an exclamation mark;
- "*Side Effect Function called by a Function with no side effect*": the definition of a function without side effect cannot call a function with side effects;
- "*Global Variable in a Function with no Side Effects*": the definition of a function without side effect cannot use global variables;
- "*A function parameter cannot be an lvalue in a global set*": during the definition of a function with side effects, none of its parameters can be modified by an assignment inside a global setting;
- "*Global setting is not allowed*": only functions with side effect may include global setting;
- "*Wrong History Index*": a command "!history *n*" has been issued but *n* is either less than 1 or greater than the total number of commands stored into the history;
- "*Wrong Definition Type*": a variable or function is redefined with a type different from the initial definition;
- "*Wrong List Usage*": a variable is expected of type list but its has been defined with a different type;
- "*Wrong Number of Formal Parameters*": in the redefinition of a function, the number of formal parameters is different from the previous definition;
- "*Wrong Parameter Type*": in the redefinition of a function, the type of a formal parameter is different from the previous definition;
- "*Wrong Return Type*": in the redefinition of a function, the return type is different from the previous definition;

- "*Duplicated Formal Parameter Name*": in the definition of a function, the name of a parameter is the same as the name of the function or of a previous parameter.

After the exception, the current command is skipped until after the character ";" that identifies the end of a command.

6.2. Debugging on the CalcuList Virtual Machine (*CLVM*)

As mentioned before, it is possible to activate the debugger by issuing the service command "!debug on" or simply "!debug" or even more simply "!α", where α is any non-empty prefix of "debug". From then on:

- After defining a function, it is shown the unit code corresponding to the function that is written in the language of *CLVM* (CalcuList Virtual Machine);
- After defining a global variable or issuing a query, it is shown both the unit code and the overall executable program that includes the unit codes of all called functions, suitably linked together with the main unit code.

An example follows.

```
$java –jar CalcuList.jar
     ***********************************************
                    ***  CalcuList   ***
     ***********************************************
     ******** Release 1.1 of November 17, 2014 ********
<<a=3;
<<!debug on;

------DEBUG ON------

<<fact(x)=x <= 1? 1: x*fact(x-1);

------UNIT CODE------
0     START       / * fact
1     PUSHFP -1
2     DEREF
3     PUSHD 1.0
4     LE
5     JUMPEQ0 8
6     PUSHD 1.0
7     JUMP  17
8     NEXT
9     PUSHFP -1
10    DEREF
11    PUSHFP -1
12    DEREF
13    PUSHD 1.0
14    SUB
15    CALL  0     / -> fact
16    MULT
17    NEXT
18    RETURN 1
---End of UNIT CODE---
<<
```

We give some intuitions on the above code – more details can be found in the appendix. The instruction START sets up the environment for the execution of the function. PUSHFP -1 copies on top to the memory stack the address of the location in the stack where the function parameter value is stored. DEREF replaces the address with the content (i.e., the actual value of the parameter). Then PUSHD 1.0 adds the value 1 to the top of the stack. LE compares the last two elements E_1 and E_2 on top at the stack (in this case, E_1 is the value 1 and E_2 is the

actual parameter value) and, after removing the two operands, add 1 to the top if $E_1 \leq E_2$ or 0 otherwise. Then JUMPEQ0 8 checks the value on top (i.e., the one computed by LE) and if it is 0, the instruction with label 8 will be executed or otherwise the control passes to the subsequent instruction with label 6. The other instructions should have an intuitive semantics – note that NEXT is a dummy instruction and CALL 0 call recursively the function. The instruction RETURN 1 ends the execution of the function and the argument 1 states that the function has one parameter that is to be removed from the stack before returning the control to the calling function.

Suppose that the session continues with the definition of a global variable whose initialization calls the function fact: as the debug is active, it is first printed the unit code of the definition and then the executable program that include the code of the *fact* function. After the printing, the execution starts and stops after the execution of the first instruction and shows the state: internal registers, current instruction, next instruction, the contents of the stack, of the heap and of the output memory.

```
<<b=fact(a)+1;
>>

------UNIT CODE------
0     INIT  2
1     PUSHI 1
2     PUSHI 0
3     DEREF
4     CALL  0     / -> fact
5     PUSHD 1.0
6     ADD
7     MODV
8     HALT
---End of UNIT CODE---

------ABSOLUTE CODE------
0     INIT  2
1     PUSHI 1
2     PUSHI 0
3     DEREF
4     CALL  9     / -> fact
5     PUSHD 1.0
6     ADD
7     MODV
8     HALT
9     START / * fact
10    PUSHFP      -1
11    DEREF
12    PUSHD 1.0
13    LE
14    JUMPEQ0 17
15    PUSHD 1.0
16    JUMP  26
17    NEXT
18    PUSHFP -1
19    DEREF
20    PUSHFP -1
21    DEREF
22    PUSHD 1.0
23    SUB
24    CALL  9     / -> fact
25    MULT
26    NEXT
27    RETURN 1
---End of ABSOLUTE CODE---
```

```
**STATE:
 FP = 0 SP = 2 HP = 49999 OP = 0
 Current Instruction : 0      INIT  2
 Next Instruction    : 1      PUSHI 1

++ STACK
Main  ------------
GV 0  0: 3.0
GV 1  1: 0.0
--------------------

++ HEAP
--------------------

++ OUTPUT
--------------------

Number of steps before next pause? --enter 0 to go to the end, -1 to stop debug
<<
```

The execution is resumed by entering a positive number *n* and stops again after the execution of *n* instructions. It is also possible to proceed either (1) by entering 0 (the execution continue in debugging mode without further interruptions) or (2) by entering -1 (the execution is completed with the debugger disabled).

To disable the debugger for the subsequent commands, type "!debug off" or simply "!α off", where α is any non-empty prefix of "debug".

7. Conclusion and What Next

An overview of the usage of CalcuList has been given in this tutorial. Additional insigths are presented in the appendices.

New desirable features to be added in forthcoming releases:

1. optimize tail recursion implementation that is presently done in the naive way of assigning a stack frame to every instance of the tail recursive function – this extension only requires some minor rewriting of some CalcuList classes;
2. remove the constraint that a function must be defined before it is used – define it as a dummy function (null in the rhs);
3. remove the limitation that a lambda function cannot be defined inside another lambda function;
4. deep remove of a global variable A = #null – A will not be accessible anymore;
5. add a first dummy first element to a list as it is done now for json;
6. remove the present limitation that a function cannot return a function – this extension should not require much work;
7. infer types for function parameters and returned value, although we shall not be able to provide a complete inference as we shall preserve the present weak typing for lists and jsons;
8. provide a statically test of whether a function with shallow list operations has side effect or not – the extension is non complex if applied to each function separately from the other ones, but the real challenge is to consider a group of functions where a function may pass a list parameter according to the typical scheme for tail recursion.

- Defining local variables inside global setting – it is necessary a suitable syntax (e.g., an initial global set {!<*list-of-local-variables*>}) but implementation should not be complex (they are kinds of additional formal parameters that can be however used as l-value in subsequente global settings - local variables are removed from the stack when the function returns the control to the calling function);
- Type Integer: the definition should be a# = 0; probably this new feature does not add much to the language (a part from introducing some syntactic controls on the usage of integers, e.g., a double can be assigned to an integer only after an explicit casting) and implementation should be straightforward – the type list will continue to have as basic type double, that is more expressive w.r.t. the other two basic types (Boolean an integer).

8. References

[1] The Python language reference, https://docs.python.org/2/reference/index.html.

Appendix A: The CalcuList Virtual Machine (CLVM)

A.1. Overview of CLVM

As for many modern programming languages (e.g., Java and Python), a CalcuList program *X* is translated into a target program *Y* to be run on a virtual machine, called *CalcuList Virtual Machine* (*CLVM*). As CLVM interpreter is written in Java and the CalcuList programming environment is written in Java as well, it turns out that the whole language platform is independent from any hosting real machine.

A CLVM program is composed by one or more *Code Units* (*CU*): a *Main Unit* (*MU*) possibly followed by a number of *Function Units* (*FU*). A program execution starts by running the MU code, which may eventually call the execution of an FU instance and temporally suspends its execution. The MU code will resume the execution at the instruction next to the call as soon as the called instance will finish its execution. In turn, the called unit instance may call other FU instances, including other instances of the same FU – note that the main unit instead cannot be recursively called. At a generic instant of the execution, the active CU instances will be: a unique running unit instance and possibly a chain of suspended unit instances. Each active CU instance has associated a *frame*, which is an area storing data accessed and stored by the unit instance. Once all FU instances (directly or indirectly) called by the MU have completed their executions and their frames have been thereafter removed, the MU will be the only active unit and will complete the overall execution.

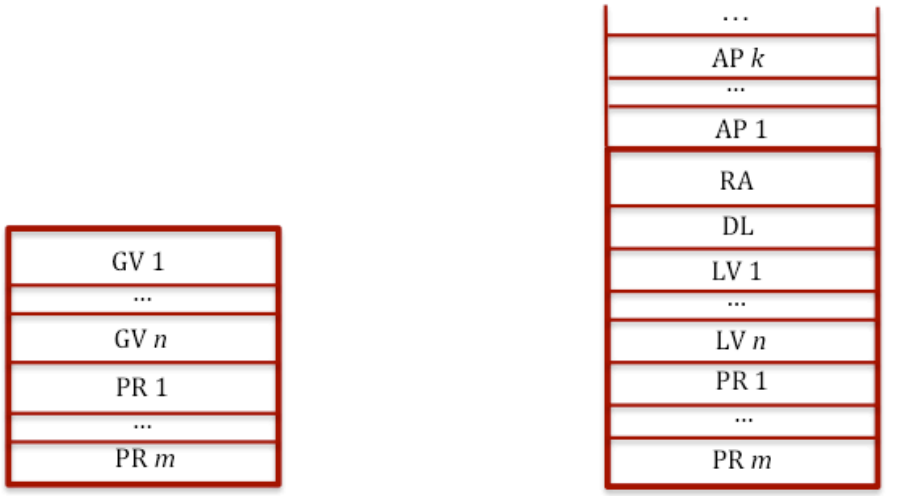
CLVM is an abstract computing machine that, like a real computing machine, has an instruction set and manipulates three memory areas, each of them organized as an array of 64-bit cells:

- a data memory, called *MEM*, which is divided into two parts: the STACK, which stores global variables and unit frames from the start of the array on, and the HEAP, which stores dynamic data, such as strings, lists and jsons, from the end of the array back – in our Java implementation of CLVM, *MEM* is defined as an array of doubles;
- a second memory, called *CODE*, storing the instructions of the running CLVM program – actually, in order to provide useful insights about the correspondence between CalcuList and CLVM codes during a debugging session, our Java implementation of CLVM defines an instruction as an object with size larger than 64 bits, consisting of three fields: the instruction code, the operand and a string storing a comment that will be used by the debugger;
- a third memory, called *OUTPUT*, which is used by the CLVM program during its execution to write the computation results that, after the program termination, are displayed by the CalcuList environment – *OUTPUT* is defined as an array of doubles.

The *MU frame* starts at the beginning of STACK and consists of the global variables and of its partial computation results. At the end of the overall CLVM program, this frame will be partly removed from the stack: only the global variables will persist. The case of a MU frame with *n* global variables (GV) and *m* partial results (PR) is shown in Figure A1(*a*).

An *FU frame* is more elaborated as it contains: the *CODE* address of the calling unit instruction that will be executed when the current unit ends (*Return Address* – *RA*), the STACK starting address of the calling unit frame (*Dynamic Link* – DL), possible local variables and partial computation results. If FU has formal parameters, their actual values (*Actual Parameters* – *AP*)

are stored immediately before the FU frame, i.e., at the end of the calling unit frame. An example of a FU frame with *n* local variables (LV), *m* partial results (PR) and *k* actual parameters (*AP*) is shown in Figure A1(*b*). Observe that parameters are listed in the reverse order the corresponding formal parameters are defined in the CalcuList function.



(a) *Frame of a Main Unit* (b) *Frame of a Function Unit*

Fig. A1 *Frame of Code Units*

During a program execution, the running unit frame is located by the register *FP* (*Frame Pointer*) that stores the address of the frame start cell. The top of STACK (i.e., the last used cell) is addressed by the register *SP* (*Stack Pointer*).

We next show how the STACK content evolves during a CLVM program execution. As an example, we consider a program with a main unit MU with one global variable, a function unit X with one formal parameter and two local variables and a function unit Y with two parameters and one local variable. A sketch of the program is presented in Figure A.2(*a*). The MU code starts with the instruction INIT 1 (which indicates that there is one global variable), ends with the halting command and includes the instruction for calling X. The X code starts with the instruction START 2 (which indicates that there are two local variables), ends with the instruction RETURN 1 (which indicates that there is one actual parameter to be removed together with the function frame) and it includes the instruction for calling Y. The Y code starts with the instruction START 1 (which states that there is one local variable) and ends with the instruction RETURN 2 (which states that there are two actual parameters to be removed together with the function frame).

At the start of the execution, the stack only contains the frame *F*(MU) of the main unit with a unique cell, the one storing the global variable – see Step 1 in Figure A.2(*b*). This frame grows by storing partial results until it calls X. Then the frame of X is put at the top of the stack: it contains the two addresses RA (pointer to the instruction of MU next to CALL X) and DL (pointer to the frame of MU) and the two local variables – see Step 2 in Figure A.2(*b*). The frame F(X) of X grows by storing partial results until it calls Y. Now it is the turn of the frame of Y to be at the top of the stack: RA points to the instruction of MU next to CALL Y, DL points to the frame of X and the unique local variable is stored next – see Step 3 in Figure A.2(*b*). When Y terminates its execution by issuing the command RETURN 2, F(Y) and the two actual parameters are removed from the stack and replaced by the value R(Y) returned by the unit – see Step 4 in Figure A.2(*b*). At this point, X resume its execution from the instruction next to CALL Y and eventually terminates by issuing the command RETURN 1. Then F(X) and the

actual parameter are removed from the stack and replaced by the value R(X) returned by the unit – see Step 5 in Figure A.2(*b*). The main unit MU resume its execution from the instruction next to CALL X and eventually terminates the whole program by issuing the command HALT. Then all elements of F(MU) are removed from the stack except the global variables, which are kept in the stack for possible future usage – see Step 6 in Figure A.2(*b*).

