Using CalcuList To MapReduce Jsons

Domenico Saccà
DIMES Dept, University of Calabria
87036 Rende, Italy
sacca@unical.it

Angelo Furfaro
DIMES Dept, University of Calabria
87036 Rende, Italy
a.furfaro@unical.it

ABSTRACT

CalcuList (*Calculator with List manipulation*), is an educational language for teaching functional programming extended with some imperative and side-effect features, which are enabled under explicit request by the programmer. In addition to strings and lists, the language natively supports json objects and may be effectively used to implement generic MapReduce recursive procedures to manipulate json lists. MapReduce is a popular model in distributed computing that underpins many NoSQL systems and a json list can be thought of as a dataset of a document NoSQL datastore. It turns out that CalcuList can be used as a tool for teaching advanced query algorithms for document datastores such as MongoDB and CouchDB.

CCS CONCEPTS

• **Software and its engineering** → **Functional languages**; • **Information systems** → *MapReduce languages*; Main memory engines;

KEYWORDS

MapReduce, Document Databases, Query Languages, Functional Language, Imperative Features

ACM Reference Format:

Domenico Saccà and Angelo Furfaro. 2018. Using CalcuList To MapReduce Jsons . In *IDEAS 2018: 22nd International Database Engineering & Applications Symposium, June 18–20, 2018, Villa San Giovanni, Italy*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3216122.3216164>

1 INTRODUCTION

Pure functional programming usually designates a specific functional programming paradigm that treats all computation as the evaluation of mathematical functions. Purely functional programming forbids changing-state and mutable data and mainly consists in ensuring that functions will only depend on their arguments, regardless of any global or local state (i.e., it has no *side effects*). An important member of the purely functional languages is Haskell [5, 6], which provides relevant features including polymorphic typing, static type checking, lazy evaluation and higher-order functions.

It also has an innovative type system which supports a systematic form of overloading and a module system.

Many so-called functional languages are less pure (*impure*), as they contain imperative features. For instance, most of the languages of the Lisp Family [16] were designed to be multi-paradigm and mainly present a purely functional interface for many operations (particularly simple mathematics and list/set operations) and an impure interface (more often object-oriented) for things where side effects are desirable, such as I/O. Haskell also expresses side effects but it does it by using an approach based on monads, which can be thought of as composable computation descriptions [9]. The essence of monad is thus separation of composition timeline from the composed computation’s execution timeline, as well as the ability of computation to implicitly carry extra data, as pertaining to the computation itself, in addition to its one (hence the name) output, that it will produce when run (or queried, or called upon). This lends monads to supplementing pure calculations with features like I/O, common environment or state, etc. Thus in Haskell, though it is a purely-functional language, side effects that will be performed by a computation can be dealt with and combined purely at the monad’s composition time.

CalcuList is a new language that combines functional and imperative programming but does not support imperative features alone, i.e., any application must have a functional core [13]. Thus, CalcuList essentially belongs to the family of “impure” functional languages even though it includes imperative features as well. In contrast with Haskell, CalcuList describes impure effects and actions inside the language itself, although it isolates impure components using an approach, different from Monads, which is derived by the semantic rules of an *Attribute Grammar* [8]. An attribute grammar extends a classical context-free grammar to specify the context-sensitive aspects of the syntax of a programming language (such as checking that an item has been declared and that its use is consistent with the declaration) as well as its operational semantics (e.g., by defining a translation into lower-level code of a specific machine architecture). In a similar way as a context-free grammar production is enriched with semantic rules to assign values to attributes associated to the symbols occurring in the production, CalcuList enriches the expression defining a function with statements assigning values to global variables and function parameters.

We stress that CalcuList (*Calculator with List manipulation*) is an *educational* 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 [15] and natively supports processing of strings, lists and json objects [4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

IDEAS 2018, June 18–20, 2018, Villa San Giovanni, Italy

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6527-7/18/06...\$15.00

<https://doi.org/10.1145/3216122.3216164>

The language is strongly typed, but the type checking is mainly dynamic as 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 an abstract computing machine, called CalcuList Virtual Machine (CLVM).

CLVM consists of three memory areas, each of them organized as an array of 64-bit cells: (1) 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 backward, (2) CODE, storing the instructions of the running CLVM program and (3) 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. CLVM is equipped with set of instructions that are implemented using micro-instructions operating with the three memories and the registers. The *clops* (CalcuList OperationS) of an instruction is the number of micro-instructions executed for its implementation.

In this paper we shall illustrate powerful features of CalcuList to support MapReduce, which is a programming model for processing and generating large data sets. It is a framework introduced by Google for support parallel computations large data sets spread over clusters of computers. Nowadays MapReduce is considered a popular model in distributed computing, inspired by the functions map and reduce commonly used in functional programming. It can be considered ‘Data-Oriented’ which process data in two primary steps: Map and Reduce. On top of that, the query is now executed on simultaneous data sources. The process of mapping the request of the input reader to the data set is called ‘Map’, and the process of aggregation of the intermediate results from the mapping function in a consolidated result is called ‘Reduce’.

MapReduce is mainly used to implement applications for NoSQL (“Not Only SQL”) systems that have been released and widely adopted in many domains poorly served by relational databases [10]. There are several different types of NoSQL systems including document stores that allow a structured document to be attached to a key – for CouchDB ([14]) the representation is json while for MongoDB ([1]) the representation format is BSON (Binary encoded Json). In document store systems there are implementations of MapReduce in spirit like Hadoop with all input coming from a collection and output going to a collection. For a practical definition, Map-Reduce in a document store is useful for batch manipulation of data and aggregation operations. In real case scenarios, in a situation where one would have used GROUP BY in SQL, MapReduce is the equivalent tool in MongoDB or CouchDB. In this paper we show how to use CalcuList to implement MapReduce programs on data structures based on json, including GROUP BY operators, and we argue that suitable interfaces could make CalcuList a powerful query environment for document datastores.

