CalcuList: A Functional Abacus

Domenico Saccà

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

Outline

The Functional Language 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.

The Functional Language 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.
- Functional programming usually designates a specific functional programming paradigm that treats all computation as the evaluation of mathematical functions.

The Functional Language 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.
- 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*).

The Functional Language 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.
- 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 purely functional language is Haskell, which provides relevant features including polymorphic typing, static type checking, lazy evaluation and higher-order functions.

Main Properties of CalcuList

- 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.

Main Properties of CalcuList

- 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 seven: (1) *double*, (2) *long*, (3) *int* , (4) *char*, (5) *bool* (with values *true* or *false*), (6) *null* (that has a unique value named `null` as well) and (7) *type*.

Main Properties of CalcuList

- 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 seven: (1) *double*, (2) *long*, (3) *int* , (4) *char*, (5) *bool* (with values *true* or *false*), (6) *null* (that has a unique value named `null` as well) and (7) *type*.
- The language also supports three compound types: *string* (immutable sequences of characters), *list* (sequences of elements of any type), and *json* (JavaScript Object Notation, a lightweight data-interchange format).

Main Properties of CalcuList

- 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 seven: (1) *double*, (2) *long*, (3) *int* , (4) *char*, (5) *bool* (with values *true* or *false*), (6) *null* (that has a unique value named `null` as well) and (7) *type*.
- The language also supports three compound types: *string* (immutable sequences of characters), *list* (sequences of elements of any type), and *json* (JavaScript Object Notation, a lightweight data-interchange format).
- As high order functions are supported, *function* is a type as well.

A Simple Session

A Simple Session

```
>> x=2+.1;
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>>z='A'+1=='B';
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>>z='A'+1=='B';
>> ^z;
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>>z='A'+1=='B';
>> ^z;
true
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>>z='A'+1=='B';
>> ^z;
true
>> y="Hello "+"Worl"+'d';
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>>z='A'+1=='B';
>> ^z;
true
>> y="Hello "+"Worl"+'d';
>>^y;
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>>z='A'+1=='B';
>> ^z;
true
>> y="Hello "+"Worl"+'d';
>>^y;
Hello World
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>>z='A'+1=='B';
>> ^z;
true
>> y="Hello "+"Worl"+'d';
>>^y;
Hello World
>> ^y[0:1]+'i '+y[5:];
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>>z='A'+1=='B';
>> ^z;
true
>> y="Hello "+"Worl"+'d';
>>^y;
Hello World
>> ^y[0:1]+'i '+y[5:];
Hi World
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>>z='A'+1=='B';
>> ^z;
true
>> y="Hello "+"Worl"+'d';
>>^y;
Hello World
>> ^y[0:1]+'i '+y[5:];
Hi World
>>^x@type;
```

A Simple Session

```
>> x=2+.1;
>> x *= 2;
>> ^x;
4.2
>>z='A'+1=='B';
>> ^z;
true
>> y="Hello "+"Worl"+'d';
>>^y;
Hello World
>> ^y[0:1]+'i '+y[5:];
Hi World
>>^x@type;
double
```

A Session with Dates

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
>> ^_pDate(y);
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
>> ^_pDate(y);
Sun Sep 23 21:22:25 CEST 2018
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
>> ^_pDate(y);
Sun Sep 23 21:22:25 CEST 2018
>> ^_pDate(0@long);
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
>> ^_pDate(y);
Sun Sep 23 21:22:25 CEST 2018
>> ^_pDate(0@long);
Thu Jan 01 01:00:00 CET 1970
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
>> ^_pDate(y);
Sun Sep 23 21:22:25 CEST 2018
>> ^_pDate(0@long);
Thu Jan 01 01:00:00 CET 1970
>> ^_pDate(9223372036854775807);
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
>> ^_pDate(y);
Sun Sep 23 21:22:25 CEST 2018
>> ^_pDate(0@long);
Thu Jan 01 01:00:00 CET 1970
>> ^_pDate(9223372036854775807);
Sun Aug 17 08:12:55 CET 292278994
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
>> ^_pDate(y);
Sun Sep 23 21:22:25 CEST 2018
>> ^_pDate(0@long);
Thu Jan 01 01:00:00 CET 1970
>> ^_pDate(9223372036854775807);
Sun Aug 17 08:12:55 CET 292278994
>> ^_pDate(-9223372036854775807);
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
>> ^_pDate(y);
Sun Sep 23 21:22:25 CEST 2018
>> ^_pDate(0@long);
Thu Jan 01 01:00:00 CET 1970
>> ^_pDate(9223372036854775807);
Sun Aug 17 08:12:55 CET 292278994
>> ^_pDate(-9223372036854775807);
Sun Dec 02 17:47:04 CET 292269055
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
>> ^_pDate(y);
Sun Sep 23 21:22:25 CEST 2018
>> ^_pDate(0@long);
Thu Jan 01 01:00:00 CET 1970
>> ^_pDate(9223372036854775807);
Sun Aug 17 08:12:55 CET 292278994
>> ^_pDate(-9223372036854775807);
Sun Dec 02 17:47:04 CET 292269055
>> ^_pDate(-62135773200000);
```

A Session with Dates