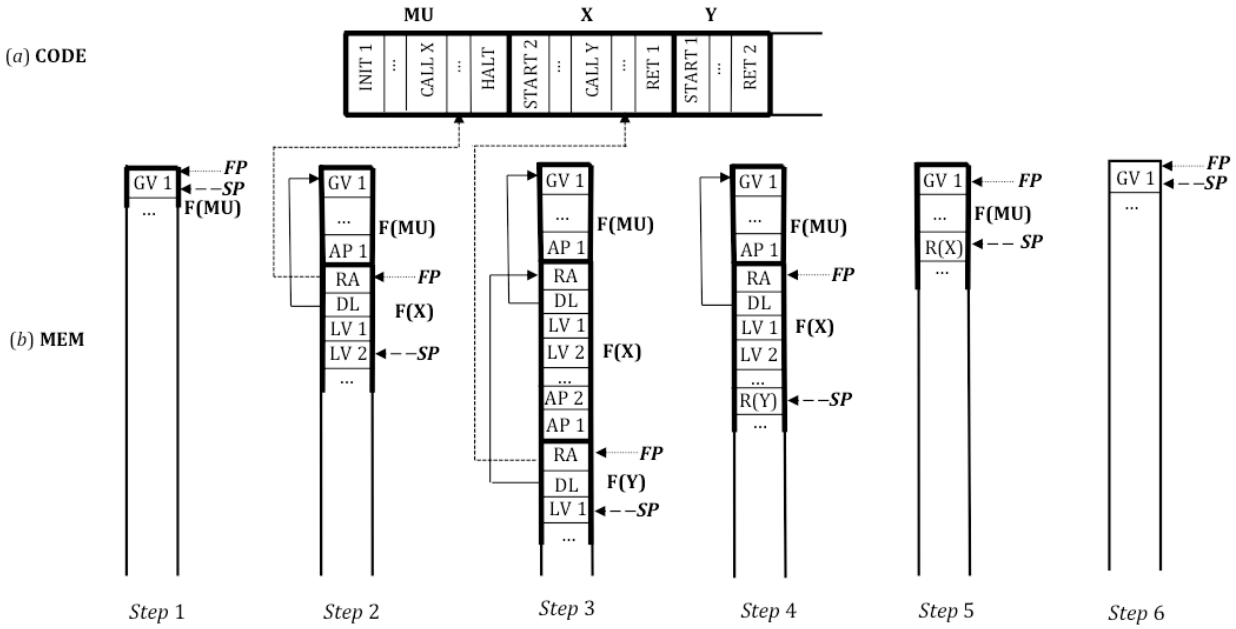


Fig. A.2: *Evolution of Stack Frames*

A.2. Architecture of CLVM

CLVM has been designed as a virtual machine with three distinct 64-bit memories:

- **MEM**: it stores the data handled by a CLVM program and is divided into two parts:
  - **STACK**: it stores global variables and unit framed, is located at the beginning of MEM and grows downwards;
  - **HEAP**: it stores values for dynamic data structures (lists and strings), is located at the end of MEM and grows upwards;
- **CODE**: it stores the running CLVM program and consists of a main unit MU and of the function units that are directly or indirectly called by MU; *in theory* instructions are stored into a 64-bit words with two formats:
  - *One Word Format*: the first 8 bits of the word stores the instruction code, the subsequent 32 bits are used for a possible integer operand and the remaining 24 bits are free;
  - *Two Word Format*: the first 8 bits of the first word stores the instruction code, the remaining 56 bits of the first word are free and the second word stores a double operand;

  We have emphasized "in theory" because, as mentioned before, our Java version of the CLVM machine interpreter instead defines an instruction as a complex object to both simplify the implementation and stores additional data to the debugger as strings;
- **OUTPUT**: it stores the results of the running CLVM program that will be eventually displayed at the end of the program execution by a CalcuList facility, external to CLVM.

CLVM is equipped with a processor having a "powerful" virtual *Arithmetic Logic Unit* (ALU) and a number of 64-bit registers:

- **FP** (*Frame Pointer*): pointer to the first element of the running unit frame;
- **SP** (*Stack Pointer*): pointer to the last (top) element of the stack;
- **HP** (*Heap Pointer*): pointer to the first free element of the heap (i.e., the last element of the heap is pointed by HP-1);
- **OP** (*Output Pointer*): pointer to the first free element of the output (i.e., the last element of the output is pointed by OP-1);
- **IP** (*Instruction Pointer*): pointer to the next instruction of the code, which will be executed after the current one;
- **IR** (*Instruction Register*): it contains the current instruction that is being executed by the ALU – as for a CODE element, in theory the register consists of two 64-bit words (the second word is for a possible double operand) but in our implementation is an object with a larger size;
- **FSPP** (*First String Pool Pointer*) and **LSPP** (*Last String Pool Pointer*): the pool of strings is implemented as a list of strings in the heap whose first element (the first stored string) is pointed by FSPP and the last element (the last stored string) is pointed by LSPP.

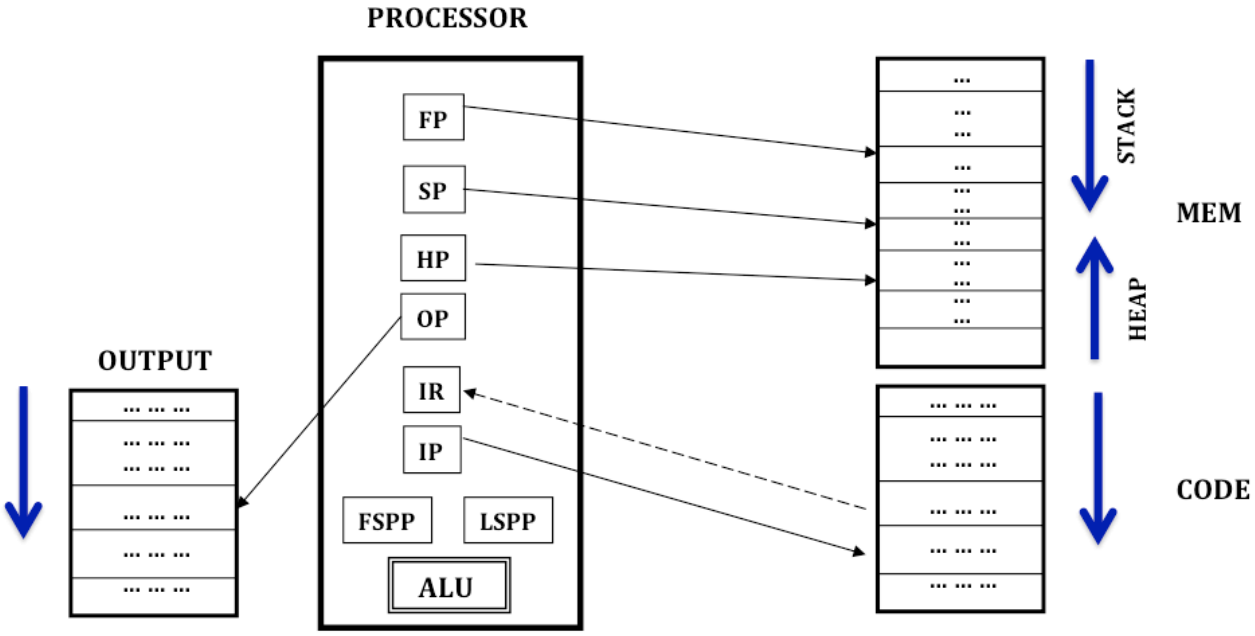


Fig. A.3: *CLVM Architecture*

The architecture of CLVM is depicted in Figure A.3. We point out the in our CLCM interpreter implementation, additional working registers are used, in particular:

- a register (WR1) for data of type double;
- four registers (WR2, WR3, WR4 and WR5) for data of type integer;
- three registers (MS, CS and OS) storing the memory sizes (respectively, for MEM, CODE and OUTPUT) – the default values are MS = 64,000, CS = 32,000 and OS = 16,000 but they can be modified at the launch of CalcuList by means of suitable external arguments, as shown in Section 6.1;
- the register GP storing the pointer to the garbage list of heap triplets that have been removed but can be reclaimed to store a json element, list element or field value,
- registers for the heap manipulation: HPC (pointer to the first available heap triplet to be used to store a json element, list element or field value), SPP (current pointer to the string pool), *NSPP* (pointer to a new string under construction).

The execution of a CLVM program follows the following scheme:

```
IP = 0 // address of the first instruction INIT k of the MU unit
while ( IP >= 0 ) {
        IR = CODE(IP) // load of the current instruction
        IP = IP +1 // IP points to the next instruction
        execute(IR) // execute the current instruction
}
```

As discussed in Section A.4 below, the IP register can be modified by jumping or calling instructions. If the operator code of the current instruction is not supported, the exception WRONG_MACHINE_OPERATOR is raised and the program terminates abnormally.

A.3. Data Types of CLVM

CalcuList is not strongly typed and a variable may change type during a session. Therefore, type checking is mainly performed at run-time; to this end, CLVM handles a number of data types and any value is stored as a pair: the actual value and its type, suitably encoded. Both actual values and type codes are stored into a 64-bit words[7] in the data memory MEM. The supported types are listed below together with the encoding used by our CLVM interpreter implementation:

- *double* with code 1 – actual double values are directly handled in the Java double format and, therefore, they follow the 64-bit double-precision format IEEE 754;
- *int* with code 2 – actual integer values are stored with the Java double format and, therefore, as 53 bits are used for the mantissa, they could take values from -2^{52} to $+2^{52}$ but CLVM enables suitable checks to restrict the range from -2147483648 to 2147483647 as for Java integers;
- *char* with code 3 – actual character values are stored with the Java double format but CLVM enables suitable checks to restrict the range to 16-bit unsigned integers representing Unicode UTF-16 codes as for Java integers;
- *bool* with code 4 – actual Boolean values are stored with the Java double format but CLVM enables suitable checks to restrict the range to 0 and 1, representing *false* and *true* respectively;
- *null* with code 5 – the typical null value is represented by zero but there is a second null value, represented by -1 and called *deleted null value*, that is used to mark the deletion of a list or json element (the element may be later inserted in a garbage list of triplets);
- *string* with code 6 – a string value is stored as a pointer to a string data structure in the heap, as it will illustrated later in this section – a pointer is represented as an integer with the Java double format as discussed above;
- *list* with code 7 – a list value is stored as a pointer to a list data structure in the heap, as it will illustrated later in this section;
- *json* with code 8 – a jason value is stored as a pointer to a json data structure in the heap, as it will illustrated later in this section;
- *type* with code 9 – there are 9 values for *type*, from 1 (*double*) to 8 (*json*);
- *metatype* with code 10 – there is a unique value for *metatype* : 9 (*type*);
- *funct* with code 11 – a function value is stored as a pointer to a CODE instruction, that is the first instruction of the function;
- *ref* with code 12 – a reference value is stored as a pointer to a stack element;

[7] Perhaps, some future version of CalcuList could introduce some memory optimization to reduce the amount of space – indeed 64 bits are too many for describing few data types!

- *field* with code 13 – this type is only internal and is used to recognize that a heap triplet is a field of a json.

The *listable values* are those with type codes between 1 (*double*) and 10 (*metatype*) and they can be included as elements of a list, i.e., they are: any value of type double, int, char, bool, string, list or json as well as the values: *double, int, char, bool, string, list, json* and *type*. The value *metatype* is not listable.

As for MEM, also data in OUTPUT are stored as pairs <value, type> and the supported types are all listable values as well as the following additional types for :

- *StringExcF* with code 14 – the same as the type *string* except that the string will be associated to a thrown exception;
- *CharQ* with code 15 – the same as the type *char* except that the character will be displayed between single apices;
- *StringQ* with code 16 – the same as the type *string* except that the string will be displayed between double apices;
- *NullQ* with code 17 – the same as the type *null* except that *null* will be actually displayed;
- *ListQF* with code 18 – the same as the type *list* except that the list will be displayed by indenting all elements as well as all possible subelements;
- *JsonQF* with code 19 – the same as the type *json* except that the json will be displayed by indenting all elements as well as all possible subelements;
- *ListQ1F* with code 20 – the same as the type *ListQF* except that the first level elements only will be indented;
- *JsonQ1F* with code 21 – the same as the type *JsonQF* except that the first level elements only will be indented.

The listable values and the five values listed above are called *printable values*.

A string "$a_1a_2...a_k$" is represented in the heap by k+2 contiguous cells: <k, Unicode(a_1), Unicode(a_2), ..., Unicode(a_k), *Pointer_Next_String*>, where *Pointer_Next_String* points to the next string in the heap. Strings are organized as a pool, implemented as a linked list whose elements are added in the order they are defined. The registers FSPP and LSPP point to the first and last string, respectively. When a new string is defined, it will be added at the end of the pool only if it is not included in it.

A list [e_1, e_2,..., e_k] is represented in the heap with (not-necessarily contiguous) k list elements, where the generic list element e_i is stored as a triple of contiguous cells (*triplet*) < value(e_i), type(e_i), *Pointer_Next_List_Element* >.

A json { k_1: v_1, k_2: v_2, ..., k_n: v_n,] is represented in the heap with (not-necessarily contiguous) n+1 fields, where the field 0, represented by the triplet < 0, field, *Pointer_Next_Json_Element* >, is fictitious and the generic json field i is stored as <*Pointer_Key_Value_i*, field, *Pointer_Next_Json_Element* >. *Pointer_Key_Value_i* points to a triplet <value(v_i), type(v_i), k_i >, where k_i is the pointer to the string describing the key.

As an example, we show how the following global variables are actually represented:

```
GV0= 3.5;
GV1=2;
GV2='h';
GV3=true;
GV4="house";
```

```
GV5=[ -3.14e-24, int,  null, 'h', "home" ];
GV6={ "home": "CS", "house": [-1, -2], "tag": 3 };
```

Note that *true*, *int* and *null* are not strings but the predefined type values. The variables are stored from the address 0 up to 13 in the stack as follows – each variable takes two memory words, one for the value and one for the type:

```
---------- [GV 0]: "GV0"
00000: 3.5
00001: 1       double
---------- [GV 1]: "GV1"
00002: 2
00003: 2       int
---------- [GV 2]: "GV2"
00004: 104     'h'
00005: 3       char
---------- [GV 3]: "GV3"
00006: 1       true
00007: 4       bool
---------- [GV 4]: "GV4"
00008: 63999       *->HEAP             pointer to the string "house"
00009: 6       string
---------- [GV 5]: "GV5"
00010: 63992       *->HEAP             pointer to the list
00011: 7       list
---------- [GV 6]: "GV6"
00012: 63971       *->HEAP             pointer to the json
00013: 8       json
----------
```

Strings, lists and jsons are stored in the heap as follows:

```
---------- [String 0]                  First String
63999: 5               length
63998: 104             'h'
63997: 111             'o'
63996: 117             'u'
63995: 115             's'
63994: 101             'e'
63993: 63974           *-> next string
---------- [List Element 0]            First Element of the list GV5
63992: -3.14E-24
63991: 1               double
63990: 63989           *->next list element
---------- [List Element 1]            Second Element of the list GV5
63989: 2               int
63988: 9               type
63987: 63986           *->next list element
---------- [List Element 2]            Third Element of the list GV5
63986: 0
63985: 5               null
63984: 63983           *->next list element
---------- [List Element 3]            Fourth Element of the list GV5
```

```
63983: 104          'h'
63982: 3            char
63981: 63974        *->next list element
-------------- [String 1]                Second String
63980: 4            length
63979: 104          'h'
63978: 111          'o'
63977: 109          'm'
63976: 101          'e'
63975: 63968        *-> next string
-------------- [List Element 4]          Fifth and Last Element of the list GV5
63974: 63980        *->HEAP               Pointer to the string "home"
63973: 6            string
63972: 0            end list
-------------- [Json Element 0]          Field 0 fictitious of the json GV6
63971: 0            null
63970: 13           field
63969: 63961        *->next json element
-------------- [String 2]                Third String
63968: 2            length
63967: 67           'C'
63966: 83           'S'
63965: 63937        *-> next string
-------------- [Key-Value 0]
63964: 63968        *->HEAP               Pointer to the string "CS"
63963: 6            string
63962: 63974        *->HEAP (key string)  Pointer to the string "home"
-------------- [Json Element 1]          Field 1 of the json GV6
63961: 63964        *->HEAP               Pointer to the Key-Value 0
63960: 13           field
63959: 63949        *->next json element
-------------- [List Element 5]          First Element of the list [-1,-2]
63958: -1
63957: 2            int
63956: 63955        *->next list element
-------------- [List Element 6]          Second and Last Element of the list [-1,-2]
63955: -2
63954: 2            int
63953: 0            end list
-------------- [Key-Value 1]
63952: 63958        *->HEAP               Pointer to the list [-1,-2]
63951: 7            list
63950: 63999        *->HEAP (key string)  Pointer to the string "house"
-------------- [Json Element 2]          Field 2 of the json GV6
63949: 63952        *->HEAP               Pointer to the Key-Value 1
63948: 13           field
63947: 63938        *->next json element
-------------- [String 3]                Fourth and Last String
63946: 3            length
63945: 116          't'
```

```
63944: 97              'a'
63943: 103             'g'
63942: 0               * last string
---------- [Key-Value 2]
63941: 3
63940: 2               int
63939: 63946           *->HEAP (key string) Pointer to the string "tag"
---------- [Json Element 3]                 Field 3 of the json GV6
63938: 63941           *->HEAP              Pointer to the Key-Value 2
63937: 13              field
63936: 0               end json
----------
```

A.4. The CLVM Instruction Set

A CLVM instruction consists of an operation code specifying the operation to be performed, followed by zero or one operand embodying a value to be operated upon. Additional operands may be stored at the top of the stack.