The remainder of this paper is organized as follows. Section 2 introduces basic notions about the CalcuList functional core and higher-order functions, describes the imperative aspects of the language and related side effects. Section 3 focus on the manipulation of json structures in CalcuList and on the usage of higher-order functions to implement MapReduce queries. Section 4 presents an interesting case-study: using MapReduce to query a dataset on Nobel Prizes published in [3]. Finally, Section 5 draws the conclusions and discusses future work.

2 AN OVERVIEW OF CALCULIST

2.1 Preliminaries

The main interface to CalcuList, like other languages (e.g. Python, Scala), is a REPL environment where the user can define functions and issue valid expressions (queries in CalcuList), which in turn are parsed, evaluated and printed before the control is given back to the user.

The basic types for CalcuList are six: (1) *double*, (2) *int*, (3) *char*, (4) *bool* (with values *true* or *false*), (5) *null* (that has a unique value named *null* as well) and (6) *type* (whose values are all the other types: *double*, *int*, etc.). The first three basic types are *numbers* and they are used in arithmetic operations. If the operands are of different type, they 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* terms returns an *int* value, as it happens in Java.

The language also supports three compound types: *string* (immutable sequences of characters), *list* (sequences of elements of any type), and *json* (JavaScript Object Notation), that is a lightweight data-interchange format. Details on the usage of strings and of lists are given next, whereas jsons will be treated in Section 3. As high order functions are supported, *function* is a type as well.

The arithmetic binary operators that may be used by number terms are $+$, $-$, $*$, $/$, $//$, $\%$. The first three operators work just like in most other languages (for example, Java or Python). On the other hand, the operator $/$ performs floating-point division also when both operands are integer, whereas $//$ performs integer division (i.e. quotient without remainder) like in Python. Finally, the modulus operator $\%$ computes the remainder like in Java.

The language includes comparison operators $>$, $>=$, $<$, $<=$, equality $==$ and inequality $!=$, which work just like in most other languages (for example, Java or Python). The two operands of a comparison operator must be both: (1) numeric, or (2) *bool* (with $false < true$) or (3) *string* (with lexicographic order). On the other hand, the two operands for equality and inequality may be of any (possibly heterogeneous) type. Finally, the logical operators follows Java syntax: $!$ (not), $\&\&$ (and), $||$ (or). Short-circuit evaluation is adopted for binary logical operators, i.e., given $A \&\& B$ (resp., $A || B$), B is evaluated only if A is true (resp., false).

A string in CalcuList is an immutable sequence (possibly empty) of characters. Two strings can be concatenated by the overloaded operator $+$ returning a new string. Slice operators à la Python are available to extract a substring. Given a string s , $s[i]$ returns the i -th character of s , $s[i:]$ returns the substring of s starting from the i -th character up to its end; $s[:i]$ returns the substring of s from the initial character up to $(i - 1)$ -th character of s ; $s[i:]$ returns the

substring of s that begins at the index i and ends up to the index $j - 1$.

The *list* is a powerful compound data type used by CalcuList for constructing dynamic data structures and implementing recursive algorithms. A list L consists of a number (possibly zero) of comma-separated elements between square brackets. As in Python, the elements can be of any type, including list and json, and heterogeneous. They are numbered with an index starting from zero: 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. List assignment is a shallow operator as the list elements are shared by the two lists involved in the assignment.

A list can be extended by adding additional comma-separated elements on top, followed by the append operator (the bar “|”) - the syntax has been inspired by Prolog [2]. For instance, given a list L , $M=[x,y|L]$ extends L by adding two new elements, x and y , on top of L . 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. Also the concatenation $L1+L2$ of two lists is shallow as the elements of $L2$ are afterwards shared by $L1$.

Another shallow operator on a list L is $L[>]$, which returns the (possibly empty) *tail* of L , i.e., the list starting from the 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.

Slice operators on a list L (say with n elements) are deep in the sense that they clone the elements. In particular, $L[:]$ clones the whole list L and $L[i:]$ clones the sublist of L from the element $L[i]$ to the end; in the same way, $L[i_1:i_2]$ and $L[:i]$ clone the corresponding sublists.

The syntax for an expression is like in Java. An expression e is either *simple* or *conditional* with format $e_1 ? e_2 : e_3$, where e_1 is a logical (i.e., with type *bool*) expression and e_2 and e_3 are expressions (possibly conditional in their turn). The meaning of a conditional expression is: its value is equal to the value of e_2 if e_1 evaluates to *true* or, otherwise, to the value of e_3 .

The language allows the user to define global variables. A *global variable* V is defined as “ $V = e$ ”, where e is an expression: the type of V is inferred from the type of e and its value is kept until a later re-definition, which may change also the type. Compound assignment operators $+=$, $-=$, $*=$, $/=$ are available as well.

At the start of a CalcuList session, the set of global variables is empty. The *state* of a session is given by the set of all defined global variables with the most recently assigned values and types for them.

A *query* is an expression e prefixed by the $^$ symbol and prints the value of the expression at the current state. Examples of variable definitions and queries are shown in the CalcuList session of Figure 1 - they are prompted by “ $>>$ ” and are terminated by a semi-colon. Observe that, given an expression e , the casting operator $e@type$ returns the type of e - in the example of Figure 1, $^z@type$ returns the type *double* of the variable z , which has value 4.2.

