CalcuList: a Functional Language Extended with Imperative Features

Domenico Saccà Angelo Furfaro

DIMES, Università della Calabria, 87036 Rende, Italy
{sacca,a.furfaro}@unical.it

September 19, 2017

OUTLINE

The CalcuList

CalcuList (*Calcu*lator 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.
The language has a Python-like syntax and interactive computation sessions with the user are established through a REPL (Read-Evaluate-Print-Loop) shell.
The object code produced by a compilation is a program that will be eventually executed by the CalcuList Virtual Machine (CLVM).

Basic Computations

The basic types for CalcuList are five: (1) *double*, (2) *int*, (3) *char*, (4) *bool* and (5) *null*.
The first three basic types are *numbers* and they are automatically casted to their most general numeric type in arithmetic operations.
The arithmetic binary operators are `+`, `--`, `*`, `/`, `//`, `%`.
The language supports 3 compound types: *string*, *list*, and *json*.
The operator + is overloaded as it may be used for concatenating strings and lists
Comparison operators: `>`, `>=`, `<`, `<=`, `==`, `!=`

Basic Computations – *Strings*

A string in CalcuList is an immutable sequence (possibly empty) of characters.
Two strings can be concatenated by the overloaded operator +.

Slice operators:

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 starting from the initial character up to the $(i-1)$-th character of s; $s[i:j]$ returns the substring that begins at the index i and ends at the index $j-1$.

Basic Computations – *Global Variables*

An expression e is either *simple* or *conditional* with format
e_1 ? e_2 : e_3
where e_1 is a logical expression and e_2 and e_3 are expressions.
The value of a conditional expression is equal to the value of e_2 if e_1 evaluates to *true* or, otherwise, to the value of e_3.

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.

See *Session* 1

Basic Computations – *Pure Functions*

A function is defined by giving it a *name*, followed by: its comma-separated *parameters* (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 type checking is done at run time – lazy evaluation is not supported.
No side effects are allowed in the basic definitions of functions. So, differently from Python, global variables are not accessible inside a function

See *Session 2*

Manipulation of Lists

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.
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.
Two lists can be compared only with the operators == and `!=`.
The *tail* of `L` is returned by `L[>]` – shallow operation.
Given a list `L`, `M=[x,y | L]` extends `L` by adding two new elements, `x` and `y`, on top of `L` – shallow operator.
Slice operators: `L[:]` clones the whole list `L`, `L[i:]` the sublist of `L` from the element `L[i]`, $\texttt{L[i}_1 : \texttt{i}_2\texttt{]}$ and `L[:i]` clone the corresponding sublists.

See *Session 3*

Manipulation of Lists: Hidden Side Effects

Function `listRev` contains the shallow operation `listRev(L[>])+[L[.]]`. However there are no side effects on `L` as the list returned by `listRev(L[>])` is constructed from an empty list (at the last step of the recursion) that is later extended with elements copied from `L`.
Also the function `rev1` contains a shallow operation `[L[.]|R]`. In this case, `R` is modified by the computation but it has been passed by the function `rev` with the initial value of an empty list so that no side effects will occur.
Session 4 shows another function that makes use of shallow operations: `merge(O1,O2)`. 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.

See *Session 4*

Manipulation of Jsons

A json is 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.
Session 5 shows the definition of an employee as a json.
Given a json J and a key k, $J[k]$ denotes the value of the field of J with key equal to k.
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 !=.

See *Session* 5

Higher-Order Functions

CalcuList supports higher-order functions since a function parameter can also be a function; however, a function cannot return a function.
A function parameter `f` is written as `f`/*n*, where *n* is the arity of the function `f`.

As an example of higher-order functions, Session 6 defines classical function *map* and *reduce*.

Session 7 shows some higher-order functions for querying a list of jsons.

See *Sessions* 6 and 7

Local Variables

To simplify the writing of a function and, sometime, to optimize its execution (mainly, by avoiding to call twice the same function), it is possible to define *local variables* inside a function as follows:
`f(...):  <LV1,..., LVn> expr`.
The *n* variables are only used by the function `f` and are stored into each of the frames of each called `f` instance.
Local variable values are manipulated by *Local Setting Commands* (LSC), that are imperative statements without side effects.
An LSC is an assignment of an expression value to a local variable, surrounded by adorned curly braces {!, !}.
LSCs inspired by the semantic rules of an Attribute Grammar. A local variable as a inherited grammar attribute and an LSCs as a semantic rule setting the attribute value.