CLVM instructions are represented in this section by entries of the form shown below, grouped by typology:

- **mnemonic**: symbolic name of the instruction
- *Operation*: short description of the instruction
- *Format*: mnemonic, possibly followed by a mnemonic name for the operand
- *Stack Operands*: a first line describes the operands on top of the stack and a second line describes the possible result that will replace the stack operands – dots represent the portion of the stack that is not involved in the operation, operand names between brackets denote typed values occupying two words (for the value and for the type)
- *Output*: description of possible values added to OUTPUT
- *Description*: a longer description detailing constraints on Stack Operands contents, the operation performed, the type of the results, the register modifications etc.
- *Run-Time Exceptions*: description of possible exceptions.
- *Remark*: Optional additional comments.

A general remark is that an exception may be directly raised by the Java code implementing the CLVM interpreter if wrong memory addresses are stored into the memory. This exception, identified as "EXEC.FAILURE" by CLVM, is in general unexpected for the code generated by CalcuList. In this case, please report the whole printed message and the session while it occurred.

A final important remark is in order. When an instruction description states that the register HP is incremented, it is meant that the size of the heap is increased and, then, the actual value of HP is decreased as the heap grows backwards.

A4.1. *Instruction for Starting and Ending Code Units*

INIT

- *Operation*: start the main execution unit
- *Format*: INIT *n*
- *Stack Operands*: none
- *Description*: The operand *n* denotes the number of global variables and must be ≥ 0. The frame of the main execution unit is started. FP is set to 0 (start of the stack) and

SP is set to $2 \times n - 1$, i.e., SP is the address of the second memory word storing the last global variable.

HALT

- *Operation*: terminate the program execution
- *Format*: HALT
- *Stack Operands*: none
- *Description*: IP is set to -1 and, therefore, the program terminates.

START

- *Operation*: start the execution of the called function unit
- *Format*: START *n*
- *Stack Operands*: …, *RA* →
  …, *RA*, *DL*, <*val_local_var*$_1$,*type_local_var*$_1$>, …, <*val_local_var*$_n$, *type_local_var*$_n$>
- *Description*: The operand *n* denotes the number of local variables and must be ≥ 0. The frame of the called function unit is started. The stack top already contains the value of *RA* (Return Address), stored by the calling unit. The value of FP is pushed into the stack to store the dynamic link *DL* (the frame address of the calling function). Then FP is set to the address of *RA* in the stack (start of FU frame) and SP is incremented by $1+2 \times n$: in addition to the space occupied by *DL*, additional $2 \times n$ words are reserved for the *n* local variables. SP now points to the second memory word occupied by the last global variable or to DL if the are no local variables.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

RETURN

- *Operation*: end the execution of the called function unit
- *Format*: RETURN *n*
- *Stack Operands*: …, <*val_act_par*$_n$,*type_act_par*$_n$>, …, <*val_act_par*$_1$,*type_act_par*$_1$>, *RA*, *DL*, …, <*retVal, retType*> →
  …, <*retVal,retType*>
- *Description*: The operand *n* denotes the number of actual parameters passed to the current function unit from the calling unit and must be ≥ 0. The entire frame of the current function unit as well as all actual parameters are removed from the stack and replaced with the value computed by the function unit. IP is set to *RA*, so that the calling unit can resume its execution from the instruction next to the function call. FP is set to DL, so that the frame of the calling unit becomes the current frame. SP points to the second memory word storing the returned value, which therefore represents the last element of the calling unit frame.

THROWE

- *Operation*: throw an exception by writing it on top of OUTPUT and terminate the program.
- *Format*: THROWE
- *Stack Operands*: …, <*valOp*$_1$, *string*>, <*valOp*$_2$, *string*> →
  …,
- *Output*: … →
  …, <*valOp*$_1$, *stingExcF*>, <*valOp*$_2$, *string*>
- *Description*: The two stack operands must be of type *string*. After the writing on OUTPUT, IP is set to -1 and, therefore, the program terminates.

- *Run-Time Exceptions*: If there is no space to write on OUTPUT the exception OUTPUT_OVERFLOW is raised. If the two stack operand types are different from *string*, the exception THROWE_NOT_SUPPORTED is raised.
- *Remark*: The exception THROWE_NOT_SUPPORTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

A4.2. *Call Instructions*

CALL

- *Operation*: call a fixed function unit
- *Format*: CALL *funct_Addr*
- *Stack Operands*: … →
  …, *IP_val*
- *Description*: The operand *funct_Addr* is the address of the first instruction of the called function unit. The current value of IP (address of the instruction next to the call), named *IP_val,* is pushed into the stack – this value is not typed and will become the first element (RA) of the called function frame. Then IP is set to *funct_Addr,* so that the next instruction to be executed is the first instruction of the called function unit. SP is incremented by 1, i.e., SP is the address of the memory word storing *IP_val*.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

CALLS

- *Operation*: call a generic function unit passed by argument to the calling unit
- *Format*: CALLS
- *Stack Operands*: …, <*funct_Addr, funct*> →
  …, *IP_val*
- *Description*: The stack operand <*funct_Addr*> is the address of the first instruction of the called function unit and must be of type *funct* (= 11). This operand is removed from the stack and replaced by the current value of IP (address of the instruction next to the call), named *IP_val,* – this value is not typed and will become the first element of the called function frame. Then IP is set to the value of *funct_Addr,* so that the next instruction to be executed is the first instruction of the called function unit. SP is decremented by 1, i.e., SP is the address of the memory word storing *IP_val*.
- *Run-Time Exceptions*: if the stack operand is not of type *funct,* the exception CALLS_NOT_SUPPORTED is raised.
- *Remark*: The exception CALLS_NOT_SUPPORTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

A4.3. *Push Instructions*

PUSHD

- *Operation*: push a double value, represented with the long floating point format, into the stack
- *Format*: PUSHD *val*
- *Stack Operands*: … →
  …, <*val, double*>
- *Description*: The operand *val* is a double value and is pushed into the stack as typed value, i.e., together with the type *double* = 1. SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.

- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.
- *Remark*: In a possible CLVM interpreter implementation that strictly uses the 64-bit format for an instruction instead of an object as in our implementation, the operator PUSHD will take two words: the first will store the operation code and the second one the operand. In addition, IP must be also incremented by 1 to skip the second word.

PUSHI

- *Operation*: push an integer value into the stack
- *Format*: PUSHI *val*
- *Stack Operands*: … →
  …, <*val, int*>
- *Description*: The operand *val* is an integer value (i.e., from -2,147,483,648 to 2,147,483,647 according to the Java format for *int*) and is pushed into the stack as typed value, i.e., together with the type *int* = 2. SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

PUSHC

- *Operation*: push a character value into the stack
- *Format*: PUSHC *val*
- *Stack Operands*: … →
  …, <*val, char*>
- *Description*: The operand *val* is a character value (i.e., an integer from 0 to 65,535) and is pushed into the stack as typed value, i.e., together with the type *char* = 3. SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

PUSHB

- *Operation*: push a boolean value into the stack
- *Format*: PUSHB *val*
- *Stack Operands*: … →
  …, <*val, bool*>
- *Description*: The operand *val* is a boolean value (stored as 0 or 1) and is pushed into the stack as a typed value, i.e., together with the type *bool* = 4. SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

PUSHF

- *Operation*: push a CODE address into the stack
- *Format*: PUSHF *val*
- *Stack Operands*: … →
  …, <*val, funct*>
- *Description*: The operand *val* is the starting address of a function unit stored in CODE and is pushed into the stack as typed value, i.e., together with the type *funct* = 11. SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.

- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.
- *Remark*: in the code produced by CalcuList, *val* is an index that identifies a function and the index is negative for a lambda function. The index is transformed into a CODE address by the linker.

PUSHT

- *Operation*: push a listable type value into the stack
- *Format*: PUSHT *val*
- *Stack Operands*: … →
  …, <*val, type*>
- *Description*: The operand val must be an integer in the range from 1 to 8 (corresponding to the types *double, int, char, bool, null, string, list, json*) and is pushed into the stack as typed value, i.e., together with the type *type* (equal to 9). SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised. If val is not in the range from 1 to 8 the exception PUSHT_NOT_SUPPORTED is raised.
- *Remark*: The exception PUSHT_NOT_SUPPORTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

PUSHMT

- *Operation*: push the meta-type *type* into the stack
- *Format*: PUSHMT
- *Stack Operands*: … →
  …, <*type, metatype*>
- *Description*: The meta-type *type* (equal to 9) is pushed into the stack as typed value, i.e., together with the type *metatype* = 10. SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

PUSHN

- *Operation*: push the null value into the stack
- *Format*: PUSHN
- *Stack Operands*: … →
  …, <*0, null*>
- *Description*: The null value (equal to 0) is pushed into the stack as typed value, i.e., together with the type *null* = 5. SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

NULLIFY

- *Operation*: replace the stack operand with a null value into the stack and, in case of list or json, remove all elements – if the instruction operand is 1 the null value will be with the deleted format
- *Format*: NULLIFY *k*
- *Stack Operands*: …, <*opVal, opType*> →
  …, <*nullVal, null*>

- *Description*: The value of *k* is either 0 or 1 and the stack operand must be of a listable type. If *opType* is equal to list or json remove all elements of *opVal* (including the associated key-value triplets for the case of a *json*) by adding them into the garbage list. The stack operand is replaced by the null value <0, *null*> if *k* = 0 or otherwise by the deleted null value <-1, *null*>.
- *Run-Time Exceptions*: If the stack operand <*opVal, opType*> is not a listable value, the exception NULLIFY_NOT_SUPPORTED is raised.
- *Remark*: The exception NULLIFY_NOT_SUPPORTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

A4.4. *Loading and Dereferencing Instructions for the Stack*

LOADGV

- *Operation*: push the address of a global variable into the stack
- *Format*: LOADGV *k*
- *Stack Operands*: … →
  …, <*addr_gv_k, ref*>
- *Description*: The operand *k* is the index of a global variable (the global variables are progressively indexed from 0 on). As a variable is stored into two words and the first global variable is located at the address 0, the address of the global variable *k* is *addr_gv_k* = 2×*k*, and is pushed into the stack as a typed value, i.e., together with the type *ref* = 11. SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

LOADLV

- *Operation*: push the address of a local variable into the stack
- *Format*: LOADLV *k*
- *Stack Operands*: … →
  …, <*addr_lv_k, ref*>
- *Description*: The operand *k* is the index of a local variable of the current function unit (the local variables are progressively indexed from 0 on). As the first local variable is stored at the address FP+2, the address of the local variable *k* is *addr_lv_k* = FP + 2 × (1+*k*), and is pushed into the stack as typed value, i.e., together with the type *ref* = 11. SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

LOADARG

- *Operation*: push the address of a local variable into the stack
- *Format*: LOADARG *k*
- *Stack Operands*: … →
  …, <*addr_arg_k, ref*>
- *Description*: The operand *k* is the index of a the argument (actual parameter) of the current function unit (the arguments are progressively indexed from 1 on and are stored in the reverse order in the stack, prior to the start of the current function frame). As the first argument is stored at the address FP-2, the address of the argument *k* is *addr_arg_k* = FP − 2 × *k*, and is pushed into the stack as typed value, i.e., together with

the type *ref* = 10. SP is incremented by 2, i.e., SP is the address of the second memory word storing the pushed value.
- *Run-Time Exceptions*: if there is not enough space in MEM to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

DEREF
- *Operation*: replace a reference value with the referenced value stored in the stack
- *Format*: DEREF
- *Stack Operands*: …, <*addr, refType*> → …, <*referenced_val, resType*>
- *Description*: The value of *refType* must be equal to *ref*. The value *addr* points to a typed value in the stack. The typed value <*addr, refType*> is replaced by <*referenced_val, resType*> in the stack by copying the pair of words at the address MEM[*addr*].
- *Run-Time Exceptions*: If the stack operand is not a typed value or its type is different from *ref*, the exception DEREF_NOT_SUPPORTED is raised.
- *Remark*: The exception DEREF_NOT_SUPPORTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

MODV
- *Operation*: modify a referenced value in the stack
- *Format*: MODV
- *Stack Operands*: …, <*addr, refType*>, <*val, typeVal*> → …
- *Description*: The value of *refType* must be equal to *ref* and *typeVal* must be a listable type (i.e., equal to *double, int, char, bool, string, list, json, type* or *metatype*) or a *ref* type. The value *addr* points to a typed value in the stack. The stack typed operand <*typed_val*> at the top of the stack is copied into the area at the address MEM[*addr*], thus updating the typed value stored in it. All stack operands are removed from the stack by setting SP = SP – 4.
- *Run-Time Exceptions*: If the stack operand <*addr, ref*> is a typed value different from *ref*, the exception DEREF_NOT_SUPPORTED is raised. If *typeVal* is neither a listable type (i.e., equal to *double, int, char, bool, string, list, type* or *metatype*) nor a *ref* type then the exception MODV_NOT_SUPPORTED is raised.
- *Remark*: The exceptions DEREF_NOT_SUPPORTED and MODV_NOT_SUPPORTED are unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

A4.5. *Arithmetic Instructions*

ADD
- *Operation*: add the two typed values at the top of the stack
- *Format*: ADD
- *Stack Operands*: …, <val_op_1, $type\_op_1$ >, <val_op_2, $type\_op_2$> → …, <*val_res, type_res*>
- *Description*: The two stack operands must be compatible for the addition operation and the operation result depends from the operand types, i.e.:
  - both of them are of numeric type (i.e., *double*, *int* or *char*): then *val_res* = val_op_1 + val_op_2, where "+" is the arithmetic sum of two numbers, and *type_res* is equal to *double* if at least one of the two operands is *double* or the result is outside the *int* range (from -2,147,483,648 to 2,147,483,647), or to *int* otherwise.

- both of them are of type *list*: then *val_res* = *addr*(*val_op*$_1$ + *val_op*$_2$), where *val_op*$_1$ and *val_op*$_2$ are the two lists in the heap pointed by the operand values, "+" is the concatenation of the two lists that is obtained by linking the last element of the first list to the first element of the second list in the heap (i.e., the operation has the side effect of modifying the first list) and *addr*(*val_op*$_1$ + *val_op*$_2$) is the heap address of the concatenated list; *type_res* is equal to *list.*
- both of them are of type *string*: then <*res*> = *addr*(*val_op*$_1$ + *val_op*$_2$), where *val_op*$_1$ and *val_op*$_2$ are the two strings in the heap pointed by the operand values, "+" is the concatenation of the two strings that is obtained by copying the characters of the two of them in the heap (i.e., the operation has no side effects) and *addr*(*val_op*$_1$ + *val_op*$_2$) is the heap address of the new string (or of an existing string equal to the concatenated one); *type_res* is equal to *string*.
- *type_op*$_1$ is equal to *string* and *type_op*$_2$ is equal to *char*: then *val_res* = *addr*(*val_op*$_1$ + *val_op*$_2$), where *val_op*$_1$ is the string in the heap pointed by the value of the first operand, *val_op*$_2$ is the character stored in the second operand, "+" is the concatenation of the string with the character and *addr*(*val_op*$_1$ + *val_op*$_2$) is the heap address of the new string (or of an existing string equal to the concatenated one); *type_res* is equal to *string*.