In addition to queries, the user may submit “service commands” (preceded by $!$) to inquiry about the variables and functions defined during the session, to list the history of all definitions, to print the state of the internal memory, to display the number of micro-instructions performed in the last execution, to enable or disable

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>> z='A'+1=='B';
>> ^z;
true
>> y="Hello "+"World"+"d";
>> ^y;
Hello World
>> ^y[0:1]+'i'+y[5:];
Hi World
>> ^z@type;
double
>> !variables;
-----GLOBAL VARIABLES-----
(GV0)[0] ans: type = double
(GV1)[2] x: double = 4.2
(GV2)[4] z: bool = true
(GV3)[6] y: string = 511973 (->Heap)
---End of GLOBAL VARIABLES---
```

Figure 1: Examples of Definitions and Queries

the debugger and the tail recursion optimizer, to save or import a session of definitions and so on. The service command `!variables` lists all global variables that have been defined during the session. The variable `ans` is built-in and stores the result of the last query returning the type of the variable `z`. Observe that the value of the variable `y` of type string is a pointer to a location in the heap storing the string characters.

2.2 Functions

A function is defined by giving it a name, followed by: its comma-separated parameter names (included in parentheses), the colon symbol “ $:$ ” and the (typically conditional) expression that computes the value of the function. Parameters as well as the return value of a function are defined without specifying types for them, so that static type checking is performed only for constant operands. A function is compiled while it is defined and, as for Python, the type checking is done at run time, after its call, when all the types become available. Lazy evaluation is not supported.

No side effects are allowed in the basic definitions of functions, i.e., functional operations do not modify current global variables and always create new data objects. Differently from Python, global variables are not accessible inside a function so that a possible change of state does not affect the function behavior.

Figure 2 includes a number of functions for handling lists. The first one is the classical member function that checks whether an element belongs to a list. The `sumL` function sums the elements of a list. The `rev` function constructs the reverse of a given list. This function uses “tail recursion”, which skips the entire chain of recursive functions returning and returns straight to the original caller. To show examples of queries, we define a list L with 1000 elements, by means of the function `range`, and issue a query to reverse L and a query to compute the sum of all elements in L .

CalcuList supports higher-order functions since a function parameter can also be a function and a function may return a function. A function parameter f is written as f/n , where n is the arity of the

```

>>member(x,L): L !=[] && (x==L[.] || member(x,L[>]));
>>sumL(L) : L==[]? 0: L[.] + sumL(L[>]);
>>rev1(L,R): L==[]? R: rev1(L[>], [L[.]|R]);
>>rev(L): rev1(L,[]);
>>range(x1,x2): x1>x2? []: [x1|range(x1+1,x2)];
>>L= range(1,1000); /* global variable */
>>^rev(L); /* query that reverses the list L */
[1000, 999, 998, 997, ..., 1]
>>^sumL(L); /* compute the sum of the elements in L */
500500

```

Figure 2: Functions on Lists

```

>>map(L,f/1,m/1) : L==[]?[]: f(L[.])?
  [m(L[.])|map(L[>],f,m)]: map(L[>],f,m);
>>d2or3(x) : x%2==0 || x%3==0;
>>^map(range(1,10),d2or3,lambda x: x*x*x);
[ 8, 27, 64, 216, 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(map(range(1,10),d2or3, lambda x: x*x*x),
  sum,0);
2556
>>^reduce(map(range(1,10),lambda x: x%2==0&& x%3==0,
  lambda x: x*x),prod,1);
36

```

Figure 3: Higher Order Functions on Lists

function f . In addition, adding $/n$ after a function head $g(. . .)$ prescribes that the function g must return a function with arity n . As an example, consider the higher-order function `twice` that takes a function, and applies the function to some value twice – note that lambda functions are written in Python like syntax:

```
twice(f/1)/1: lambda x: f(f(x));
```

For instance, the query `^twice(lambda x: x+3)(7)` returns the value 13.

Some examples of higher-order function, based on the MapReduce paradigm, are presented in Figure 3. The function `map` receives a list L and two functions as parameters, both with arity 1: f checks whether an element has a certain property and, in case the test succeeds, m maps the element into another value. The function `map` scans all elements of L , for each of them it performs the possible mapping and eventually returns the list of the results. The first map query filters all numbers divisible by 2 or by 3 (see function `d2or3`) in the range from 1 to 10 and replaces each of these values with their cube (see the lambda function) – let M denote the result of this query.

The subsequent function `reduce` receives a list L , a function parameter f (mapping two values into another value) and an initial value that is returned at final stage of the recursion, when L reduces to an empty list. Then `reduce` applies f to two actual parameters: the list head and the result of `reduce` for the list tail. Therefore, the list is eventually reduced to a single value by recursively applying f from right to left. The first `reduce` query computes the sum of all elements in M (the list returned by `map`), whereas the second one computes the product of the squares (see second lambda function

parameter) of all integers in the range from 1 to 10 that are divisible both by 2 and by 3.

2.3 Imperative Aspects of CalcuList

One of the peculiar characteristics of *pure* functional programming languages is being stateless, i.e. each computation can only work on data which are received as its input parameters and on data obtained as outputs from operations invoked during its execution. The use of global variables within functions is then not allowed. While this forces the user to develop, and hence reason, in functional style, there are some situations where the ability to have side-effects may simplify the developed code or even result in a better computational efficiency.

A function in CalcuList can be simply declared with side effects by adding $*$ next to its name (*star function*) so that the usage of a number of global variables is enabled – the property of a function to be with or without side effects cannot be changed after its first definition. Global variables to be used by a star function must be explicitly listed in its definition body. Then, they not only are made accessible to the function, but also they may have assigned new values during the function execution, as in an imperative programming language. Actually also parameter values can be modified during the execution of a star function. Obviously, a star function cannot be called by a non-star one.

There are situations (e.g., while importing definitions elaborated in previous sessions) in which a user may accidentally erase the value of a global 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 $L: V1, V2, V3$ declares three labeled variables $L.V1$, $L.V2$ and $L.V3$ – 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.

To be used inside a function definition, a global variable must be declared as a labeled variable and its label (say L), followed by $*$, must be listed between brackets as $\langle L* \rangle$ at the beginning of the function definition – so labels implements a namespacing mechanism. The labeled variables are then addressed inside the function without writing their label. It is possible to list more than one label for a function, which are to be included between brackets together with possible local variables.

A star function may have side effects, that is: (1) labeled global variables can be used as terms inside the expression defining the function, in addition to constants, 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). A GSC is an assignment of an expression value to a labeled global variable or to a function parameter and represents an imperative statement with side effects. A GSC is an assignment of an expression value to a local variable, surrounded by adorned curly braces $\{!, !\}$.

If expr is a simple expression, GSCs can be inserted both before and after the expression expr :

$$\text{GSC}_1^* \text{expr} \text{GSC}_2^*$$