```
>> y=_gDate(); /* issued yesterday evening */
>> ^y;
1537730545975
>> ^y@type;
long;
>> ^_pDate(y);
Sun Sep 23 21:22:25 CEST 2018
>> ^_pDate(0@long);
Thu Jan 01 01:00:00 CET 1970
>> ^_pDate(9223372036854775807);
Sun Aug 17 08:12:55 CET 292278994
>> ^_pDate(-9223372036854775807);
Sun Dec 02 17:47:04 CET 292269055
>> ^_pDate(-62135773200000);
Sat Jan 01 00:00:00 CET 1
```

Functions

- A function is defined as $\mathtt{fname(par_1, \ldots, par_n)} :< \mathtt{expr} >$.

Functions

- A function is defined as $\texttt{fname}(\texttt{par}_1, \ldots, \texttt{par}_\texttt{n}) :< \texttt{expr} >$.
- No type is to be assigned to parameters and to the return value so type checking is done at run time when all the types become available. Lazy evaluation is not supported.

Functions

- A function is defined as $\texttt{fname}(\texttt{par}_1, \ldots, \texttt{par}_n) :< \texttt{expr} >$.
- No type is to be assigned to parameters and to the return value so type checking is done at run time 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.

A Session with Functions

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
38157;
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
38157;
>> f1(x,f2,f1,k):  x==k?f1:  f1(x,f1,f1+f2,k+1);
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
38157;
>> f1(x,f2,f1,k):  x==k?f1:  f1(x,f1,f1+f2,k+1);
>> fibe(x) :  x <= 1?  x:  f1(x,0,1,1);
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
38157;
>> f1(x,f2,f1,k):  x==k?f1:  f1(x,f1,f1+f2,k+1);
>> fibe(x) :  x <= 1?  x:  f1(x,0,1,1);
>> ^fibe(10); 55;
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
38157;
>> f1(x,f2,f1,k):  x==k?f1:  f1(x,f1,f1+f2,k+1);
>> fibe(x) :  x <= 1?  x:  f1(x,0,1,1);
>> ^fibe(10); 55;
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
38157;
>> f1(x,f2,f1,k):  x==k?f1:  f1(x,f1,f1+f2,k+1);
>> fibe(x) :  x <= 1?  x:  f1(x,0,1,1);
>> ^fibe(10); 55;
>> !clops; 3360;
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
38157;
>> f1(x,f2,f1,k):  x==k?f1:  f1(x,f1,f1+f2,k+1);
>> fibe(x) :  x <= 1?  x:  f1(x,0,1,1);
>> ^fibe(10); 55;
>> !clops; 3360;
>> ^fib(30); 832040;
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
38157;
>> f1(x,f2,f1,k):  x==k?f1:  f1(x,f1,f1+f2,k+1);
>> fibe(x) :  x <= 1?  x:  f1(x,0,1,1);
>> ^fibe(10); 55;
>> !clops; 3360;
>> ^fib(30); 832040;
>> !clops; 580241737;
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
38157;
>> f1(x,f2,f1,k):  x==k?f1:  f1(x,f1,f1+f2,k+1);
>> fibe(x) :  x <= 1?  x:  f1(x,0,1,1);
>> ^fibe(10); 55;
>> !clops; 3360;
>> ^fib(30); 832040;
>> !clops; 580241737;
>> ^fibe(30); 832040;
```