SP is decremented by 2, i.e., SP is the address of the second memory word storing the result in the stack.

- *Run-Time Exceptions*: if there is not enough space in MEM to extend the heap after the concatenating a string with a string or a character, then the exception STACK_HEAP_OVERFLOW is raised. If the two stack operands are not compatible for the addition, the exception ADD_NOT_SUPPORTED is raised.
- *Remark*: Possible values for operands and result equal to NaN (Not a Number), positive infinite or negative infinite are handled by the Java interpreter implementation.

SUB

- *Operation*: subtract the two typed values at the top of the stack
- *Format*: SUB
- *Stack Operands*: <*val_op*$_1$, *type_op*$_1$ >, <*val_op*$_2$, *type_op*$_2$> → …, <*val_res, type_res*>
- *Description*: Both the two stack operands must be of numeric type (i.e., *double*, *int* or *char*). The result *val_res* is equal to *val_op*$_1$ – *val_op*$_2$, "–" is the arithmetic subtraction of two numbers, and *type_res* is equal to *double* if at least one of the two operands is *double* or the result is not in the *int* range (from -2,147,483,648 to 2,147,483,647), or to *int* otherwise. SP is decremented by 2, i.e., SP is the address of the second memory word storing the result in the stack.
- *Run-Time Exceptions*: If one of the two stack operands is not of numeric type, the exception SUB_NOT_SUPPORTED is raised.
- *Remark*: Possible values for operands and result equal to NaN (Not a Number), positive infinite or negative infinite are handled by the Java interpreter implementation.

MULT

- *Operation*: multiply the two typed values at the top of the stack
- *Format*: MULT
- *Stack Operands*: …, <*val_op*$_1$, *type_op*$_1$ >, <*val_op*$_2$, *type_op*$_2$> → …, <*val_res, type_res*>

- *Description*: Both the two stack operands must be of numeric type (i.e., *double*, *int* or *char*). The result *val_res* is equal to $val\_op_1 \times val\_op_2$, where "×" is the arithmetic multiplication of two numbers, and *type_res* is equal to *double* if at least one of the two operands is *double* or the result is not in the *int* range (from -2,147,483,648 to 2,147,483,647), or to *int* otherwise. SP is the address of the second memory word storing the result in the stack.
- *Run-Time Exceptions*: If one of the two stack operands is not of numeric type, the exception MULT_NOT_SUPPORTED is raised.
- *Remark*: Possible values for operands and result equal to NaN (Not a Number), positive infinite or negative infinite are handled by the Java interpreter implementation.

DIV

- *Operation*: divide the two typed values at the top of the stack
- *Format*: DIV
- *Stack Operands*: …, <val_op_1, $type\_op_1$ >, <val_op_2, $type\_op_2$> → …, <*val_res, type_res*>
- *Description*: Both the two stack operands must be of numeric type (i.e., *double*, *int* or *char*) and the second operand must not be equal to zero. The value of *type_res* is always *double* and *val_res* is equal to val_op_1 / val_op_2, where "/" is the arithmetic division of two numbers with possible decimal part. SP is decremented by 2, i.e., SP is the address of the second memory word storing the result in the stack.
- *Run-Time Exceptions*: If one of the two stack operands is not of numeric type, the exception DIV_NOT_SUPPORTED is raised. If the second operand has value zero then the exception ZERO_DIVIDE is raised.
- *Remark*: Possible values for operands and result equal to NaN (Not a Number), positive infinite or negative infinite are handled by the Java interpreter implementation.

NEG

- *Operation*: change the sign to the typed value at the top of the stack
- *Format*: NEG
- *Stack Operands*: …, <*val_op*, *type_op*> → …, <*val_res, type_res*>
- *Description*: The stack operand must be of numeric type (i.e., *double*, *int* or *char*). The result *val_res* is equal to –*val_op* and , *type_res* is equal to *type_op* if *type_op* is *double* or *int* or to *int* if *type_op* is *char*.
- *Run-Time Exceptions*: If the stack operand is not of numeric type, the exception NEG_NOT_SUPPORTED is raised.
- *Remark*: Possible values for operands and result equal to NaN (Not a Number), positive infinite or negative infinite are handled by the Java interpreter implementation.

A4.6. *Casting Instructions*

TOINT

- *Operation*: convert the typed value at the top of the stack into an integer
- *Format*: TOINT
- *Stack Operands*: …<*valOp, typeOp*> → …, <*intRes, int*>
- *Description*: The stack operand must have one of the following types:
  - *typeOp* = *double* and *valOp* is in the range from -2,147,483,648 to 2,147,483,647: *intRes* is the truncated value of *valOp.*

- *typeOp* is equal to *int* or *char* or *bool*: *intRes* is equal to *valOp* (and, then, for the case *typeOp* = *char,* to the Unicode character code and, for the case *typeOp* = *bool,* to 0 or 1 depending on whether *valOp* is *false* or *true*).

The typed stack operand is removed from the stack and replaced by the typed casted result.

- *Run-Time Exceptions*: If the stack operand type is different from *double*, *char*, *int* or *bool*, the exception TOINT_NOT_SUPPORTED is raised. This exception is also raised when the operand value is outside the range from -2,147,483,648 to 2,147,483,647.

TOCHAR

- *Operation*: convert the typed value at the top of the stack into a character
- *Format*: TOCHAR
- *Stack Operands*: …<*valOp, typeOp*> → …, <*charRes, char*>
- *Description*: The stack operand must have one of the following types:
  - *typeOp* is equal to *double* or to *int* and *valOp* is in the range from 0 to 65533: *intRes* is equal to the greatest integer less than or equal to *valOp.*
  - *typeOp* is equal to *char*: *intRes* is equal to *valOp* (i.e., no conversion is made).

  The typed stack operand is removed from the stack and replaced by the typed casted result.
- *Run-Time Exceptions*: If the stack operand type is different from *double*, *char* or *int*, the exception TOCHAR_NOT_SUPPORTED is raised. This exception is also raised when the operand value is outside the range from o to 65533.

TOTYPE

- *Operation*: convert the typed value at the top of the stack into its type
- *Format*: TOTYPE
- *Stack Operands*: …<*valOp, typeOp*> → …, <*valRes, typeRes*>
- *Description*: The stack operand must be a listable value with *typeOp* different from *metatype* (i.e., *typeOp* equal to *double, int, char, bool, string, list* or *type*) – there are two possible cases:
  - *typeOp* is equal to *double, int, char, bool, string, list* or *json*: *valRes* is equal to *typeOp* and *typeRes* is equal to *type.*
  - *typeOp* is equal to *type*: *valRes* is equal to *type* and *typeRes* is equal to *metatype*.

  The typed stack operand is removed from the stack and replaced by the result.
- *Run-Time Exceptions*: If the stack operand is not a listable value or its type is equal to *metatype*, the exception TOTYPE_NOT_SUPPORTED is raised.

TOLIST

- *Operation*: convert the typed value at the top of the stack into a list
- *Format*: TOLIST
- *Stack Operands*: …<*valOp, typeOp*> → …, <*valRes, list*>
- *Description*: The stack operand must have one of the following types:
  - *typeOp* = *list*: *typeRes* is equal to *valOp* (i.e., no conversion is made).
  - *typeOp* = *string*: a new list is constructed in the heap, whose elements are the characters of the string in the heap pointed by *valOp*; *valRes* is the address of the first element of this list.
  - *typeOp* = *json*: a new list is constructed in the heap, whose elements are k sublists [$fieldKey_i$, $fieldVal_i$], one for each of the *k* fields of the json *valOp*,

where *fieldKey*$_i$ and *fieldVal*$_i$ are respectively the key (represented as a string) and the value of the field *i*; *valRes* is the address of the first element of this list.

The typed stack operand is removed from the stack and replaced by the typed casted result.

- *Run-Time Exceptions*: If there is not enough space in MEM to extend the heap to construct the new list, then the exception STACK_HEAP_OVERFLOW is raised. If *typeOp* is not equal to *list* or *string* or *json*, the exception TOLIST_NOT_SUPPORTED is raised.

TOSTRING

- *Operation*: convert the typed value at the top of the stack into a string
- *Format*: TOSTRING
- *Stack Operands*: …<*valOp, typeOp*> → …, <*valRes, string*>
- *Description*: The stack operand must have one of the following types:
  - *typeOp* = *string*: *typeRes* is equal to *valOp* (i.e., no conversion is made).
  - *typeOp* = *char*: the string with the unique character *valOp* is constructed in the heap (if it does not already exist) and *valRes* is equal to the address of the string in the heap.
  - *typeOp* = *list* and the list pointed by *valOp* is only composed by characters: a string is constructed in the heap (if it does not already exist), whose characters are the elements of the list in the heap pointed by *valOp*; *valRes* is the address of the string in the heap.

  The typed stack operand is removed from the stack and replaced by the typed casted result.
- *Run-Time Exceptions*: If there is not enough space in MEM to extend the heap to construct the new string, then the exception STACK_HEAP_OVERFLOW is raised. If *typeOp* is not equal to *string, char* or *list* or if the list pointed by *valOp* contains elements different from characters, the exception TOSTRING_NOT_SUPPORTED is raised.

TOJSON

- *Operation*: convert the typed value at the top of the stack into a json
- *Format*: TOJSON
- *Stack Operands*: …<*valOp, typeOp*> → …, <*valRes, json*>
- *Description*: The stack operand must have one of the following types:
  - *typeOp* = *json*: *typeRes* is equal to *valOp* (i.e., no conversion is made).
  - *typeOp* = *list*: *valOp* must be a list with *k* sublists of two elements; the generic element *i* must have the format [*key*$_i$, *val*$_i$], is *key*$_i$ a string and *val*$_i$ is a listable value; where eac a new list is constructed in the heap, whose elements are k sublists [*fieldKey*$_i$, *fieldVal*$_i$], one for each of the *k* fields of the json *valOp*, where *fieldKey*$_i$ and *fieldVal*$_i$ are respectively the key and the value of the field *i*; *valRes* is the address of the json { *key*$_1$: *val*$_1$, …, *key*$_k$, *val*$_k$ }.

  The typed stack operand is removed from the stack and replaced by the typed casted result.
- *Run-Time Exceptions*: If there is not enough space in MEM to extend the heap to construct the new list, then the exception STACK_HEAP_OVERFLOW is raised. If *typeOp* is not equal to *json* or *list* or is equal to json but *valOp* is not structured as a list of sublists [Key, Value], the exception TOJSON_NOT_SUPPORTED is raised.

TUPLE

- *Operation*: convert the json at the top of the stack into a list of values, one for each field
- *Format*: TOLIST
- *Stack Operands*: …<*valOp, json*> → …, <*valRes, list*>
- *Description*: The stack operand must a json. A new list is constructed in the heap, whose elements are k values $fieldVal_i$, one for each of the k fields of the json *valOp*, where $fieldVal_i$ is the value of the field i; *valRes* is the address of the first element of this list. The stack operand is removed from the stack and replaced by the list result. HP is incremented by $3 \times (n\text{-}1)$, where n is the number of fields of *valOp,* including the fictitious initial field – HP's increment may be actually smaller if some garbage triplets are eventually reused.
- *Run-Time Exceptions*: If the stack operand is not a json, the exception TUPLE_NOT_SUPPORTED is raised. If there is not enough space in MEM to extend the heap to construct the new list, then the exception STACK_HEAP_OVERFLOW is raised.

A4.7. *Instructions for Mathematical Built-In Functions*

The instructions of this typology implement mathematical functions that are definitely beyond the "typical" capabilities of a realistic machine. They can be perhaps thought of as instructions executed by an advanced coprocessor.

RAND

- *Operation*: push a randomly-generated double value in the range from 0 to 1
- *Format*: RAND
- *Stack Operands*: … → …, <*val, double*>
- *Description*: A double number *val* in the range from 0 to 1 is randomly generated using the Java method *double random.nextDouble*(), where *random* is an object of the class *util.RANDOM*. SP is incremented by 2.
- *Run-Time Exceptions*: If there is not enough space in MEM to extend the stack the exception STACK_HEAP_OVERFLOW is raised.

EXP

- *Operation*: compute the natural exponential function of a number
- *Format*: EXP
- *Stack Operands*: …, <*valOp, typeOp*> → …, <*val, double*>
- *Description*: The stack operand must have a numeric type, i.e., *typeOp* must be equal to *double*, *int* or *char*. The number *val* is equal to the natural exponential function e^x, where e is the Euler's number and x is *valOp*, which is computed by the Java method *double Math.exp*(*double x*).
- *Run-Time Exceptions*: If the stack operand is not of numeric type, the exception EXP_NOT_SUPPORTED is raised.
- *Remark*: Possible values for operands and result equal to NaN (Not a Number), positive infinite or negative infinite are handled by the Java interpreter implementation.

LOG

- *Operation*: compute the natural logarithm of a number
- *Format*: LOG
- *Stack Operands*: …, <*valOp, typeOp*> → …, <*val, double*>

- *Description*: The stack operand must have a numeric type, i.e., *typeOp* must be equal to *double*, *int* or *char*. The value *valOp* must be non-negative. The number *val* is equal to the natural logarithm function $\log_e x$, where *e* is the Euler's number and x is *valOp*, which is computed using the Java method *double Math.log*(*double x*).
- *Run-Time Exceptions*: If the stack operand is not of numeric type, the exception LOG_NOT_SUPPORTED is raised.
- *Remark*: Possible values for operands and result equal to NaN (Not a Number), positive infinite or negative infinite are handled by the Java interpreter implementation.

POW

- *Operation*: compute the power of a number raised to another number
- *Format*: POW
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*val, double*>
- *Description*: The two stack operands must have a numeric type, i.e., , $typeOp_1$ and $typeOp_2$ must be equal to *double*, *int* or *char*. The number *val* is equal to a^b, where $a = valOp_1$ and $b = valOp_2$, and is computed by the Java method *double Math.log*(*double a, double b*). SP is decremented by 2.
- *Run-Time Exceptions*: If the stack operand is not of numeric type, the exception POW_NOT_SUPPORTED is raised.
- *Remark*: Possible values for operands and result equal to NaN (Not a Number), positive infinite or negative infinite are handled by the Java interpreter implementation.

REMAIN

- *Operation*: compute the remainder of two numbers
- *Format*: REMAIN
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*val, type*>
- *Description*: The two stack operands must have a numeric type, i.e., , $typeOp_1$ and $typeOp_2$ must be equal to *double*, *int* or *char*. The number *val* is equal to the remainder of the two operands and is computed by the Java operation $valOp_1$ % $valOp_2$. In general the result *val* is of type *int*; however, if one of the two operands is of type *double*, *type* is equal to double as well. SP is decremented by 2.
- *Run-Time Exceptions*: If the stack operand is not of numeric type, the exception REMAINDER_NOT_SUPPORTED is raised.
- *Remark*: Possible values for operands and result equal to NaN (Not a Number), positive infinite or negative infinite are handled by the Java interpreter implementation.

A4.8. *Comparison and Logical Instructions*

EQ

- *Operation*: check whether two values are equal
- *Format*: EQ
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*val, bool*>
- *Description*: The two operands must be listable values. The value *val* is equal to true either if the two operands have the same value and type or they are both numeric (i.e., *double, int* or *char*) and $valOp_1$ and $valOp_2$ are two arithmetically equal numbers. SP is decremented by 2.

- *Run-Time Exceptions*: If one of the stack operands is not of listable type, the exception EQ_NOT_SUPPORTED is raised.