```

>>MATH: zeroD;
>>div*(x,y): <MATH*>  {! zeroD=false !} y==0 ? 0
    {! zeroD=true !}: x/y;
>>^div(3,0);
0
>>^MATH.zeroD;
true
>>swap*(L,i,j): <t> true  {! t=L[i] !}
    {! L[i]=L[j] !} {! L[j]=t !};
>>K=[1,2,3,4];
>>^swap(K,1,3);
true
>>^K;
[1, 4, 3, 2]
>>REDc: ListCount;
>>add1LC*(LC,e): <REDc*> LC==[]?
    true  {! ListCount +=[[e,1]] !}:
    LC[.][0]==e? true  {! LC[0][1]+=1 !}:
    add1LC(LC[>],e);
>>reduceCount_1*(L): <REDc*> L==[]? ListCount:
    {! &add1LC(ListCount,L[.]) !} reduceCount_1(L[>]);
>>reduceCount*(L): <REDc*>
    {! ListCount=[] !} reduceCount_1(L);
>>^reduceCount("Hello_World"@list);
[ [ 'H', 1 ], [ 'e', 1 ], [ 'l', 3 ], [ 'o', 2 ],
  [ ' ', 1 ], [ 'W', 1 ], [ 'r', 1 ], [ 'd', 1 ] ]

```

Figure 4: Global Labeled Variables within Functions

where GSC_1 is executed before the function starts its execution and GSC_2 at the end of its execution. Note that, as indicated by the Kleene operator $*$, GSC_1 and GSC_2 may indeed consist of a sequence of commands, each of then surrounded by curly braces $\{ ! ! \}$.

In case $expr$ is a conditional expression, the global setting commands may be also inserted before and after each sub-expression, e.g.,

$$GSC_1^* \text{ cond } ? GSC_2^* \text{ expr}_1 GSC_3^* : GSC_4^* \text{ expr}_2 GSC_5^*$$

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 $expr_1$ or (2) GSC_4 and GSC_5 are executed before and after the execution of $expr_2$. If $expr_1$ (or $expr_2$) is in turn a conditional expression, additional local setting commands can be introduced using the same schema as above.

Note that the usage of GSCs has been inspired by the semantic rules of an Attribute Grammar [8]. A global variable can be thought of as either an inherited or a synthesized grammar attribute and a GSCs corresponds to a semantic rule setting the attribute value.

In Figure 4 the variable `zeroD` is defined with label `MATH` and is used in the `div` function to detect division by zero. An example of star function without global variables is the function `swap`, reported in the figure, that swaps two elements i and j of a list L . The function always returns `true` after swapping the two elements. The `swap` function 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 `reduceCount` function takes a list L of possibly duplicated elements and returns a list LC of

pairs $[e, n]$, where e is an element in L and n is the number of occurrences of e in L . The function sets the global variable `ListCount` to empty and calls `reduceCount_1` by passing the list L . The function `reduceCount_1` takes every element e in L and checks whether it is already in `ListCount` or not by calling the function `add1LC` with arguments LC (an alias for `ListCount`) and the element e . The last query in Figure 4 returns the list of characters and their numbers of occurrences in the string "Hello_World". Note that "Hello_World"@list transforms the string in a list of characters.

2.4 Implementation of CalcuList

The CalcuList programming environment has been implemented as a small-sized Java project in Eclipse 4.4.1 with 6 packages:

- **ioHandler**, which handles the interaction with the user by means of three classes: `IOH` (reading commands, definitions and queries issued by the user), `Output` (printing the results of queries and service commands) and `History` (storing the history of all successful definitions and queries issued by the user since the start of the session);
- **synt**, which performs the syntactic analysis by means of two classes: `Parser` (implementing a descendent parser that uses an LL(1) grammar of CalcuList and launches static and dynamic semantic actions) and `Lex` (performing the lexical analysis and the recognition of tokens to be passed to `Parser`);
- **sem**, which performs both static and dynamic semantic by means of eight classes: `SymbH` (managing all defined symbols in the context of their actual usage), `SymbolTableH` (handling the table of all symbols), `FuncInfo` (storing function properties), `LambdaFuncInfo` (managing lambda function properties), `Transl` (performing the translation to instructions of the virtual machine), `CodeUnit` (collecting the instructions produced by `Transl` into code units, that are one main unit and zero or one or more function units), `PrintCode` (printing the code units when the debugging is enabled by the user) and `Linker` (which takes one or more code units and combines them into a single executable program for the virtual machine);
- **exec**, which implements the *CalcuList Virtual Machine (CLVM)* by means of five classes: `Instruction` (defining the instruction set for CLVM), `Exec` (executing CLVM programs), `Debug` (implementing a machine-level debugger), `FrameDebugHandler` (constructing data structures to print the frames of the code units under execution in debugging mode) and `PrintOutput` (handling an output memory where the results of a query are stored for their print at the end of a query execution);
- **asm**, which provides an assembler environment to run CLVM programs using an assembler language – it consists of two classes: `Asm` (implementing the interaction with the assembler programmer – as there is no editor available, an assembler program is read from a text file) and `LexAsm` (performing the syntactic analysis of an assembler program – note that the CLVM assembler is a regular language).

The Jar File for using CalcuList may be downloaded from the link in [11]. The size of this file is rather small: 134 kb. The draft of a tutorial on CalcuList may be downloaded from the link in [12].

CLVM is an abstract computing machine that has an instruction set like a real computing machine 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 backward;
- a second memory, called CODE, storing the instructions of the running CLVM program;
- 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.

CLVM is equipped with a processor having a “powerful” virtual Arithmetic Logic Unit (ALU) and a number of 64-bit registers. Instructions are stored into 64-bit words and are implemented using micro-instructions operating with the three memories and the registers. The clops (CalcuList OperationS) of an instruction is the number of micro-instructions executed for its implementation.

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 implements a query or a variable assignment and may eventually call the execution of an FU instance, thus temporally suspending its execution. In turn, the called unit instance may call other FU instances, including other instances of the same FU. Each active CU instance has associated a frame in the stack storing local variables, parameters and intermediate results. The frame of the MU contains the global variables in the first positions. By the end of the execution, all frames are removed and the stack will only contain the global variables.

3 MANIPULATION OF JSONS

3.1 Definition and Representation of Jsons

Json (JavaScript Object Notation) [4] is language-independent data interchange format, which has been originally introduced as a subset of the JavaScript scripting language. The use of json is now widespread in many applications, e.g. web-services, and support for it has been added to the standard libraries of many programming languages. Recently, a great deal of interest is focused on *json data-base stores* that are NoSQL database systems using json documents instead of relation tuples [7].

CalcuList natively supports json objects (referred to simply as *json*), which are at runtime represented as (possibly empty) sequences of fields separated by comma and enclosed into 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.

In Figure 5 we define a variable `emps` as a list of three jsons, each representing an employee. For instance, `emps[1]` is the employee with name “e2”, aged 32 and working for the projects “p1” and “p2”. 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 such a field then the value `null` is returned as it happens for `^emps[0][“projects”]`