See *Session 8*

Global Variables inside Functions

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.
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.
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 `<L*>` at the start of the definition. The labeled variables are addressed inside the function without writing their label. One or more labels may be used by a function and are listed between brackets together with local variables.

See *Session 9*

Functions with Side Effects

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).
The syntax of GSCs is similar to the one of LSCs. The general format for GSC is: vspace-0.12cm

$$\mathrm{GSC}_1^*\ \texttt{cond}\ ?\ \mathrm{GSC}_2^*\ \texttt{expr}_1\ \mathrm{GSC}_3^*:\ \mathrm{GSC}_4^*\ \texttt{expr}_2\ \mathrm{GSC}_5^*$$

Also GSCs correspond to semantic rules of an Attribute Grammar. A global variable can be thought of as either an inherited or a synthesized grammar attribute and a GSCs corresponds to semantic rules setting the attribute values.

See *Session* 10

Summing Up

1. CalcuList is a new educational functional programming language extended with imperative programming features, which are enabled under explicit request by the programmer.
2. CalcuList expressions and functions are first compiled and then executed when a query is issued. Execution by the CalcuList Virtual Machine (CLVM). also available an assembler component to run assembler CLVM programs.
3. 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 (134 kb) and a draft of a tutorial on CalcuListmay be dowloaded from my personal site: `http://sacca.deis.unical.it`.

CalcuList's Story

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 course used to focus on basic notions of languages, grammars and compilers and on two styles of declarative paradigms: logic programming (mainly Prolog and Datalog) and a functional language.
As it was hard to introduce another language to illustrate the functional paradigm in a limited number of teaching hours, the first idea was to focus on the functional features supported by classical imperative languages.
The results were very disappointing: the declarative style was at most adopted as syntactic sugar that flied away at the first serious attempt to replace iteration with recursion.

CalcuList's Story/2

At that point, also considered that students were eager to learn new emerging languages such as Python, the first author invented a new functional language that at the appearance looked as Python, but it did not support any iterative construct. So, to pass the exam, a student had to eventually learn a functional programming language, although with a practical look.
The first version of CalcuList was rather rudimental and a number of features have been later on added year after year on demand.
In six years CalcuList has moved from a simple pure functional core to a flexible functional programming environment with some (limited and controlled) imperative features that sometimes may very much simplify a strict pure functional notation.

Future Extensions of CalcuList

1. remove the present limitation that a function cannot return a function – this extension should not require much work
2. 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
3. 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 classes;
4. 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 passing list parameters each other.

```
/* Session 1*/
x=2+.1;
x *= 2;
^x;
x='A'+1=='B';
^x;
y="Hello "+"Worl"+'d';
^y;
^y[0:1]+'i'+y[5:];
!mem;
```

```
/* Session 2 */
fibe1(x,f2,f1,k) : x==k? f1: fibe1(x,f1,f1+f2,k+1);
fibe(x) : x <= 1? x: fibe1(x,0,1,1);
z=fibe(10);
^z;
^z@type;
!debug on;
^fibe(4);
!debug off;


```

```
/* Session 3 */
member(x,L): L !=[] && (x==L[.] || member(x,L[>]));
listRev(L): L==[]? []: listRev(L[>])+[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);
^listRev(L);
!clops;
^rev(L);
!clops;

/* restart session */
L = [1,2,3];
S = "string";
!mem;


```

```
42 /* Session 4 */
43 L1 = [1, 3, 5, 7, 9];
44 L2 = [0, 2, 4, 6, 8, 10, 12];
45 lMerge(O1, O2) : O1==[]? O2[:]: O2==[]? O1[:]:
 … O1[.]<O2[.]? [O1[.]|lMerge(O1[>], O2)]:
 … [O2[.]|lMerge(O1, O2[>])];
46 ^ lMerge(L1, L2) ;
47 ^ L1;
48 ^ L2;
49 !mem;
50
51 /* restart session */
52 member(x,L): L !=[] && (x==L[.] || member(x,L[>]));
53 L = [1,2,3];
54
55
```

```
/* Session 5 */
emps = [
  { "name": "e1",  "age": 30 },
  { "name": "e2", "age": 32, "projects": [ "p1","p2" ] },
  { "name": "e3", "age": 28, "projects": [ "p1", "p3" ] }
];
!mem;

^emps[2]["projects"];
^emps[0]["projects"];
^emps[1]@list;
^ans@json;
```

```
/* Session 6 */
range(x1,x2): x1>x2? []: [x1|range(x1+1,x2)];
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);
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);
^reduce(map(range(1,10),lambda x:
…x%2==0&&x%3==0,lambda x:x*x),prod,1);