NEQ

- *Operation*: check whether two values are not equal
- *Format*: NEQ
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*val, bool*>
- *Description*: The two operands must be listable values. The value *val* is equal to true if any of the following conditions holds: (i) the two operands have different type and one of them is not numeric (i.e., *double, int* or *char*), (ii) the two operands have the same type but different value, (ii) they are both numeric and $valOp_1$ and $valOp_2$ are two arithmetically non-equal numbers. SP is decremented by 2.
- *Run-Time Exceptions*: If one of the stack operands is not of listable type, the exception NEQ_NOT_SUPPORTED is raised.

LT

- *Operation*: check whether the first operand value is less than the second one
- *Format*: LT
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*val, bool*>
- *Description*: The two operands must be either both numeric (i.e., *double, int* or *char*) or both Boolean or both strings. The value *val* is equal to *true* if
  - the operands are of type numeric: $valOp_1 < valOp_2$ in the arithmetic precedence order for numbers, or
  - the operands are of type *bool*: $valOp_1$ is *false* and $valOp_2$ is *true*, or
  - the operands are of type *string*: $valOp_1$ precedes $valOp_2$ in the lexicographic precedence order.

  SP is decremented by 2.
- *Run-Time Exceptions*: If the stack operands are not both of type numeric or *bool* or *string*, the exception LT_NOT_SUPPORTED is raised.

LTE

- *Operation*: check whether the first operand value is less than or equal to the second one
- *Format*: LTE
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*val, bool*>
- *Description*: The two operands must be either both numeric (i.e., *double, int* or *char*) or both Boolean or both strings. The value *val* is equal to *true* if
  - the operands are of type numeric: $valOp_1 \leq valOp_2$ in the arithmetic precedence order for numbers, or
  - the operands are of type *bool*: $valOp_1$ is *false* and $valOp_2$ is *false* or *true*, or both values are *true,* or
  - the operands are of type *string*: $valOp_1$ precedes $valOp_2$ in the lexicographic precedence order or the two operands point to the same string.

  SP is decremented by 2.
- *Run-Time Exceptions*: If the stack operands are not both of type numeric or *bool* or *string*, the exception LTE_NOT_SUPPORTED is raised.

GT

- *Operation*: check whether the first operand value is greater than the second one
- *Format*: GT
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*val, bool*>
- *Description*: The two operands must be either both numeric or both Boolean or both strings. The value *val* is equal to *true* if
  - the operands are of type numeric: $valOp_1 > valOp_2$ in the arithmetic precedence order for numbers, or
  - the operands are of type *bool*: $valOp_1$ is *true* and $valOp_2$ is *false*, or
  - the operands are of type *string*: $valOp_1$ follows $valOp_2$ in the lexicographic precedence order.

  SP is decremented by 2.
- *Run-Time Exceptions*: If the stack operands are not both of type numeric or *bool* or *string*, the exception GT_NOT_SUPPORTED is raised.

GTE

- *Operation*: check whether the first operand value is greater than or equal to the second one
- *Format*: GTE
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*val, bool*>
- *Description*: The two operands must be either both numeric or both Boolean or both strings. The value *val* is equal to *true* if
  - the operands are of type numeric: $valOp_1 \geq valOp_2$ in the arithmetic precedence order for numbers, or
  - the operands are of type *bool*: $valOp_1$ is *true* and $valOp_2$ is *false* or *true*, or both values are *false,* or
  - the operands are of type *string*: $valOp_1$ follows $valOp_2$ in the lexicographic precedence order or the two operands point to the same string.

  SP is decremented by 2.
- *Run-Time Exceptions*: If the stack operands are not both of type numeric or *bool* or *string*, the exception GTE_NOT_SUPPORTED is raised.

NOT

- *Operation*: implement the not Boolean operator
- *Format*: NOT
- *Stack Operands*: …, <*valOp, typeOp*>,→ …, <*val, bool*>
- *Description*: The value *typeOp* must be *bool*. The result *val* is *true* if *valOp* is *false* or *false* otherwise. The stack operand is replaced by the Boolean result in the stack.
- *Run-Time Exceptions*: If the stack operand is not of type *bool*, the exception NOT_NOT_SUPPORTED is raised.

A4.9. *Branching Instructions*

JUMP

- *Operation*: unconditional jump to another instruction
- *Format*: JUMP *codeAddr*
- *Description*: The operand *codeAddr* must a valid address of CODE. The register IP is set equal to *codeAddr*.

JUMPZ

- *Operation*: if the stack operand is *false* (i.e., equal to 0) jump to another instruction
- *Format*: JUMPZ *codeAddr*
- *Stack Operands*: …, <*valOp, typeOp*>→
  …
- *Description*: The operand *codeAddr* must a valid address of CODE and the stack operand must be of type *bool*. The value *typeOp* must be *bool*. If *valOp* is *true* then the register IP is set equal to *codeAddr*., otherwise is left unaltered. The stack operand is removed from the stack and SP is decremented by 2.
- *Run-Time Exceptions*: If the stack operand is not of type *bool*, the exception JUMPZ_NOT_SUPPORTED is raised.
- *Remark*: The exception JUMPZ_NOT_SUPPORTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

JUMPNZ

- *Operation*: if the stack operand is *true* (i.e., equal to 1) jump to another instruction
- *Format*: JUMPZ *codeAddr*
- *Stack Operands*: …, <*valOp, typeOp*>→
  …
- *Description*: The operand *codeAddr* must a valid address of CODE and the stack operand must be of type *bool*. The value *typeOp* must be *bool*. If *valOp* is *false* then the register IP is set equal to *codeAddr*., otherwise is left unaltered. The stack operand is removed from the stack and SP is decremented by 2.
- *Run-Time Exceptions*: If the stack operand is not of type *bool*, the exception JUMPNZ_NOT_SUPPORTED is raised.
- *Remark*: The exception JUMPNZ_NOT_SUPPORTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

NEXT

- *Operation*: dummy instruction performing no operations
- *Format*: NEXT
- *Description*: The instruction does not perform any operation and simply passes the control to the next instruction.

A4.10. *List Manipulation Instructions*

SLIST

- *Operation*: start the construction of a list whose elements will be next inserted
- *Format*: SLIST
- *Stack Operands*: … →
  …, *startList, endList*
- *Description*: Two non-typed values *startList* and *endList* equal to zero (denoting the end of a list) are pushed into the list – the two values point to the first and to the last element of a new list, that is temporally empty. SP is incremented by 2.
- *Run-Time Exceptions*: If there is not enough space to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

HLIST

- *Operation*: include a first element into the list under construction
- *Format*: HLIST
- *Stack Operands*: …, *startList, endList*, <*valOp, typeOp*> →

…, *startList, endList*

- *Description*: The operand <*valOp, typeOp*> must be a listable value (i.e., *typeOp* equal to *double, int, char, bool, null, string, list, type* or *metatype*) and is added to the heap as the first element of the list under construction – say that its heap address is *h*. The values *startList* and *endList* are both set equal to *h*. SP is decremented by 2 and HP is incremented by 3, unless a triplet in the garbage list is reused.
- *Run-Time Exceptions*: If the stack operand <*valOp, typeOp*> is not a listable value, the exception HLIST_NOT_SUPPORTED is raised. If there is not enough space to extend the heap, the exception STACK_HEAP_OVERFLOW is raised.
- *Remark*: The exception HLIST_NOT_SUPPORTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

CLIST

- *Operation*: continue the list construction by appending a new element
- *Format*: CLIST
- *Stack Operands*: …, *startList, endList*, <*valOp, typeOp*> → …, *startList, endList*
- *Description*: The operand <*valOp, typeOp*> must be a listable value (i.e., *typeOp* equal to *double, int, char, bool, null, string, list, json, type* or *metatype*) and is added to the heap as the (temporally) last element of the list under construction – say that its heap address is *h*. The pointer of the previous last element as well as value *endList* are set to *h,* whereas *startList* is not modified and, therefore, it continues to point to the first element of the list. SP is decremented by 2 and HP is incremented by 3, unless a triplet in the garbage list is reused.
- *Run-Time Exceptions*: If the stack operand <*valOp, typeOp*> is not a listable value, the exception CLIST_NOT_SUPPORTED is raised. If there is not enough space to extend the heap, the exception STACK_HEAP_OVERFLOW is raised.
- *Remark*: The exception CLIST_NOT_SUPPORTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

ELIST

- *Operation*: complete the construction of a list
- *Format*: ELIST
- *Stack Operands*: …, *startList, endList* → …, <*startList, list*>
- *Description*: The value *endList* is set equal to the type *list* so that the pair of values at the stack top becomes a list-typed value with *startList* pointing to the first element of the under-construction list, which is now completed.

ALIST

- *Operation*: append an existing list to the list under the construction
- *Format*: ELIST
- *Stack Operands*: …, *startList, endList,* <*valOp, typeOp*> → …, <*startList, list*>
- *Description*: The value *typeOp* must be *list*. If the list *valOp* is not empty, the next pointer of the last element of the list under construction (pointed by *endList*) is set to address the first element of *valOp*. The typed value <*valOp, typeOp*> is removed from the stack and the pair of values at the stack top becomes a list-typed value with *startList* pointing to the first element of the under-construction list, which is now completed. SP is decremented by 2.

- *Run-Time Exceptions*: If *typeOp* is not equal to *list*, the exception LIST_EXPECTED is raised.

HEAD

- *Operation*: get the head of a list
- *Format*: HEAD
- *Stack Operands*: …, <*valOp, typeOp* > → …, <*val, valType*>
- *Description*: The value *typeOp* must be *list*. The list *valOp* must not be empty. The typed value <*valOp, typeOp*> is removed from the stack and the head element of the list is pushed into the stack.
- *Run-Time Exceptions*: If the stack operand is not of type *list*, the exception LIST_EXPECTED is raised. If the list is empty the exception EMPTY_LIST is raised.

TAIL

- *Operation*: return the tail of a list
- *Format*: TAIL
- *Stack Operands*: …, <*valOp, typeOp* > → …, <*val, list*>
- *Description*: The value *typeOp* must be *list*. The list *valOp* must not be empty. The typed value <*valOp, typeOp*> is removed from the stack. The pair <*val, list*> is pushed into the stack, where *val* is set equal to the pointer to the list tail.
- *Run-Time Exceptions*: If the stack operand is not of type *list*, the exception LIST_EXPECTED is raised. If the list is empty the exception EMPTY_LIST is raised.

LISTEL

- *Operation*: get the element of a list with a given index
- *Format*: LISTEL
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*valEl, typeEl*>
- *Description*: The value of $typeOp_1$ must be *list* and the value of $typeOp_2$ must be *int* or *char.* The value of $valOp_2$ must be in the range from 0 to *n*-1, where *n* is the list length. The list element with index $valOp_2$ is copied into <*valEl, typeEl*>. The two operands are removed from the stack and the element of the list $valOp_1$ with the index $valOp_2$ is pushed into the stack.
- *Run-Time Exceptions*: If $typeOp_1$ is not equal to *list*, the exception LIST_EXPECTED is raised. If $typeOp_2$ is not equal to *int* or *char*, the exception INT_EXPECTED is raised. If $valOp_2$ is negative, the exception NEGATIVE_LIST_INDEX is raised. If $valOp_2$ is greater than *n*-1, where *n* is the list length, the exception LIST_OUT_BOUND is raised. SP is decremented by 2.
- *Remark*: LISTEL with $valOp_2 = 0$ has the same effect as HEAD, but it is less efficient.

MODVEL

- *Operation*: modify the head element of a list
- *Format*: MODVEL
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …
- *Description*: The value of $typeOp_1$ must be *list* and the value of $typeOp_2$ must be listable (i.e., equal to *double, int, char, bool, string, list, type* or *metatype*). The list $valOp_1$ must

not be empty. The two operands are removed from the stack and the head element of the list $valOp_1$ is modified into <$valOp_2$, $typeOp_2$>. SP is decremented by 4.

- *Run-Time Exceptions*: If $typeOp_1$ is not equal to *list*, the exception LIST_EXPECTED is raised. If $typeOp_2$ is not listable, the exception MOVEL_NOT_SUPPORTED is raised. If $valOp_1$ is negative, the exception EMPTY_LIST is raised.
- *Remark*: The exceptions LIST_EXPECTED and MODVEL_NOT_SUPPORTED are unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

LCLONE

- *Operation*: clone the elements of a given list from a given index on or from a first index to a second index (excluded)
- *Format*: LCLONE *k*
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> [, <$valOp_3$, $typeOp_3$>] → …, <*val*, *list*>
- *Description*: The instruction operand *k* can be either 0 or 1 and the third stack operand is present only if $k = 1$. The value of $typeOp_1$ must be *list* and the values of $typeOp_2$ and of $typeOp_3$ as well, if present, must be *int* or *char.* The values of $valOp_2$ and of $valOp_3$ (if present) must be non-negative. If $k = 0$, assume that $valOp_3 = n$, where n is the list length. There are three possibilities:
  - If $valOp_2 < valOp_3 < n$, where n is the list length, a new list is constructed in the heap by copying all the list elements from the index $valOp_2$ on until the index $valOp_3 - 1$ – *val* is set equal to the address of the first element of the new list;
  - if $valOp_2 < n$ and $valOp_3 \geq n$, a new list is constructed in the heap by copying all the list elements from the index $valOp_2$ on until the end of the list – *val* is set equal to the address of the first element of the new list;
  - otherwise (i.e., $valOp_2 \geq n$ or $valOp_2 \geq valOp_3$) *val* is set to *endList* (i.e., 0), i.e., the empty list is returned.

  SP is decremented by 2 if $k = 0$ or by 4 otherwise. HP is at most incremented by $3 \times n$ as some triplets in the garbage list can be reused..
- *Run-Time Exceptions*: If $typeOp_1$ is not equal to *list*, the exception LIST_EXPECTED is raised. If $typeOp_2$ or $typeOp_3$ is not equal to *int* or *char*, the exception INT_EXPECTED is raised. If $valOp_2$ or $valOp_3$ is negative, the exception NEGATIVE_LIST_INDEX is raised. If there is not enough space to extend the heap, the exception STACK_HEAP_OVERFLOW is raised.

A4.11. *String Manipulation Instructions*

SSTRING

- *Operation*: start the construction of a string whose characters will be next inserted
- *Format*: SSTRING
- *Stack Operands*: … → …
- *Description*: The heap area for storing the string is prepared by allocating a cell for storing the string size, which is initialized to 0. HP is therefore incremented by 1.
- *Run-Time Exceptions*: If there is not enough space to extend the heap, the exception STACK_HEAP_OVERFLOW is raised.

CSTRING

- *Operation*: continue the string construction by appending a new character
- *Format*: CSTRING
- *Stack Operands*: …, <*valOp, typeOp*> →
  …
- *Description*: The value of *typeOp* must be equal to *char*. The character *valOp* is added to the structure in the heap storing the string under construction and the string size is increased by 1. SP is decremented by 2 and HP is incremented by 1.
- *Run-Time Exceptions*: If *typeOp* is not equal to *char*, the exception CHAR_EXPECTED is raised.
- *Remark*: The exception CHAR_EXPECTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

ESTRING

- *Operation*: end the construction of a string
- *Format*: ESTRING
- *Stack Operands*: … →
  …, <*val*, *string*>
- *Description*: The construction is ended. There are two possibilities:
  - the string is not already stored in the string pool: then a new cell is added to the heap structure storing the string, which implements the list of all strings in the pool. Then HP is incremented by 1 and *val* is set equal to starting address of the heap structure storing the new string;
  - the string is already in the pool: HP is decremented by 1+*k*, where *k* is the size of the string under construction, i.e. the area storing the string under construction is made available for future reuse; *val* is set equal to the starting address of the heap structure storing the same string in the pool.