```
>>emps = [ { "name": "e1", "age": 30 },
  { "name": "e2", "age": 32, "proj": ["p1", "p2"], "bonus": 10 },
  { "name": "e3", "age": 28, "proj": ["p1", "p3"] } ];
>>^emps[2][“proj”];
[ "p1", "p3" ]
>>^emps[0][“proj”] %*;
null
```

Figure 5: *Json Definition and Manipulation in CalcuList*

(the string `%*` next to the query is a display option to force the writing of the null value). Given 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 and value v . The assignment of a json to a variable is a shallow operation. To clone a json J , it is sufficient to type $J[:]$. Two jsons can be compared only with the operators `==` and `!=` and the equality is true if and only if the two json values point to the same data structure in the memory. The length of a json J , i.e., the number of its fields, can be obtained by invoking the built-in function `_len(J)`.

The representation of the list of json defined by the variable `emps` is shown in Figure 6. The variable `emps` (see the dashed box) is stored in the memory stack as a pair of data, separated by a dotted line: the type (list) and the value (a pointer to the first element of the json list, stored in the heap). Each list element consists of a json value (type json and pointer to the first field of the json) and pointer to the next list element. Each of the three jsons is represented as a sequence of field elements, represented by dark gray boxes. Each field element consists of a value (type field and pointer to a key-value storage) and a pointer to the next field element. A key-value storage, represented by a light gray box, consists of a pair value-type and of a key of type string. In the figure, any string value is represented in a simplified way: it directly points to the quoted string (possibly repeated), instead of pointing to a non-duplicated element in the string pool.

Figure 6: *Representation of a list of jsons*


```

>>jsFilter(LJ, filtC/3,K,V):LJ==[]?[]:filtC(LJ[.],K,V)?
  [LJ[.]]jsFilter(LJ[>],filtC,K,V):
  jsFilter(LJ[>],filtC,K,V);
>>selVeqK(J,K,V): J[K]!=null && J[K]==V;
>>^jsFilter(emps, selVeqK, "age", 28);
[ { "name": "e3", "age": 28, "proj": ["p1", "p3"] } ]
>>selVInK(J,K,V) : J[K]!=null && member(V,J[K]);
>>^jsFilter(emps, selVInK, "proj", "p1") ;
[{"name": "e2", "age": 32, "proj": ["p1", "p2"], "bonus": 10},
 { "name": "e3", "age": 28, "proj": ["p1", "p3"] } ]
>>mapProj(E) : E==[]? []: E[.]["proj"] != null?
  E[.]["proj"][:]+mapProj(E[>]): mapProj(E[>]);
>>^reduceCount(mapProj(emps));
[ [ "p1", 2 ], [ "p2", 1 ], [ "p3", 1 ] ]

```

Figure 7: Higher Order Functions on Jsons

3.2 Higher-Order Functions and Function with Side Effects on Jsons

Figure 7 presents a high-order function to filter the jsons that satisfy some conditions in one of the fields: `jsFilter`, whose filtering condition parameter is expressed by a function `filterC`, that receives the current json and a field, say with key K and value V , to be used for the evaluation. Next, the figure shows two implementations for `filterC`: (1) `selKV`, which checks whether the json has a field equal to (K, V) , and (2) `selKinV`, which checks whether the value V is included in the list of elements that represents the value for the field K in the json. Two queries on the variable `emps` defined in Figure 5 are issued: the first one filter all employees with age 28 and the second one the employees that work for at least the project "p1".

Figure 7 also presents a classical map function that scans all employees to produce the list of all projects for which they work – note that, by adding the option `[:]` to `E[.]["proj"]`, the project lists are cloned to avoid their coalescence. The query calling the function `reduceCount`, defined in Figure 4, aggregates the list of projects by storing each project once as a pair: project name and number of its occurrences (that is the total number of employees working for it).

Figure 8 includes a star function that updates a list of jsons. As a list of jsons can be thought of as a collection in a documentary database, star functions have the crucial role of updating the database. As an example we have written the function `giveBonus` that assigns a bonus of a given amount to all employees in a list – the amount is incremented if an employee already has a bonus. Obviously the function has side effects as it updates the elements in the parameter `emps`. We present a query that assigns a bonus of 100 Euro to all employees in the global variable `emps` (defined in Figure 5) who work in project `p1`.

4 MAPREDUCING JSONS IN CALCULIST: A NOBEL PRIZE CASE STUDY

4.1 The Dataset on Nobel Prizes

In this section, we define a number of functions for handling jsons, using a json dataset on nobel prizes published in a GitHub web page, listing a number of so-called “awesome” json datasets [3].