```

```
/* Session 7 */
jsFilter(LJ,filtC/3,K,V):
LJ==[]?[]:filtC(LJ[.],K,V)?
    [LJ[.]|jsFilter(LJ[>],filtC,K,V)]:
    jsFilter(LJ[>],filtC,K,V);
select1KV(J,K,V): _isKey(J,K) && J[K]==V;
^jsFilter(emps,select1KV,"age",28);
select1KinV(J,K,V) : _isKey(J,K) && J[K]@type==list
&& member(V,J[K]);
^jsFilter(emps,select1KinV,"projects","p1") ;

```

```
92 /* Session 8 */
93 rotate(L,k) : <n,k1> {! n=_len(L) !}  k==0? []:
94  {! k1=k%n !} k<0? L[-k1:] + L[:-k1]: L[n-k1:] +
… L[:n-k1];
95 ^rotate([5,1,4,20,15,13], 3);
96 ^rotate([5,1,4,20,15,13 ],-4);
97
98 part1(x,L,o/2,T0,T1) : L==[]?[T0,T1]: o(L[0],x)?
… part1(x,L[>],o,[L[0]|T0],T1):
99 part1(x,L[>],o,T0, [L[0]|T1]);
100 part(x,L,o/2) : part1(x,L,o,[],[]);
101 quicksort(L,o/2) : <T01> L==[]? []: L[>]==[]?
… [L[0]]: {! T01=part(L[0],L[>],o) !}
102 quicksort(T01[0],o)+[L[.]|quicksort(T01[1],o)];
103 ^quicksort([3,11,2,8,6,5], lambda x,y: x<=y);
104
105
106
```

```
/* Session 9 */
Lab: i;
fLab*(x) : <Lab*> x+i;
Lab.i=0;
^fLab(10);
Lab.i=1;
^fLab(10);
```

```
/* Session 10 */
MATH: zeroD;
div*(x,y): <MATH*>  {! zeroD=false !} y==0 ?  0 {! zeroD=true !}: x/y;
MATH1: numErr, somma;
listDiv1*(x,L): <MATH*, MATH1*,d>  L==[]? []:
{!d=div(x,L[0]) !} zeroD?
    {! numErr +=1 !}  ['*' | listDiv1(x,L[>])]:
    [d | listDiv1(x,L[>])] {! somma+=d!};
listDiv*(x,L):  <MATH1*> {! numErr=0 !} {! somma=0 !} listDiv1(x,L);
^listDiv(4,[ 2, 0, -4, 0, 20 ]);
^MATH1.numErr;
^MATH1.somma;

newVx_1(m,j) : j >=m? []: [0 | newVx_1(m,j+1)];
newVx(m) : m <= 0? []: newVx_1(m,0);
triangLS_sum(A,X,i,j,n1) : j>n1? 0:
A[i][j]*X[j]+triangLS_sum(A,X,i,j+1,n1);
triangLS_1*(A,B,X,i,n1) : i<0? X: A[i][i] == 0?
exc("matrix A is singular"):
{! X[i]=(B[i]-triangLS_sum(A,X,i,i+1,n1))/A[i][i] !}
triangLS_1(A,B,X,i-1,n1);
triangLS*(A,B) : <n> {! n=_len(A) !} n!=_len(A[0])
|| n!= _len(B)?
        exc("matrix A and vector B are not conformant"):
        triangLS_1(A,B,newVx(n),n-1,n-1);
A=[ [1, 2, -1], [0, 2, 4], [0, 0, -2]];
B=[1,0,3];
^triangLS(A,B);
```

```
140
141 giveBonus*(emps,value):emps==[]?true:emps[0]["bonus"
… ]==null?
142     {! emps[0]["bonus"]=value !}
… giveBonus(emps[>],value):
143     {! emps[0]["bonus"]+=value !}
… giveBonus(emps[>],value);
144 ^giveBonus(jsFilter(emps,select1KinV,"projects","p1"
… ),100);
145 ^emps;
146
```