A Session with Functions

```
>> fib(x):  x <=1?  x:  fib(x-1)+fib(x-2);
>> ^fib(10);
55;
>> !clops;
38157;
>> f1(x,f2,f1,k):  x==k?f1:  f1(x,f1,f1+f2,k+1);
>> fibe(x) :  x <= 1?  x:  f1(x,0,1,1);
>> ^fibe(10); 55;
>> !clops; 3360;
>> ^fib(30); 832040;
>> !clops; 580241737;
>> ^fibe(30); 832040;
>> !clops; 9920;
```

A Session with Functions and Lists

A Session with Functions and Lists

```
>> member(x, L) : L!=[] && ( x == L[.] || member(x, L[>]) );
```

A Session with Functions and Lists

```
>> member(x,L) : L!=[] && ( x == L[.] || member(x,L[>]) );
>> sumL(L) : L==[]? 0: L[.]+sumL(L[>]);
```

A Session with Functions and Lists

```
>> member(x,L) : L!=[] && ( x == L[.] || member(x,L[>]) );
>> sumL(L)  :  L==[]?  0:  L[.]+sumL(L[>]);
>> range(x1,x2) : x1 > x2?[] : [ x1 | range(x1 + 1, x2) ];
```

A Session with Functions and Lists

```
>> member(x,L) : L!=[] && ( x == L[.] || member(x,L[>]) );
>> sumL(L)  :  L==[]?   0:   L[.]+sumL(L[>]);
>> range(x1,x2) : x1 > x2?[] : [x1 | range(x1 + 1,x2)];
>> L= range(1,1000);
```

A Session with Functions and Lists

```
>> member(x,L) : L!=[] && ( x == L[.] || member(x,L[>]) );
>> sumL(L)  :  L==[]?  0:  L[.]+sumL(L[>]);
>> range(x1,x2) : x1 > x2?[] : [x1 | range(x1 + 1,x2)];
>> L= range(1,1000);
>>^sumL(L);
```

A Session with Functions and Lists

```
>> member(x, L) : L!=[] && ( x == L[.] || member(x, L[>]) );
>> sumL(L) : L==[]? 0: L[.]+sumL(L[>]);
>> range(x1, x2) : x1 > x2?[] : [ x1 | range(x1 + 1, x2) ];
>> L= range(1,1000);
>>^sumL(L);
500500
```

Higher-Order Functions

- CalcuList supports higher-order functions since a function parameter can also be a function and a function may return a function

Higher-Order Functions

- 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 function `f`. In addition, adding `/`n after a function head `g(...)` prescribes that the function g must return a function with arity n.

Higher-Order Functions

- 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 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(f/1)/1 : lambdax : f(f(x));`

Higher-Order Functions

- 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 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(f/1)/1 : lambdax : f(f(x));`
- the query `^twice(lambda x:  x+3)(7);` returns the value 13

A Session with Higher-Order Functions

A Session with Higher-Order Functions

$$>> \ \mathtt{map(L, f/1, m/1) : L == [\,]?\,[\,] : f(L[.])?}$$
$$\mathtt{[\, m(L[.]) \mid map(L[>], f, m)\,] : map(L[>], f, m);}$$

A Session with Higher-Order Functions

```
>> 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;
```

A Session with Higher-Order Functions

```
>> 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);
```

A Session with Higher-Order Functions

```
>> 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 ]
```

A Session with Higher-Order Functions

```
>> 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));
```

A Session with Higher-Order Functions

```
>> 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));
>> ^reduce(map(range(1, 10), d2or3, lambda x : x * x * x),
                lambda x, y : x + y, 0);
```

A Session with Higher-Order Functions

```
>> 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));
>> ^reduce(map(range(1, 10), d2or3, lambda x : x * x * x),
                 lambda x, y : x + y, 0);
2556
```

A Session with Higher-Order Functions

```
>> 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));
>> ^reduce(map(range(1, 10), d2or3, lambda x : x * x * x),
                lambda x, y : x + y, 0);
2556
>> d2and3(x)  :   x%2==0  &&  x%3==0;
```

A Session with Higher-Order Functions

```
>> 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));
>> ^reduce(map(range(1, 10), d2or3, lambda x : x * x * x),
                lambda x, y : x + y, 0);
2556
>> d2and3(x)  :   x%2==0  &&  x%3==0;
>> ^reduce(map(range(1, 10), d2and3, lambda x : x * x),
                lambda x, y : x * y, 0);
```

A Session with Higher-Order Functions

```
>> 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));
>> ^reduce(map(range(1,10),d2or3,lambda x : x * x * x),
                 lambda x,y : x + y,0);
2556
>> d2and3(x)  :   x%2==0 && x%3==0;
>> ^reduce(map(range(1,10),d2and3,lambda x : x * x),
                 lambda x,y : x * y,0);
36
```

Definition of Json Objects

- Json objects (referred to simply as *json*) are 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.
- Two fields of a json object must not have the same key.