```

>>aBonus*(E,v):E==[]?true:E[.]["bonus"]==null?
  {! E[0]["bonus"]=v !} giveBonus(E[>],v):
  {! E[0]["bonus"]+=v !} giveBonus(E[>],v);
>>aBonus(jsFilter(emps, selVInK, "proj", "p1"), 100);
true
>>^emps;
[ { "name": "e1", "age": 30 },
  { "name": "e2", "age": 32,
    "proj": ["p1", "p2"], "bonus": 110 },
  { "name": "e3", "age": 28,
    "proj": ["p1", "p3"], "bonus": 100 }
]

```

Figure 8: Functions with Side Effects on Jsons

We use the dataset in [3], named “Laureate”, that is a json with exactly one field, whose key is “laureates” and the value is a list of jsons, one for each laureate (i.e., nobel prize winner). The informal schema of each laureate json includes a number of properties for the laureate (code, name, gender, date of birth, date of possible death, born country, born city, possible died country and city) and the list of prizes won by the laureate, in general only one prize except for 5 laureates who were winners of two prizes and 1 laureate with three prizes.

Each prize is described by the year (from 1901 to 2017), the category (Physics, Chemistry, Medicine, Literature, Peace, Economy), the share (from 1 to 4), the motivation and the list of affiliations for the laureates, typically one, which were determinant in letting her/him achieve the prize. Regarding the share values, if the number of laureates for a prize 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).

The dataset is copied into a text file, named “CL_laureates.dat”, after removing a number of invalid or incomplete data by means of an editor.

4.2 MapReducing Nobel Prizes

We now present a new Calculist session. We start with loading the file “CL_laureates.dat” into the variable `nobel_laureates`, as shown in Figure 9. Next, we extract the list of all laureates, store it into the variable `laureates`, print the sixth laureate in the list and compute the number of laureates using the built-in function `_len`.

Later on, in order to perform a number of queries, we simplify the structure of laureates in two way: (1) reduce the number of fields by removing the ones that are not relevant for the queries and (2) flatten the structure so that data on laureates with multiple prizes are replicated for each prize. To give an idea of the restructuring, the original json about Marie Curie shown in Figure 9 is restructured into two jsons as follows:

```

{ "id": "6", "bornCountryCode": "PL", "gender": "female",
  "year": "1903", "category": "physics", "share": "4" }
{ "id": "6", "bornCountryCode": "PL", "gender": "female",
  "year": "1911", "category": "chemistry", "share": "1" }

```

```

>>nobel_laureates=<<("./CL_laureates.dat");
/* read laureate dataset from a text file */
- data are correct
>>laureates=nobel_laureates["laureates"];
>>^laureates[5] %*;
{"id":"6", "firstname":"Marie", "surname":"Curie",
 "born": "1867-11-07", "died": "1934-07-04",
 "bornCountry": "Poland", "bornCountryCode": "PL",
 "bornCity": "Warsaw",
 "diedCountry": "France", "diedCountryCode": "FR",
 "diedCity": "Sallanches", "gender": "female",
 "prizes": [
 { "year": "1903", "category": "physics", "share": 4,
  "motivation": "...researches on the radiation..",
  "affiliations": []
 },
 { "year": "1911", "category": "chemistry", "share": 1,
  "motivation": "...discovery of..radium and polonium",
  "affiliations": [ { "name": "Sorbonne University",
                     "city": "Paris", "country": "France" }
 ]
 }
 ]
 }
>>^_len(laureates); /* total number of laureates */
916

```

Figure 9: Starting a session with the dataset “Laureates”

```

>>toIntY(y): (y[0]-'0')*1000+(y[1]-'0')*100+
              (y[2]-'0')*10+(y[3]-'0');
>>add1L(id,bCC,gender,bYear, L1,LP): LP==[]?L1:
    add1L(id,bCC,gender,bYear,
    [ {"id":id, "bornCC": bCC,
      "gender": gender, "year": LP[0]["year"],
      "age": toIntY(LP[0]["year"])-bYear,
      "category": LP[0]["category"], "share":
        LP[0]["share"]} | L1], LP[1]);
>>flattenL1(L,L1) : L==[]? L1: flattenL1(L[>],
    add1L(L[0]["id"],L[0]["bornCountryCode"],
    L[0]["gender"], toIntY(L[0]["born"][0:4]),
    L1,L[0]["prizes"]));
>>flattenL(L) : flattenL1(L,[]);
>>elaureates = flattenL(laureates);
>>^_len(elaureates);
923

```

Figure 10: Restructuring the dataset “Laureates”

As shown in Figure 10, the restructuring is made by the function `flattenL`, which in turn calls the function `flattenL1` to process every laureate in the original list. The latter function calls the function `add1L` to process every prize achieved by a laureate. As shown by the two queries with the `_len` operator, the number of laureates increases from 916 to 923 because of multiple prizes. Note that the key `bornCountryCode` is simplified into `bornCC` and a new field is added that stores the age of a laureate when s/he received the prize. As years are represented by strings, the computation of the age is performed by the function `toIntY`.

