Expression evaluation, Implicit Indexing, Contexts and Default Functors in the newLISP Scripting Language

Lutz Mueller, 2007-2015. Last edit December 6th 2013, rev r9

Implicit indexing and Default Functors in newLISP are an extension of normal LISP expression evaluation rules. Contexts provide lexically closed state-full namespaces in a dynamically scoped programming language.

S-expression evaluation and implicit indexing

In an earlier paper it was explained how s-expression evaluation in newLISP relates to ORO (One Reference Only) automatic memory management [1]. The following pseudo code of the expression evaluation function in newLISP shows how implicit indexing is an extension of Lisp s-expression evaluation rules:

```
function evaluateExpression(expr)
    {
    if typeOf(expr) is constant (BOOLEAN, NUMBER, STRING, CONTEXT)
        return(expr)

    if typeOf(expr) is SYMBOL
        return(symbolContents(expr))

    if typeOf(expr) is QUOTED
        return(unQuotedContents(expr))

    if typeOf(expr) is LIST
        {
        func = evaluateExpression(firstOf(expr))
        args = rest(expr)
        if typeOf(func) is BUILTIN_FUNCTION
            result = evaluateFunc(func, args)
        else if typeOf(func) = LAMBDA_FUNCTION
            result = evaluateLambda(func, args)
                /* extensions for default functor */
        if typeOf(func) is CONTEXT
            func = defaultFunctor(func)
                if typeOf(func) = LAMBDA_FUNCTION
                    result = evaluateLambda(defaultFunctor(func), args)
```

```
        /* extensions for implicit indexing */
        else if typeOf(func) = LIST
            result = implicitIndexList(func, args)
        else if typeOf(func) = STRING
            result = implicitIndexString(func, args)
        else if typeOf(func) = ARRAY
            result = implicitIndexArray(func, args)
        else if typeOf(func) = NUMBER
            result = implicitNrestSlice(func, args)
        }
    }

    return(result)
    }
```

The general working of the function reflects the general structure of the *eval* function as described by *John McCarthy* in 1960, [2].

The function first processes atomic expression types. Constants evaluate to themselves and are returned. Symbols evaluate to their contents.

If the expression is a list the first element gets applied to the rest elements. As in Scheme, newLISP evaluates the first element before applying it the result to its arguments.

Traditional Lisp or Scheme only permit a built-in function, operator or user defined lambda expression in the first, functor position of the s-expression list for evaluation. In newLISP the context symbol type, list, array and number type also act as functions when in the functor position.

Built-in functions are evaluated calling *evaluateFunc(func, args)*, functors which are lambda expressions call the *evaluateLambda(func, args)* function. Both functions in turn will call *evaluateExpression(expr)* again for evaluation of function arguments.

The working of a context symbol in the functor position of an s-expression will be explained further down in the chapter about *namespaces and default functors*.

The list type causes newLISP to evaluate the whole s-expression to the element indexed by the number(s) following the list and interpreted as index or indices (in newLISP elements in nested lists can be addressed using multiple indices):

```
(set 'lst '(a b (c d e) f g))
(lst 2)  → (c d e)
(lst 2 1)  → d
```

```
(set 'str "abcdefg")
(str 2)  → "c"
```

A number in the functor position will assume slicing functionality and slice the following list at the offset expressed by the number. When the number is followed by a second number, the second number specifies the size or length of the sliced part of the list:

```
(1 lst)  → (b (c d e) f g)
(1 2 lst)  → (b (c d e))

(1 2 str)  → "bc"
```

On first sight it seems logical to extend the same principle to the boolean data type. A ternary conditional expressions could be constructed without the necessity of the `if` operator, but in practical programming this leads to difficulties when reading code and causes too much ambiguity in error messages. Most of the time implicit indexing leads to better readable code, because the data object is grouped together with it's indices. Implicit indexing performs faster, but is also optional. The keywords `nth`, `first`, `last` and `rest` and `slice` can be used in the few cases where readability is better when using the explicit form of indexing.

The environment stack and dynamic scoping

In the original Lisp *eval* function a *variable environment* is implemented as an association list of symbols and their values. In newLISP a symbol is implemented as a data structure with one value slot and the environment is not an association list but a binary tree structure of symbols and an *environment stack* storing previous values of symbols in higher evaluation levels.

When entering a lambda function, the parameter symbols and their current values are pushed on to the environment stack. When leaving the function, the symbols are restored to their previous values when popping the environment stack. Any other function called will see symbol values as currently defined in the scope of the calling function. The variable environment changes dynamically while calling and returning from functions. The scope of a variable extends dynamically into lower call levels.