  SP is incremented by 2.
- *Run-Time Exceptions*: If there is not enough space to extend the stack, the exception STACK_HEAP_OVERFLOW is raised.

STRINGEL

- *Operation*: get the character of a string with a given index
- *Format*: STRINGEL
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> →
  …, <*valEl, char*>
- *Description*: The value of $typeOp_1$ must be *string* and the value of $typeOp_2$ must be *int* or *char.* The value of $valOp_2$ must be in the range from 0 to *n*-1, where *n* is the string length. The string character with index $valOp_2$ is copied into *valEl*. The two operands are removed from the stack and the element of the list $valOp_1$ with the index $valOp_2$ is pushed into the stack. SP is decremented by 2.
- *Run-Time Exceptions*: If $typeOp_1$ is not equal to *string*, the exception STRING_EXPECTED is raised. If $typeOp_2$ is not equal to *int* or *char*, the exception INT_EXPECTED is raised. If $valOp_2$ is negative, the exception NEGATIVE_STRING_INDEX is raised. If $valOp_2$ is greater than *n*-1, where *n* is the string length, the exception STRING_OUT_BOUND is raised.

SUBSTR

- *Operation*: construct a new string composed by the characters of a string from a given index on or from a first index to a second index (excluded)
- *Format*: SUBSTR *k*

- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> [, <$valOp_3$, $typeOp_3$>] → …, <*val, string*>
- *Description*: The instruction operand *k* can be either 0 or 1 and the third stack operand is present only if *k* = 1. The value of $typeOp_1$ must be *string* and the values of $typeOp_2$ and of $typeOp_3$ as well, if present, must be *int* or *char.* The values of $valOp_2$ and of $valOp_3$ (if present) must be non-negative. If *k* = 0, assume that $valOp_3$ = *n*, where *n* is the string length. There are three possibilities:
  - If $valOp_2$ < $valOp_3$ < *n*, where *n* is the string length, a new string is constructed in the heap by copying all the string characters from the index $valOp_2$ on until the index $valOp_3$ – 1;
  - if $valOp_2$ < *n* and $valOp_3$ ≥ *n*, a new string is constructed in the heap by copying all the string characters from the index $valOp_2$ on until the end of the string;
  - otherwise (i.e., $valOp_2$ ≥ *n* or $valOp_2$ ≥ $valOp_3$) *val* is set to the starting heap address of the empty string.

  The value of *val* is set equal to the address of the first element of the new string – the new string is actually added only if it does not already exist in the pool. SP is decremented by 2 if *k* = 0 or by 4 otherwise. HP is either incremented by 2 + *m*, where *m* is the length of the new string, or is left unmodified otherwise.
- *Run-Time Exceptions*: If $typeOp_1$ is not equal to *list*, the exception STRING_EXPECTED is raised. If $typeOp_2$ or $typeOp_3$ is not equal to *int* or *char*, the exception INT_EXPECTED is raised. If $valOp_2$ or $valOp_3$ is negative, the exception NEGATIVE_LIST_INDEX is raised. If there is not enough space to extend the heap, the exception STACK_HEAP_OVERFLOW is raised.

STRIND

- *Operation*: given two strings S and T and an index i, return the index within the string S of the first occurrence of the string T, starting at the index i or -1 if T does not occur in S, starting at i.
- *Format*: STRIND
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$>, <$valOp_3$, $typeOp_3$> → …, <*val, int*>
- *Description*: The value of $typeOp_1$ and of $typeOp_2$ must be *string* and the value of $typeOp_3$ must be *int* or *char.* The values of $valOp_3$ must be non-negative. There are two possibilities:
  - If the string $valOp_2$ occurs in the string $valOp_1$, starting at the index $valOp_3$, *val* is set equal to the index of the first occurrence of the string T, starting at this index; *n*, where *n* is the string length, a new string is constructed in the heap by copying all the string characters from the index $valOp_2$ on until the index $valOp_3$ – 1;
  - If the string $valOp_2$ does not occur in the string $valOp_1$, starting at the index $valOp_3$, *val* is set equal to -1.

  SP is decremented by 4.
- *Run-Time Exceptions*: If $typeOp_1$ or $typeOp_2$ is not equal to *string*, the exception STRING_EXPECTED is raised. If $typeOp_3$ is not equal to *int* or *char*, the exception INT_EXPECTED is raised. If $valOp_3$ is negative, the exception NEGATIVE_STRING_INDEX is raised.

A4.12. *Json Manipulation Instructions*

SJSON

- *Operation*: start the construction of a json by inserting an initial null field into the stack
- *Format*: SJSON
- *Stack Operands*: … →
  …, *firstFieldAddr*, *lastFieldAddr*
- *Description*: A null (i.e., fictitious) field is added into the heap that will represent the first field of the json under construction – this field is represented as <*nullValue, field, nullNextAddress*>, stating that is a null field without a next field following it. The address of the null field is copied in both *firstFieldAddr* and *lastFieldAddr,* i.e., at this stage, the json starts and ends with the unique null field. SP is incremented by 2 and HP is incremented by 3 unless a triplet in the garbage list is reused.
- *Run-Time Exceptions*: If there is not enough space to extend the memory (stack and/or heap), the exception STACK_HEAP_OVERFLOW is raised.

CJSON

- *Operation*: continue the json construction by appending a new field
- *Format*: CJSON
- *Stack Operands*: … , *firstFieldAddr*, *lastFieldAddr*, <*valOp, field*> →
  …, *firstFieldAddr*, *lastFieldAddr*
- *Description*: The triplet <*valOp, field, nullNextAddress*> is added as last field of the json by linking the previous last field to it and update the value of *lastFieldAddr* by setting it to the address of the new triplet. On the other hand, *firstFieldAddr* remains unchanged. json is constructed into the heap that consists of a unique field, which is fictitious (i.e., a null field). SP is decremented by 2 and HP is incremented by 3 unless a triplet in the garbage list is reused.
- *Run-Time Exceptions*: If the second stack operand is not a field, the exception the exception CJSON_NOT_SUPPORTED is raised. If the json contains a field with the same key as the new field, the exception DUPLICATED_KEY is raised. If there is not enough space to extend the memory (stack and/or heap), the exception STACK_HEAP_OVERFLOW is raised.

EJSON

- *Operation*: end the construction of a json
- *Format*: EJSON
- *Stack Operands*: … *firstFieldAddr*, *lastFieldAddr* →
  …, <*firstFieldAddr, json*>
- *Description*: The construction of the json pointed by *firstFieldAddr* is completed by simply replacing *lastFieldAddr* (the pointer to the last field) with the type value *json*.

NEWFLD

- *Operation*: construct a new field
- *Format*: NEWFLD
- *Stack Operands*: … <*keyVal, string*>, <*fieldVal, fieldType*>→
  …, <*keyValAddr, field*>
- *Description*: The operand <*fieldVal, fieldType*> must be a listable value (i.e., *typeOp* equal to *double, int, char, bool, null, string, list, json, type* or *metatype*). First a new Key-Value element is constructed in the heap as the triplet: <*fieldVal, fieldType, keyVal*>, say with address *keyValAddr*. Then a new field is constructed into the heap as a triplet: <*keyValAddr*, *field*, *nullPointer*>. SP is decremented by 2 and HP is incremented by 6 unless some triplets in the garbage list are reused.

- *Run-Time Exceptions*: If <*fieldVal, fieldType*> is not a listable value, the exception NEWFLD_NOT_SUPPORTED is raised. If the first stack operand is not equal to the type value *string*, then the exception STRING_EXPECTED is raised. If there is not enough space to extend the heap, the exception STACK_HEAP_OVERFLOW is raised.

FLDVAL

- *Operation*: replace the field on top of the stack with its value
- *Format*: FIELDVAL
- *Stack Operands*: …, <*keyValAddr, field*> → …, <*resVal, resType*>
- *Description*: If the stack operand <*keyValAddr, field*> is a null field (i.e., *keyValAddr* = 0*)*, then <*resVal, resType*> is set to the null value, i.e., <0, *null*>. Otherwise, the Key-Value pointed by *keyValAddr*is, say <*resVal, resType, keyAddr*> is retrieved from the heap and the field value <*resVal, resType*> is inserted into the stack.
- *Run-Time Exceptions*: If the stack operand is not a field, the exception FIELD_EXPECTED is raised.

FLDFIND

- *Operation*: given a json and a key on top of the stack, replace them with the json field corresponding to that key if it exists or otherwise with the null field or with a new field with a null value, that is added to the json
- *Format*: FLDFIND *k*
- *Stack Operands*: …, <*jsonAddr, json*>, <*stringAddr, string*> → …, <*keyValAddr, field*>
- *Description*: The instruction operand *k* can be either 0 or 1. The fields of the json (first stack operand) are scanned for finding a field with key equal to the second stack operand. If such a field is found then this field, say <*keyValAddr, field*>, is put in the stack. Otherwise, there are two possibilities:
  - if *k* = 0, <*keyValAddr, field*> is the the null field, or
  - if *k* = 1, <*keyValAddr, field*> is a new field of the json that points to the key value <0*, null, stringAddr*>, say with address *keyValAddr.*

  SP is decremented by 2. If *k* = 1 and the field is not found, HP is incremented by 3 unless a triplet in the garbage list is reused.
- *Run-Time Exceptions*: If the first stack operand is not a json, the exception JSON_EXPECTED is raised. If the second stack operand is not a string, the exception STRING_EXPECTED is raised. If there is not enough space to extend the heap, the exception STACK_HEAP_OVERFLOW is raised.

JCLONE

- *Operation*: clone the json on top of the stack
- *Format*: JCLONE
- *Stack Operands*: …, <*opVal, json*> → …, <*resVal, json*>
- *Description*: A new json <*resVal, json*> is constructed into the heap by copying all the fields (as well as the associated key-values) of the json <*opVal, json*>. Recall that *resVal* points to the list of the new fields. SP is decremented by 2. HP is incremented by $3 \times n + 3 \times m$, where *n* is the number of fields in *opVal* and *m* is the number of such fields that are null – the increase of HP may be smaller if some triplets in the garbage list are reused.
- *Run-Time Exceptions*: If the stack operand is not a json, the exception JSON_EXPECTED is raised. If there is not enough space to extend the heap, the exception STACK_HEAP_OVERFLOW is raised.

A4.13. *Generic Manipulation Instructions for Lists or Strings or Jsons*

LEN

- *Operation*: compute the length of a list or a string or a json
- *Format*: LEN
- *Stack Operands*: …, <*valOp, typeOp*> → …, <*val, int*>
- *Description*: The stack operand must be of type *list* or *string* or *json*. The integer *val* is equal to the length *n* of *valOp*, which is equal (1) to the number of list elements if *typeOp* = *list* or (2) to the number of fields following the initial fictitious field if *typeOp* = *list* or (3) to the length stored in the string heap storage if *typeOp* = *string* .
- *Run-Time Exceptions*: If the stack operand is neither *list* or *string or json*, the exception LEN_NOT_SUPPORTED is raised.

HDFLDV

- *Operation*: return the head value of a list or the value of a field
- *Format*: HDFLDV
- *Stack Operands*: …, <*valOp, typeOp*> → …, <*valRes, typeRes*>
- *Description*: There are two possible cases:
  1. $typeOp_1$ = *list*: the instruction is implemented in the same way as the instruction HEAD,
  2. $typeOp_1$ = *field*: the instruction is implemented in the same way as the instruction FLDVAL.
- *Run-Time Exceptions*: If the stack operand type is neither *list* nor *field*, the exception LIST_FIELD_EXPECTED is raised.
- *Remark*: When the stack operand type is known, it is convenient to directly use the instruction HEAD or FLDVAL. This instruction is used by CalcuList while compiling assignments such as "{! X[Y][Z] = <*rhs*>}", for which is not yet known whether "X[Y]" is a list or a field that must be further inspected by "[Z]" – note that it excluded that it is a string since a string is immutable and, therefore, cannot arise in the left hand side of an assignment. By assuming that X, Y and Z are the arguments 3,2 and 1, respectively, the target code is: "LOADARG 3; DEREF; LOADARG 2; DEREF; LTLSEL; HDFLDV; LOADARG 1; DEREF; …".

LJEL

- *Operation*: return an element of a list with a given index or the value of a field with a given key
- *Format*: LJEL
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*valRes, typeRes*>
- *Description*: There are two possible cases:
  1. $typeOp_1$ = *list*: the instruction is implemented in the same way as the instruction LISTEL,
  2. $typeOp_1$ = *json*: the instruction is implemented in the same way as the instruction FLDFIND 1.
- *Run-Time Exceptions*: If $typeOp_1$ is neither *list* nor *field*, the exception LIST_FIELD_EXPECTED is raised.
- *Remark*: When the stack operand type is known, it is convenient to directly use the instruction LISTEL or FLDFIND 1. This instruction is used by CalcuList while compiling assignments such as "{! X[Y][Z] = <*rhs*>}" – see the previous remark.

LSJCLONE

- *Operation*: return a copy of a list or json or string – the copy is cloned for the case of list or json
- *Format*: LSJCLONE
- *Stack Operands*: …, <*valOp, typeOp*>, → …, <*valRes, typeRes*>
- *Description*: There are three possible cases:
  1. *typeOp* = *list*: the instruction is implemented in the same way as the sequence of two instructions: "PUSHI 0; LCLONE 0"
  2. *typeOp* = *json*: the instruction is implemented in the same way as the instruction JCLONE.
  3. *typeOp* = *string*: <*valRes, typeRes*> is set equal to <*valRes, typeRes*>, i.e., the address of the operand string is returned.
- *Run-Time Exceptions*: If *typeOp* is neither *list* nor *json* nor *string*, the exception LIST_STRING_JSON_EXPECTED is raised.
- *Remark*: When the stack operand type is known, it is convenient to directly use the equivalent specific instructions. This instruction is used by CalcuList while compiling espressions such as "X[:]" inside a function where the type of X is unknown (list, json or string). By assuming that X is the first function argument, the target code for the example is: "LOADARG 1; DEREF; LSJCLONE; …".

LSCLONE

- *Operation*: construct a sublist or a substring whose elements are within given indices – the copy is cloned for the case of list
- *Format*: LSCLONE *k*
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> [, <$valOp_3$, $typeOp_3$>] → …, <*valRes, typeRes*>
- *Description*: The instruction operand *k* can be either 0 or 1 and the third stack operand is present only if *k* = 1. The value of $typeOp_1$ must be *list* or *string* and the values of $typeOp_2$ and of $typeOp_3$ as well, if present, must be *int* or *char.* The values of $valOp_2$ and of $valOp_3$ (if present) must be non-negative. There are two possible cases:
  1. $typeOp_1$ = *list*: the instruction is implemented in the same way as the instruction instructions: "LCLONE *k*"
  2. $typeOp_1$ = *string*: the instruction is implemented in the same way as the instruction SUBSTR *k*.
- *Run-Time Exceptions*: If *typeOp* is neither *list* nor *string*, the exception LIST_STRING_EXPECTED is raised.
- *Remark*: When the stack operand type is known, it is convenient to directly use the equivalent specific instructions. This instruction is used by CalcuList while compiling espressions such as "X[2:4]" inside a function where the type of X could be either list or string. By assuming that X is the first function argument, the target code for the example is: "LOADARG 1; DEREF; PUSHI 2; PUSHI 4; LSCLONE 1; …".