```

>>filterCount(L,f/2,Q): L==[]?0: f(L[0],Q)?
    1+filterCount(L[>],f,Q): filterCount(L[>],f,Q);
>>EU_Codes = [ "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[0] ||
    lFilter1(L1,key,vL[>]));
/* true if the value of field "key" for L1 is in vL */
>>laureateFilter(L1,Q) : Q==[] ||
    lFilter1(L1,Q[0],Q[1])&&laureateFilter(L1,Q[>]);
/* true if laureate L1 satisfies all conditions in Q */
>>laureateGroup1(L,Q): L==[]?0:laureateFilter(L[0],Q)?
    1+laureateGroup1(L[>],Q): laureateGroup1(L[>],Q);
/* count laureates in L satisfying conditions in Q */
>>^filterCount(elaureates,laureateFilter,
    [ ["bornCC", EU_Codes] ]);
445
>>Sciences= ["physics","chemistry","medicine"];
>>^filterCount(elaureates,laureateFilter,
    [ ["bornCC", EU_Codes], ["category", Sciences] ]);
302

```

Figure 11: First Group of Queries on “Laureates”

The first group of queries is shown in Figure 11. We first define the function `filterCount` that filters the elements of a collection L using a generic filter function f with arity 2 and counts the number of filtered elements. We next provide the implementation `laureateFilter` for the generic filter function f , where the filter condition is a list Q of the format $[Q_1, \dots, Q_i, \dots, Q_m]$, where $Q_i = [keyF_i, [val_1F_i, \dots, val_{n_i}F_i]]$ ($1 \leq i \leq m$), representing the conjunction of inclusion conditions, i.e.: a laureate L_1 is filtered if for each Q_i in Q , $L_1[keyF_i]$ is equal to one of the values in $[val_1F_i, \dots, val_{n_i}F_i]$. define the list `EU_Codes` of European Country Codes and, then, issue the query for computing the number of nobel prize winners who were born in Europe by launching `filterCount` with actual arguments: the collection of laureate, the filter function `laureateFilter` and the filter condition `[["bornCC", EU_Codes]]`. Next, we define the list `Sciences` as the set of scientific disciplines (physics, chemistry and medicine) and issue the query that further restricts the answer to the winners of prizes in `Sciences` using the conjunction of two conditions.

In Figure 12, we add a further filter R expressed as a conjunction of range conditions: $[[keyF_1, val_1F_1, val_2F_1], \dots, [keyF_m, val_1F_m, val_2F_m]]$. A laureate L_1 is filtered if for each $[keyF_i, val_1F_i, val_2F_i]$ in R , $val_1F_i \leq L_1[keyF_i] \leq val_2F_i$. For instance, `[["year", "2000", "2017"]]` states that the year of a prize is included in the range from 2000 to 2017. The range condition is checked by the function `rangeFilter`. We next define the filter function `laureateFilterR` that performs the conjunction of two conditions, represented as the two elements of a list Q : the element $Q[0]$ is passed to the function `laureateFilter`, defined in Figure 11, while $Q[1]$ is passed to `rangeFilter`. The filtering is used to compute the number of nobel laureates, both born in Europe and with a science prize achieved in the present millenium. The subsequent count query further restricts the filtering by requiring the laureates to be “young” (i.e., with age ≤ 60) when they received the prize.


```

>>rangeFilter(L1,R) : R==[] ||
    R[.][1] <= L1[R.][0]]&&L1[R.][0]] <= R[.][2]&&
    rangeFilter(L1,R[>]);
>>laureateFilterR(L1,Q): laureateFilter(L1,Q[0]) &&
    rangeFilter(L1,Q[1])
>>filterCount(elaureates, laureateFilterR,
    [ [ ["bornCC",EU_Codes], ["category", Sciences] ],
    [ ["year", "2000", "2017"] ] ]);
46
>>^filterCount(elaureates, laureateFilterR,
    [ [ ["bornCC",EU_Codes], ["category", Sciences] ],
    [ ["year", "2000", "2017"], ["age",0,60] ] ]);
9

```

Figure 12: Second Group of Queries on “Laureates”

```

>>laureateMap(L,f/2,Q,m/1): L==[]? []: f(L[.],Q)?
    [m(L[.]) | laureateMap(L[>],f,Q,m) ] :
    laureateMap(L[>],f,Q,m);
>>mapG(L1): L1["gender"];
>>^reduceCount(laureateMap(elaureates, laureateFilterR,
    [ [ ["bornCC",EU_Codes], ["category", Sciences] ],
    [ ["year", "2000", "2017"] ] ], mapG));
[ ["male", 44], ["female", 2] ]
>>^reduceCount(laureateMap(elaureates, laureateFilterR,
    [ [ ["bornCC",EU_Codes], ["category", Sciences] ],
    [ ] ], mapG));
[ ["male", 293], ["female", 9] ]
>>^reduceCount(laureateMap(elaureates, laureateFilterR,
    [ [ ["bornCC",EU_Codes] ], [ ] ], mapG));
[ ["male", 425], ["female", 20] ]

```

Figure 13: Third Group of Queries on “Laureates”

In Figure 13, we present some applications of the MapReduce scheme, using the function `reduceCount`, defined in Figure 4. To this end, we first define the function `laureateMap` that filters the elements of a collection according to the filter function `f` and maps the filtered elements by means of the generic map function `m`. An implementation of `m` is `mapG` that extracts the gender of each filtered laureate. Then we issue three queries computing the following numbers, itemized by gender: (1) the number of nobel laureates, both born in Europe and with a science prize achieved in the present millenium, (2) the number of nobel laureates, both born in Europe and with a science prize and (3) the number of nobel laureates born in Europe.

In Figure ??, we define the function `reduceAvg` that computes the average of a list of values. This is done by combining summation and counting and by eventually computing their quotient. The implementation of the function is based on Global Setting Commands. The function `reduceAvg` is applied to the result computed by the function `laureateMap`. As implementation of `m`, we use `mapA` that extracts the age of each filtered laureate. Then we issue three queries computing the following average ages: (1) for the nobel laureates, both born in Europe and with a science prize achieved in the present millenium, (2) for the nobel laureates, both born in Europe and with a science prize and (3) for the nobel laureates born in Europe.