Definition of Json Objects

- Json objects (referred to simply as *json*) are 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.
- Two fields of a json object must not have the same key.

```
>> emps = [ { "name":  "e1", "age":  30 },
{"name":"e2","age":32,"proj":["p1","p2"],"b":10},
{"name":"e3", "age":28, "proj":["p1","p3"] } ];
```

Definition of Json Objects

- Json objects (referred to simply as *json*) are 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.
- Two fields of a json object must not have the same key.

```
>> emps = [ { "name": "e1", "age": 30 },
{"name":"e2","age":32,"proj":["p1","p2"],"b":10},
{"name":"e3", "age":28, "proj":["p1","p3"] } ];
>> ^emps[2]["proj"];
```

Definition of Json Objects

- Json objects (referred to simply as *json*) are 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.
- Two fields of a json object must not have the same key.

```
>> emps = [ { "name": "e1", "age": 30 },
{"name":"e2","age":32,"proj":["p1","p2"],"b":10},
{"name":"e3", "age":28, "proj":["p1","p3"] } ];
>> ^emps[2]["proj"];
[ "p1", "p3" ]
```

Definition of Json Objects

- Json objects (referred to simply as *json*) are 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.
- Two fields of a json object must not have the same key.

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

Definition of Json Objects

- Json objects (referred to simply as *json*) are 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.
- Two fields of a json object must not have the same key.

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

MapReducing a List of Jsons

MapReducing 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);
```

MapReducing 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;
```

MapReducing 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);
```

MapReducing 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"]  ]
```

MapReducing 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]);
```

MapReducing 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");
```

MapReducing 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");
[ {"n":"e2","a":32,"p":["p1","p2"],"b":10},
  {"name" : "e3","age" : 28,"proj" : ["p1","p3"]} ]
```

MapReducing a List of Jsons – continued

```
>> selVinK(J,K,V) : J[K]!=null && member(V,J[K]);
>> ^jsFilter(emps,selVinK,"proj","p1");
[ {"n":"e2","a":32,"p":["p1","p2"],"b":10},
  {"name" : "e3","age" : 28,"proj" : ["p1","p3"]} ]
```

MapReducing a List of Jsons – continued

```
>> selVinK(J,K,V) : J[K]!=null && member(V,J[K]);
>> ^jsFilter(emps,selVinK,"proj","p1");
[ {"n":"e2","a":32,"p":["p1","p2"],"b":10},
  {"name" : "e3", "age" : 28, "proj" : ["p1", "p3"]} ]
>> mapProj(E) : E==[]?  []:  E[.]["proj"] !=
null?  E[.]["proj"][:]+mapProj(E[>]):
mapProj(E[>]);
```

MapReducing a List of Jsons – continued

```
>> selVinK(J,K,V) : J[K]!=null && member(V,J[K]);
>> ^jsFilter(emps,selVinK,"proj","p1");
[ {"n":"e2","a":32,"p":["p1","p2"],"b":10},
  {"name" : "e3", "age" : 28, "proj" : ["p1", "p3"]} ]
>> mapProj(E) : E==[]?  []:  E[.]["proj"] !=
null?  E[.]["proj"][:]+mapProj(E[>]):
mapProj(E[>]);
>> ^reduceCount(mapProj(emps));
```

MapReducing a List of Jsons – continued

```
>> selVinK(J,K,V) : J[K]!=null && member(V,J[K]);
>> ^jsFilter(emps,selVinK,"proj","p1");
[ {"n":"e2","a":32,"p":["p1","p2"],"b":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 ] ]
```

Conclusion: Wake-Up, it's almost over!

- We have presented CalcuList, a new educational functional programming language extended with imperative programming features, which are enabled under explicit request by the programmer.

Conclusion: Wake-Up, it's almost over!

- We have presented CalcuList, a new educational functional programming language extended with imperative programming features, which 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). There is also an assembler component to run CLVM programs using an assembler language.

Conclusion: Wake-Up, it's almost over!

- We have presented CalcuList, a new educational functional programming language extended with imperative programming features, which 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). There is also 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 size of the Jar File is rather small: 134 kb.