LSELV

- *Operation*: get the character of a string or an element of a list with a given index
- *Format*: LSELV
- *Stack Operands*: …, <$valOp_1$, $typeOp_1$>, <$valOp_2$, $typeOp_2$> → …, <*valRes, typeRes*>

- *Description*: The value of *typeOp*$_1$ must be *list* or *string*, *typeOp*$_2$ must be equal to *int* or *char* and *valOp*$_2$ must be in the range from 0 to *n*-1, where *n* is the string or list length. There are two possible cases:
    1. *typeOp*$_1$ = *list*: the instruction is implemented in the same way as the sequence of two instructions: "LISTEL; HEAD"
    2. *typeOp*$_1$ = *string*: the instruction is implemented in the same way as the instruction STRINGEL.
- *Run-Time Exceptions*: If *typeOp*$_1$ is not equal to *list* or *string*, the exception LIST_STRING_EXPECTED is raised.
- *Remark*: When the stack operand type is known, it is convenient to directly use the equivalent specific instructions. This instruction is used by CalcuList while compiling espressions such as "X[2]" inside a function where the type of X could be either list or string. By assuming that X is the first function argument, the target code for the example is: "LOADARG 1; DEREF; PUSHI 2; LSELV; …".

LSJELV

- *Operation*: get the character of a string with a given index or an element of a list with a given index or the value of a json field with a given key
- *Format*: LSJELV
- *Stack Operands*: …, <*valOp*$_1$, *typeOp*$_1$>, <*valOp*$_2$, *typeOp*$_2$> → …, <*valRes*, *typeRes*>
- *Description*: The value of *typeOp*$_1$ must be *list* or *string* or *json.* There are three possible cases:
    1. *typeOp*$_1$ = *list*: *typeOp*$_2$ must be equal to *int* or *char* and the instruction is implemented in the same way as the sequence of two instructions: "LISTEL; HEAD"
    2. *typeOp*$_1$ = *string*: *typeOp*$_2$ must be equal to *int* or *char* and the instruction is implemented in the same way as the instruction STRINGEL
    3. *typeOp*$_1$ = *json*: *typeOp*$_2$ must be equal to *string* and the instruction is implemented in the same way as the sequence of two instructions: "FLDFIND; FLDVAL".
- *Run-Time Exceptions*: If *typeOp*$_1$ is not equal to *list* or *string* or *json*, the exception LIST_STRING_JSON_EXPECTED is raised.
- *Remark*: When the stack operand type is known, it is convenient to directly use the equivalent specific instructions. This instruction is used by CalcuList while compiling espressions such as "X[i]" inside a function where the type of X could be either list or string or json and the type of i could be either int or char. By assuming that X and i are respectively the first and second function arguments, the target code for the example is: "LOADARG 2; DEREF; LOADARG 1; DEREF; LSJELV; …".

A4.14. *Print Instruction*

PRINT

- *Operation*: print a listable value on top of OUTPUT
- *Format*: PRINT *format*
- *Stack Operands*: … <*valOp, typeOp*> → …
- *Output*: … → …, <*valOut, typeOut*>

- *Description*: The instruction operand *format* is equal to 0 or greater than 0. The stack operand must be a listable value, i.e., *typeOp* must be equal to equal to *double, int, char, bool, null, string, list, json, type* or *metatype.* The value of *valOP* is copied into *valOut*. If *format* = 0 then also *typeOP* is copied into *typeOut.* Otherwise, the type printed in output is modified as follows:
    - if *typeOp* = *string* then *typeRes* is set to *stringQ* (i.e., the string will be printed between double quotes);
    - if *typeOp* = *char* then *typeOut* is set to *charQ* (i.e., the caracter will be printed between single quotes);
    - if *typeOp* = *null* then *typeOut* is set to *nullQ* (i.e., the string *null* will be printed, whereas in general the null value is not printed);
    - if *typeOp* = *list* or *json* then *typeOut* is set to *listQ* or *jsonQ,* respectively (i.e., the list or json will be printed in a so called "pretty format", which prescribes indentation of elements and fields);
    - in all other cases *typeOut* is set equal to *typeOp*.

  SP is decremented by 2 and OP is incremented by 2.
- *Run-Time Exceptions*: If *typeOp* is different from *double, int, char, bool, null, string, list, json, type* or *metatype,* the exception PRINT_NOT_SUPPORTED is raised. If there is not enough space to extend the output, the exception OUTPUT_OVERFLOW is raised.
- *Remark*: Any element of a list or of a json with type *string*, *char* or *null* will be always printed using the format *stringQ, charQ* or *nullQ,* respectively. The exception PRINT_NOT_SUPPORTED is unexpected for a program compiled by CalcuList. In this case, please report the whole printed message and the CalcuList session while it occurred.

A.5. The CLVM Assembler

A5.1. *Structure and Grammar of CLVM Assembler*

The CLVM machine provides an assembler environment (*CLVM_ASS*) to write CLVM programs using an assembler language. When a session of CLVM_ASS is started, the user is required to enter the name of a text file containing one or more assembler programs – the default extension for such files is "*clm*". Then, each assembler program will be syntactically checked and will be eventually transformed into a CLVM program (object code) that will be launched for the execution on the CLVM machine. The programs stored in the same file share the same memory so that global variables and heap values stored by a program can be used by subsequent ones.

An assembler program is the symbolic description of a CLVM program and, therefore, consists of a main unit possibly followed by a number of function units. The text lines of the program (called *AsmLines*) are of the following three types:

1. *AsmBeginEnd*: the line contains the command "$BEGIN", denoting the start of a program, or the command $END", denoting its end – the two commands will not be included into the CLVM object code;
2. *AsmInstr*: the line contains the symbolic description of a CLVM instruction with the following format: an optional unsigned integer label, the operator symbolic name and the possible operand expressed as a string representing a positive integer, a negative integer or a double, according to the type supported by the operator – the assembler instruction will be transformed into a CLVM instruction;
3. *AsmBlank*: the line only contains spaces and will not be included into the CLVM object code.

The end of line is either the line feed character "\n" or the character "/", i.e., all characters following "/" are skipped for they are used as a comment. To represent the grammar for text file for CLVM_ASS, we adopt the following notation:

- *nonterminal-symbols* are strings enclosed between brackets while *terminal symbols* are enclosed between apices;
- the terminal symbols are: 'A', ..., 'Z', '0', ..., '9', '+', '-', '.' and the space character, denoted by *sp* – as letters are all converted to upper cases, the assembler language is not case sensitive;
- a *grammar rule* is represented as A : α, where A is nonterminal-symbol and α is a *regular expression* of nonterminal and terminal symbols, represented in the EBNF (Extended Backus-Naur Form) format, using the following additional meta-symbols: the square brackets indicate that a sub-expression is optional, curly braces denote the Kleene closure, indicating that a sub-expression may appear zero or more times, the symbol '|' denotes the union of two sub-expressions delimited by parenthesis;
- the axiom is the meta-symbol <asm_file>.

The grammar rules of the CLVM assembler language are given next:

<asm_file> /* *a sequence of programs, possibly separated by blanks* */
: <asm_blank> { <asm_blank> | <asm_program> }
| <asm_program> { <asm_blank> | <asm_program> }

<asm_blank> /* *a line of all spaces – possibly empty* */
: { *sp* } <le>

<asm_program> /* *a program is a sequence of instructions enclosed by marking lines (begin and end), possibly separated by blank lines* */
: <asm_begin> { (<asm_instr> | <asm_blank>) } <asm_end>

<asm_begin> /* *line starting a program* */
: { *sp* } "$BEGIN" {*sp*} <le>

<asm_end> /* *line ending a program* */
: { *sp* } "$END" {*sp*} <le>

<asm_instr> /* *operation, possibly preceded by a label* */
: { *sp* } [<label>] <oper> { *sp* } <le>

<oper> /* *four types of operations* */
: <op0> /* *no operand* */
| <op1> *sp* { *sp* } <int> /* *non-negative integer operand* */
| <op2> *sp* { *sp* } [<sign>] <int> /* *integer operand* */
| <op3> *sp* { *sp* } <double> /* *double operand* */

<label> /* *a non-negative integer as instruction label* */
: <int> *sp* { *sp* }

<le> /* *end of a line: new line character or start of a comment* */
: '\n' | '/'

<int> /* *non-negative integer* */
: ('0' | ... | '9') { ('0' | ... | '9') }

<sign> /* *positive or negative sign* */
: '+'
| '-'

<double> /* *double in the floating point format* */
: [<sign>] (<int> ['.' [<int>]] | "." <int>) ['E' [<sign>] <int>]

```
<op0>          /* operations with no operand */
               : "HALT" /* end the execution of a program */
               | "CALLS" /* call a function whose address in on the stack */
               | "THROWE" /* output a string and end the program */
               | "DUPL" /* duplicate the pair val-type on top of the stack */
               | "POP" /* remove the pair val-type on top of the stack */
               | "PUSHMT" /* push the value "type" with type "metatype" */
               | "PUSHN" /* push the null value on top of the stack */
               | "DEREF" /* replace a memory address with its content */
               | "MODV" /* store the value on top of the stack into the memory area
                           addressed by the preceding value on top of the stack */
               | "ADD" /* add the two values on top of the stack */
               | "SUB" /* subtract the two values on top of the stack */
               | "MULT" /* multiply the two values on top of the stack */
               | "DIV" /* divide the two values on top of the stack */
               | "NEG" /* change sign to the value on top of the stack */
               | "TOINT" /* cast to int */
               | "TOCHAR" /* cast to char */
               | "TOTYPE" /* cast to type */
               | "TOSTRING" /* cast to string */
               | "TOLIST" /* cast to list */
               | "TOJSON" /* cast to json */
               | "TUPLE" /* return the list of values of the json on top of the stack */
               | ""RAND" /* push a random number to the stack */
               | "EXP" /* compute e^v, where e is the Euler's number and
                          v is the value on top of the stack */
               | "LOG" /* compute log_e(v), where e is the Euler's number and
                          v is the value on top of the stack */
               | "POW" /* compute v2^v1, where v2 and v1 are
                          the values on top of the stack */
               | "REMAIN" /* compute v2 % v1, i.e., their remainder */
               | "EQ" /* return true if v2 is equal to v1 */
               | "NEQ" /* return true if v2 is not equal to v1 */
               | "LT" /* return true if v2 < v1 */
               | "LTE" /* return true if v2 ≤ v1 */
               |"GT" /* return true if v2 > v1 */
               | "GTE" /* return true if v2 ≥ v1 */
               | "NOT" /* invert true and false on topo of the stack */
               | "NEXT" /* dummy instruction */
               | "SLIST" /* start the construction of a list */
               | "HLIST" /* add the first element to the list */
               | "CLIST" /* add an additional element to the list */
               | "ELIST" /* end the construction of the list */
               | "ALIST" /* append a list to the last element of a list */
               | "HEAD" /* return the valued of the list head */
               | "TAIL" /* return the sublist starting from the index 1 */
               | "SUBTAIL" /* return the sublist starting from an index i */
               | "LISTEL" /* get the element of a list with a given index */
               | "MODVEL" /* modify the head element of a list */
               | "SSTRING" /* start the construction of a string */
```

| “CSTRING” /* *continue the string construction,*
one char at the time */
| “ESTRING” /* *end the construction of the string* */
| “STRINGEL” /* *get the character of a string with a given index* */
| “STRIND” /* *find the start of a substring within a string* */
| “SJSON” /* *start the construction of a json* */
| “CJSON” /* *continue the construction of a json, by adding a field* */
| “EJSON” /* *end the construction of a json* */
| “NEWFLD” /* *construct a new field* */
| “FLDVAL” /* *replace the field on top of the stack with its value* */
| “JCLONE” /* *clone the json on top of the stack* */
| “LEN” /* *compute the length of a list or a string or a json* */
| “HDFLDV” /* *return the head value of a list or the value of a field* */
| “LJEL” /* *return an element of a list with a given index or*
the value of a field with a given key */
| “LSJCLONE” /* *return a copy of a list or json or string –*
the copy is cloned for the case of list or json/
| “LSELV” /* *return the character of a string or*
an element of a list with a given index */
| “LSJELV” /* *return the character of a string or an element of a list*
or the value of a json field */

<op1> /* *operations with one non-negative integer operand* */
: “INIT” /* *start the main unit* */
| “START” /* *start a function unit* */
| “RETURN” /* *end a function unit* */
| “CALL” /* *call a function unit* */
| “PUSHC” /* *push a char into the stack* */
| “PUSHB” /* *push a bool into the stack* */
|“PUSHT” /* *push a type into the stack* */
| “NULLIFY” /* *remove the elements of a list or the fields of a json* */
| “LOADGV” /* *push the address of a global variable* */
| “LOADLV” /* *push the address of a local variable* */
| “LOADARG” /* *push the address of a function argument* */
| “JUMP” /* *unconditional jump to another instruction* */
| “JUMPZ” /* *conditional jump (if false) to another instruction* */
| “JUMPNZ” /* *conditional jump (if true) to another instruction* */
| “LCLONE” /* *clone the elements of a list within a given range* */
| “SUBSTR” /* *construct a new string composed by the characters*
of a string within a given range */
| “FLDFIND” /* *find a json field with a given key* */
| “LSCLONE” /* *construct a sublist or a substring –*
the copy is cloned for the case of list */
| “PRINT” /* *add the value on top of the stack into OUTPUT* */

<op2> /* *operations with one, possibly negative, integer operand* */
: “PUSHI” /* *push an integer into the stack* */
| “PUSHF” /* *push the address to a function into the stack –*
negative if the function is lambda */

<op3> /* *operations with one double operand* */
: “PUSHD” /* *push a double into the stack* */

As all grammar rules are not recursive, it turns out that the CLVM assembler language is regular. Actually, the rules only define a regular approximation of the language, which is instead context sensitive because the operand of an instruction JUMP, JUMPZ or JUMNPNZ must be a natural number that occurs as a label of some other instruction in the program. As shown in the next section, this check will be performed by suitable semantic actions that also tranform the jump operand into the actual address of the referenced instruction. There is also a static semantic action to verify that labels are not duplicated inside the same program – labels of different programs in the same file may be duplicated.

An example of a text file for CLVM_ASS is shown next. The file contains two assembler programs:

- the first program defines two global variables, constructs the heterogeneous list [-2, 'H', int, 3.14] and the string "CLVM" and stores them into the first and the second global variable, respectively; in addition, both the list and the string are printed into separated lines by means of an intermediate printing of a linefeed; the string is actually printed twice into two different formats: with and without quotes;
- the second program uses the global variables defined by the previous program and takes the list [-2, 'H', int, 3.14] stored in the first one to compute and print the sum of all elements with type *int* or *double* (thus the result 1.14 is printed); the sum is performed by a first function unit (with one argument and no local variables) that recursively scans the list elements and sums those whose type is *int* or *double*; the later check is done by the second function unit having one argument (the list element whose type is to be checked) and a local variable where the list element type is stored once for all before checking that it is equal to *int* or *double*.