```

>>reduceAvg_1(L): <CS> L==[]? [0,0]:
    {! CS=reduceAvg_1(L[>]) !} [ CS[0]+1, CS[1]+L[.] ];
>>reduceAvg(L):<CS>{! CS=reduceAvg_1(L) !}CS[1]/CS[0];
>>mapA(L1): L1["age"];
>>^reduceAvg(laureateMap(elaureates, laureateFilterR,
    [ [ ["bornCC",EU_Codes], ["category", Sciences] ],
    [ ["year", "2000", "2017"] ] ], mapA));
69.54347826086956
>>^reduceAvg(laureateMap(elaureates, laureateFilterR,
    [ [ ["bornCC",EU_Codes], ["category", Sciences] ],
    [ ] ], mapG));
56.63245033112583
>>^reduceAvg(laureateMap(elaureates, laureateFilterR,
    [ [ ["bornCC",EU_Codes] ], [ ] ], mapG));
59.6561797752809

```

Figure 14: Fourth Group of Queries on “Laureates”

5 CONCLUSION

In this paper we have presented a new educational functional programming language extended with imperative programming features: CalcuList, whose imperative features are enabled under explicit request by the programmer. CalcuList expressions and functions are first compiled and then executed each time a query is issued. The object code produced by a compilation is a program that will be eventually executed by the CalcuList Virtual Machine (CLVM). The system provides an assembler component to run CLVM programs using an assembler language. The CalcuList programming environment has been implemented as a small-sized Java project in Eclipse 4.4.1 with 6 packages and 20 classes all together. The Jar File for using CalcuList may be downloaded from the link in [11]. The size of this file is rather small: 134 kb. The draft of a tutorial on CalcuList may be downloaded from the link in [12].

CalcuList has been used since 2011 by the first author as a didactic tool for his course of Formal Languages, thought at the first year of the Master in Computer Engineering at University of Calabria. The first version of CalcuList was rather rudimental and a number of features have been later on added year after year on demand. For instance, when it became popular the MapReduce programming paradigm, higher-order functions were introduced. Later on, as the manipulation of jsons became a new frontier, they were added to CalcuList as first class objects. In addition, to enable the update of a json document while filtering it, Global Setting Commands (GSCs) were introduced to perform operations with side effects using a style borrowed by the semantic rules of attribute grammars. In sum, in six years CalcuList has moved from a simple pure functional core to a flexible functional programming environment with interesting capabilities to query document databases. Further work is devoted to reinforce the interface between CalcuList and documentary datastores, such as MongoDB [1] and CouchDB [14], so that the language may be used as a power query system for teaching advanced query algorithms for such document datastores.

REFERENCES

- [1] K. Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.
- [2] W. F. Clocksin and C. S. Mellish. *Programming in Prolog (2Nd Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.
- [3] J. Dorfman. Awesome json datasets. <https://github.com/jdorfman/awesome-json-datasets#nobel-prize>, 2018.

- [4] ECMA Int. The JSON Data Interchange Format. Standard ECMA-404, Ecma International, 2013.
- [5] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.
- [6] S. Marlow. Haskell 2010 Language Report.
- [7] A. Nayak, A. Poriya, and D. Poojary. Type of NoSQL Databases and its Comparison with Relational Databases. *International Journal of Applied Information Systems*, 5(4):16–19, March 2013. Foundation of Computer Science, New York, USA.
- [8] J. Paakki. Attribute Grammar Paradigms – a High-level Methodology in Language Implementation. *ACM Comput. Surv.*, 27(2):196–255, June 1995.
- [9] S. L. Peyton Jones and P. Wadler. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM.
- [10] V. Reniers, D. Van Landuyt, A. Rafique, and W. Joosen. On the State of NoSQL Benchmarks. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 107–112, New York, NY, USA, 2017. ACM.
- [11] D. Saccà. The CalcuList Environment, Release 4.1.0. https://github.com/saccad/CalcuList/blob/master/CalcuList_R_4_1_0.jar, 2017.
- [12] D. Saccà. The CalcuList Release 4.1 Tutorial, Version 1. https://github.com/saccad/CalcuList/blob/master/CalcuList_Tutorial_Release_4_v3_5.pdf, 2017.
- [13] D. Saccà and A. Furfaro. CalcuList: a Functional Language Extended with Imperative Features. *CoRR*, abs/1802.06651, 2018.
- [14] C. Team. *CouchDB 2.0 Reference Manual*. Samurai Media Limited, United Kingdom, 2015.
- [15] G. van Rossum and J. Fred L. Drake. *The Python Language Reference Manual (version 3.2)*. Network Theory Limited, 2011.
- [16] P. Winston and B. Horn. *LISP. Second edition*. Addison Wesley Pub., Reading, MA, Jan 1986.