The content of whole file, named *Example_A_5_3_Tutorial.cl*, is reported below – the lines are suitable commented to increase their readability:

```
/ Example of Assembler text file for Section A.5.3

/ ** First Program
        $BEGIN
        INIT        2     / Program with 2 global variables not yet initialized
        LOADGV      0     / load the address of the first global variable
        SLIST             / start a new list
        PUSHI       -2    / push the integer -2 into the stack
        HLIST             / -2 becomes the head of the new list
        PUSHC       72    / push the character 'H' into the stack
        CLIST             / 'H' becomes the second element of the new list
        PUSHT       2     / push the type int into the stack
        CLIST             / int becomes the third element of the new list
        PUSHD       3.14  / push the double 3.14 into the stack
        CLIST             / 3.14 becomes the fourth element of the new list
        ELIST             / end of the new list [-2, 'H', int, 3.14]
        MODV              / the new list is assigned to the first global variable
        LOADGV      0     / load the address of the first global variable
        DEREF             / replace the address with the value (i.e. the list)
        PRINT       0     / the list [-2, 'H', int, 3.14] is printed
        LOADGV      1     / load the address of the second global variable
        SSTRING           / start e new string
        PUSHC       67    / push the character 'C' into the stack
        CSTRING           / 'C' become the first character of the new string
        PUSHC       76    / push the character 'L' into the stack
        CSTRING           / 'L' become the second character of the new string
        PUSHC       86    / push the character 'V' into the stack
        CSTRING           / 'V' become the third character of the new string
```

```
        PUSHC     77   / push the character 'M' into the stack
        CSTRING        / 'M' become the second character of the new string
        ESTRING        / end of the new string
        MODV           / the new string is assigned to the first global variable
        PUSHC     10   / the character Line Feed (LF) is pushed into the stack
        PRINT     0    / LF is printed to insert a line break in the output
        LOADGV    1    / load the address of the second global variable
        DEREF          / replace the address with the value (i.e. the string)
        PRINT     1    / the string is printed in a quoted format: "CLVM"
        PUSHC     10   / the character Line Feed (LF) is pushed into the stack
        PRINT     0    / LF is printed to insert a line break in the output
        LOADGV    1    / load the address of the second global variable
        DEREF          / replace the address with the value (i.e. the string)
        PRINT     0    / the string is printed without quotes: CLVM
        HALT           / terminate the program
        $END

/ ** Second Program
        $BEGIN
        INIT      2    / Program with 2 global variables initialized by the first program
        LOADGV    0    / load the address of the first global variable (the list)
        DEREF          / replace the address with the list [-2, 'H', int, 3.14]
        CALL      60   / call the function unit 1 (FU 1) whose first instruction has label 60
        PRINT     0    / print the result returned by the function unit: 1.14
        HALT           / terminate the program
        / FUNCTION UNIT 1 (FI 1) (with 0 local variables and 1 argument)
60      START     0    / start the FU 1
        LOADARG   1    / load the address of unit's unique argument
        DEREF          / replace the address with its value (a list)
        SLIST          / start a new list
        ELIST          / end the list - empty list
        EQ             / check whether the first list operand is equal to the empty list
        JUMPZ     160  / if not, go to the instruction with label 150
        PUSHI     0    / push the integer 0 into the stack
        RETURN    1    / terminate FU 1 (having 1 argument) by returning the value 0
160     LOADARG   1    / load the address of unit's unique argument
        DEREF          / replace the address with its value (a list)
        HEAD           / get the list head
        CALL      50   / call the FU 2 (whose first instruction has label 50)
        JUMPZ     30   / if the returned value is false, go to the instruction with label 30
        LOADARG   1    / load the address of unit's unique argument
        DEREF          / replace the address with its value (a list)
        HEAD           / get the list head (with type either int or double)
        LOADARG   1    / load the address of unit's unique argument
        DEREF          / replace the address with its value (a list)
        TAIL           / replace the list with its tail
        CALL      60   / recursive call of FU 1
        ADD            / sum the list head value to the value returned by FU 1
        RETURN    1    / terminate FU 1 (having 1 argument) by returning the sum
30      NEXT           / dummy instruction
        LOADARG   1    / load the address of unit's unique argument
        DEREF          / replace the address with its value (a list)
        TAIL           / replace the list with its tail
        CALL      60   / recursive call of FU 1
        RETURN    1    / terminate FU 1 by returning the value
                              / returned by the previous instance of FU 1
        / FUNCTION UNIT 2 (FI 2) with 1 local variable and 1 argument
50      START     1    / start the FU 2
        LOADLV    0    / load the address of the first local variable
        LOADARG   1    / load the address of the unit's unique argument
```

```
        DEREF           / replace the address with its value (a list element)
        TOTYPE          / get the type of the list element
        MODV            / assign this type to the first local variable
        LOADLV  0       / load the address of the first local variable
        DEREF           / replace the address with its value (a type or the metatype type)
        PUSHT   2       / push the type int into the stack
        EQ              / check whether the local variable is equal to type int
        JUMPNZ  55      / if so go to the instruction with label 55
        LOADLV  0       / load the address of the first local variable
        DEREF           / replace the address with its value (a type or the metatype type)
        PUSHT   1       / push the type double into the stack
        EQ              / check whether the local variable is equal to type double
        RETURN  1       / terminates FU 2 by returning the check result
                        / i.e., true if and only if the list element has type double
55      PUSHB   1       / push true into the stack as the list element has type int
        RETURN  1       / terminates FU 2 by returning true
        $END
/ End of Example of Assembler text file for Section A.5.3
```

CLVM_ASS generates two CLVM programs by replacing each assembler instruction with the corresponding CLVM one and by replacing each reference to a label with the labeled instruction address.

The CLVM code for the first program, whose instructions are suitably numbered, is:

```
0       INIT    2
1       LOADGV  0
2       SLIST
3       PUSHI   -2
4       HLIST
5       PUSHC   72
6       CLIST
7       PUSHT   2
8       CLIST
9       PUSHD   3.14
10      CLIST
11      ELIST
12      MODV
13      LOADGV  0
14      DEREF
15      PRINT   0
16      LOADGV  1
17      SSTRING
18      PUSHC   67
19      CSTRING
20      PUSHC   76
21      CSTRING
22      PUSHC   86
23      CSTRING
24      PUSHC   77
25      CSTRING
26      ESTRING
27      MODV
28      PUSHC   10
29      PRINT   0
30      LOADGV  1
31      DEREF
32      PRINT   1
33      PUSHC   10
34      PRINT   0
35      LOADGV  1
```

```
36      DEREF
37      PRINT   0
38      HALT
```

This CLVM program coincides with the assembler program as no labels are used. Instead, the CLVM code, reported next, solves a number of label references, indicated in bold bleu color:

```
0       INIT    2
1       LOADGV  0
2       DEREF
3       CALL    6
4       PRINT   0
5       HALT
6       START   0
7       LOADARG 1
8       DEREF
9       SLIST
10      ELIST
11      EQ
12      JUMPZ   15
13      PUSHI   0
14      RETURN  1
15      LOADARG 1
16      DEREF
17      HEAD
18      CALL    35
19      JUMPZ   29
20      LOADARG 1
21      DEREF
22      HEAD
23      LOADARG 1
24      DEREF
25      TAIL
26      CALL    6
27      ADD
28      RETURN  1
29      NEXT
30      LOADARG 1
31      DEREF
32      TAIL
33      CALL    6
34      RETURN  1
35      START   1
36      LOADLV  0
37      LOADARG 1
38      DEREF
39      TOTYPE
40      MODV
41      LOADLV  0
42      DEREF
43      PUSHT   2
44      EQ
45      JUMPNZ  51
46      LOADLV  0
47      DEREF
48      PUSHT   1
49      EQ
50      RETURN  1
51      PUSHB   1
52      RETURN  1
```

A5.2. *The Finite State Automaton to recognize CLVM Assembler Instructions*

We construct a finite state automaton that, given an assembler text file line, checks whether the line belongs to the three admissible types: *AsmBeginEnd, AsmInstr* or *AsmBlank.* The automaton, shown in Figure A.4, contains 12 states from 0 (the initial state) to 11. The final states are three (the ones with double circles):

- state 9: it recognizes that the line is of type *AsmBlank*, i.e. is a blank line that will be skipped;
- state 10: it recognizes that the line is of type *AsmInstr* and also returns the corresponding CLVM instruction;
- state 11: it recognizes that the line is of type *AsmBeginEnd* and also returns whether it is the command $BEGIN or $END.

In order to simplify the structure of the automaton, after reading the character '$' in the state 0 and passing to the state 4, instead of checking the two admissible string "BEGIN" or "END" character by character, a whole string is first collected and then the string check is made as a static semantic action, indicated by [ASM(Begin|End)]. The same approach is used at the state 3 to check whether the operator is one of the four admissible types: Op1 (no operand), Op2 (unsigned integer), Op3 (possibly signed integer) and Op3 (double). The check is indicated by [tOp(0)] if the operator is of type Op0 or [tOp(1|2|3)] otherwise.

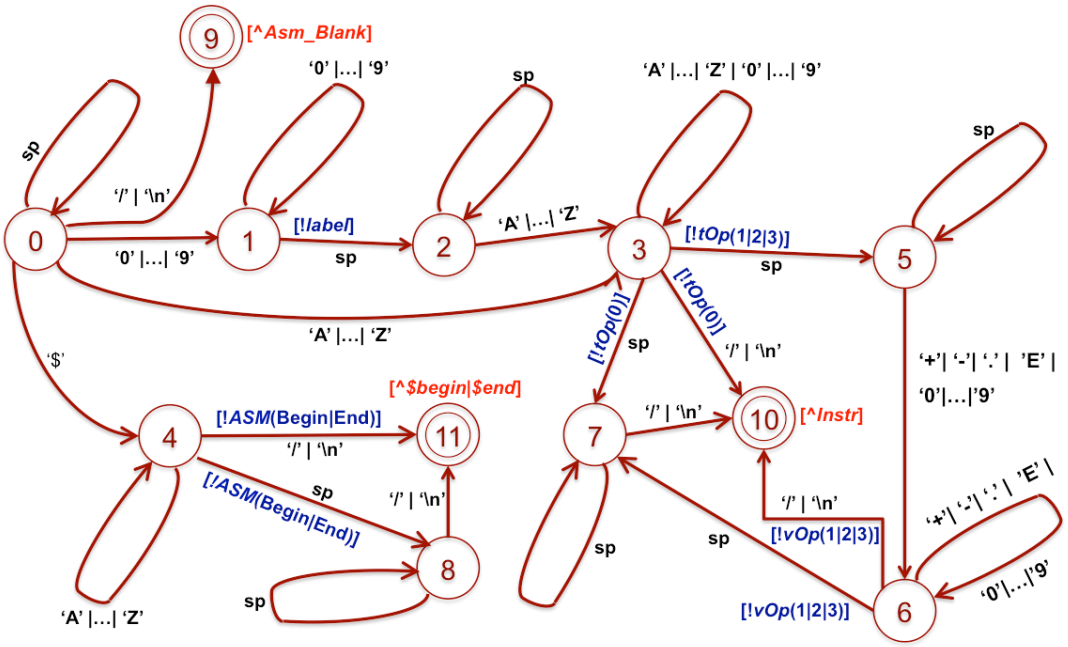


Fig. A.4: *Finite State Automaton*

The automaton's moves are the same for the three operator types Op1 or Op2 or Op3. The actual operator type is stored in order to perform the appropriate check for the operand before exiting from state 6, indicated by [vOp(1|2|3)] . Also for this check, the approach is to first accumulate the whole string and then making the check using a suitable static semantic action.

A5.3. *Using CLVM_ASS*

We construct a finite state automaton that, given an assembler text file line, checks whether the line belongs to the three admissible types: *AsmBeginEnd, AsmInstr* or *AsmBlank.* The

CLVM_ASS is invoked by typing the command "java –jar ClvmAss_R_4_0_1.jar" to the shell[8]. Then the following welcome message will be printed:

```
$java –jar ClvmAss_R_4_0_1.jar
     ***********************************************
                    ***  CLVM Assembler   ***
     ***********************************************
     ******** Release 4.0.1 of October 31, 2016 ******

Enter file name (path w.r.t the current working directory or full path):
>>
```

In launching CLVM_ASS, it is possible to include a number of arguments at the end of the java command to increase the sizes of some data structures, that are internally used and have been dimensioned for a typical usage. The arguments must be given as pairs, each consisting of a parameter (denoting an internal data structure) and a positive integer (providing a new value for the parameter overriding the default one). The available parameters are:

- *EXECCS*: it refers to the size of CODE (the memory used by CLVM to store an executable program) – the default value is 32,000
- *EXECMS*: it refers to the size of MEM (the memory used by CLVM) to store data – the default value is 64,000
- *EXECOS*: it refers to the size of the memory used by the virtual machine of Calculist to store the output – the default value is 16,000.

As an example, assume that we want to increase the output memory size to 20,000 and to extends the data memory to 80.000 elements. Then, we write:

java –jar ClvmAss_R_4_0_1.jar EXECOS 20000 EXECMS 80000

A new value for a parameter that is less than the default one does not have effect, i.e., the default value is preserved.

A CLVM Assembler session handles all assembler programs included in a text file. As an example, we enter the file described in the previous section – the default extension is "clvm".

```
>>Example_A_5_3_Tutorial
** Importing file "Example_A_5_3_Tutorial.clvm"  **
```

The first program of the file is analyzed and all the related lines are numbered and displayed.

```
1:    / Example of Assembler text file for Section A.5.3
2:
3:    / ** First Program
4:          $BEGIN      -- PROGRAM #1 --
5:          INIT  2     / Program with 2 global variables not yet initialized
6:          LOADGV 0    / load the address of the first global variable
…
40:         LOADGV 1    / load the address of the second global variable
41:         DEREF       / replace the address with the value (i.e. the string)
42:         PRINT 0     / the string is printed without quotes: CLVM
43:         HALT        / terminate the program
44:         $END  -- PROGRAM #1 --

Printing listing? (Y/N)
>>
```

[8] The jar file name refers to the release R.4.0.1. The suffix "R_4_0_1" in the name changes if a new release is used.

All the file lines up to the first "$END" instruction are loaded, listed, analyzed and possible errors are reported. The lines are also numbered to facilitate their retrieval inside a text editor, e.g., to make possible error correction.

We reply "n" to the request of printing the listing of the program – we shall select this option for the second program. Next we are asked whether the debugger is to be abilitated or not. We replay again "no" and, then, the program will be executed and the results are eventually printed.

```
>>n
Debug? (Y/N)
>> n
[ 2, 'H', int, 3.14 ]
"CLVM"
CLVM

Quit the session? (Y/N)
>>
```

We replay "y" so that the second program is loaded. Note that the line numbering is continued and the listing ends as soon as the "$END" instruction is reached.

```
>>n
45:
46:   / ** Second Program
47:   $BEGIN      -- PROGRAM #2 --
48:   INIT  2     / Program with 2 global variables initialized by the first
program
49:   LOADGV 0    / load the address of the first global variable (the list)
50:   DEREF       / replace the address with the list [-2, 'H', int, 3.14]
51:   CALL  60    / call the function unit 1 (FU 1) whose first instruction has
label 60
52:   PRINT 0     / print the result returned by the function unit: 1.14
53:   HALT        / terminate the program
54:   / FUNCTION UNIT 1 (FI 1) (with 0 local variables and 1 argument)
55:   60 START 0  / start the FU 1
56:   LOADARG 1   / load the address of unit's unique argument
…
103:  55 PUSHB 1  / push true into the stack as the list element has type int
104:  RETURN 1    / terminates FU 2 by returning true
105:  $END  -- PROGRAM #2 --

Printing listing? (Y/N)
>>
```

We now replay "yes" so that the second program instruction are listed and numbered starting from zero – the number of an instruction is its address in CODE. Labels are replaced by instruction addresses.

```
>>y
** Listing of the program unit **
0     INIT  2
1     LOADGV 0
2     DEREF
3     CALL  6
4     PRINT 0
5     HALT
6     START 0
7     LOADARG 1
…
51    PUSHB 1
52    RETURN 1
** End of Listing **
```

```
Debug? (Y/N)
>>
```

We again replay “n” so that the debbuger is not enabled. Then the program is executed and the result is displayed.

```
 >>n
5.140000000000001

Quit the session? (Y/N)
>>
```

We reply “y” so that the remaing file lines are loaded.

```
 >>n
106:  / End of Example of Assembler text file for Section A.5.3

Bye
```

As the are non more programs included in the file, the session is terminated.