The following example sets two variables and defines two lambda functions. After the function definitions the functions are used in a nested fashion. The changing parts of the variable environment are shown in bold type face:

```
; x  → nil, y  → nil;
; foo  → nil
; double  → nil
; environment stack: [ ]

(define (foo x y)
        (+ (double (+ x 1)) y))

; x → nil, y  → nil,
; foo  → (lambda (x y) (+ (double (+ x 1)) y))
; double  → nil
; environment stack: [ ]

(define (double x)
        (* 2 x))

; x  → nil, y  → nil
; foo  → (lambda (x y) (+ (double (+ x 1)) y))
; double  → (lambda (x) (* 2 x)))
; environment stack: [ ]

(set 'x 10) (set 'y 20)

; x  → 10, y  → 20
; foo  → (lambda (x y) (+ (double (+ x 1)) y))
; double  → (lambda (x) (* 2 x)))
; environment stack: [ ]
```

Similar to Scheme newLISP uses the same namespace for variable symbols and symbols holding user-defined lambda functions. The `define` function is just a short-cut for writing:

```
(set 'foo (lambda (x y) (+ (double (+ x 1)) y)))
```

During all these operations the environment stack stays empty `[ ]`. Symbol variables holding lambda expressions are part of the same namespace and treated the same way as variables holding data. Now the the first function `foo` gets called:

```
(foo 2 4)

; after entering the function foo
; x → 2, y → 4
; foo → (lambda (x y) (+ (double (+ x 1)) y))
; double → (lambda (x) (* 2 x)))
; environment stack: [x -> 10, y -> 20]
```

After entering the functions, the old values of `x` and `y` are pushed on the environment stack. This push-operation is initiated by the function *evaluateLambda(func, args),* discussed later in this paper. Inside `foo` the second function `double` is called:

```
(double 3)
```

```
; after entering the function double
; x -> 3, y -> 4
; foo → (lambda (x y) (+ (double (+ x 1)) y))
; double → (lambda (x) (* 2 x)))
; environment stack: [x -> 10, y -> 20, x -> 2]

; after return from double
; x → 2, y → 4
; foo → (lambda (x y) (+ (double (+ x 1)) y))
; double → (lambda (x) (* 2 x)))
; environment stack: [x -> 10, y -> 20]

; after return from foo
; x → 10, y → 20
; foo → (lambda (x y) (+ (double (+ x 1)) y))
; double → (lambda (x) (* 2 x)))
; environment stack: [ ]
```

Note that in newLISP dynamic scoping of parameter symbols in lambda expressions does not create lexical state-full closures around those symbols as in the Scheme dialect of Lisp. On return from the lambda function the symbol contents gets destroyed and memory is reclaimed. The parameter symbols regain their old values on exit from the lambda function by popping them from the environment stack.

In newLISP lexical state-full closures are not realized using lambda closures but using lexical namespaces. Lambda functions in newLISP do not create closures but can create a new scope and new temporary content for existing symbols during lambda function execution.

Lambda function evaluation

All of the processing just described happens in `evaluateLambda(func, args)`. The following pseudo code shows more detail:

```
function evaluateLambda(lambda-func, args)
    {
    for each parameter symbol in lambda-func
        pushEnvironmentStack(symbol, value)

    for each arg in args and the symbol belonging to arg
        ; evaluation of arg happens in old symbol environment
        assignSymbolValue(symbol, evaluateExpression(arg))

    for each body expression expr in lambda-func
        result = evaluateExpression(expr)

    for each parameter symbol in lambda-func
        popEnvironmentStack()

    return(result)
    }
```

The `evaluateExpression(args)` function and `evaluateLambda(func, args)` call each other in a recursive cycle.

Note that arguments to lambda functions are evaluated in the variable environment as defined previous to the lambda function call. Assignments to parameters symbols do happen after all argument evaluations. Only the arguments are evaluated which have a corresponding parameter symbol. If there are more parameter symbols than arguments passed, then parameter symbols are assigned `nil` or a default value.

Namespaces and the translation and evaluation cycle

All memory data objects in newLISP are bound directly or indirectly to a symbol. Either memory objects are directly referenced by a symbol or they are part of an enclosing s-expression memory object referenced by a symbol. Unbound objects only exist as transient objects as returned values from evaluations and are referenced on the result stack for later deletion [1].

Except for symbols, all data and program objects are referenced only once. Symbols are created and organized in a binary tree structure. Namespaces, called *contexts* in newLISP, are sub-branches in this binary tree. A context is bound to a symbol in the root context `MAIN`, which itself is a symbol in the root context.

With few exceptions all symbols are created during the code loading and translation phase of the newLISP interpreter. Only the functions `load`, `sym`, `eval-string` and a special syntax of `context` create symbols during runtime execution.

The two symbols `MAIN:x` and `CTX:x` are two different symbols at all times. A symbol under no circumstances can change it's context after it was created. A context, e.g `CTX`, itself is a symbol in `MAIN` containing the root pointer to a sub-branch in the symbol tree.

The working of context switching is explained using the following two code pieces:

```
(set 'var 123)
(define (foo x y)
    (context 'CTX)
    (println var " " x " " y))
```

The `(context 'CTX)` statement only has been included here to show, it has no effect in this position. A switch to a different namespace context will only have influence on subsequent symbol creation using `sym` or

`eval-string`. By the time `(context 'CTX)` is executed `foo` has already been defined and all symbols used in it have been looked up and translated. Only when using `(context ...)` on the top level it will influence the symbol creation of code following it:

```
(context 'CTX)
(set 'var 123)
(define (foo x y)
    (println var " " x " " y))
(context MAIN)
```

Now the context is created and switched to on the top-level. When newLISP translates the subsequent set-statement and function definition, all symbols will be part of `CTX` as `CTX:var`, `CTX:foo`, `CTX:x` and `CTX:y`.

When loading code newLISP reads a top-level expression translating then evaluating it. This cycle is repeated until all top-level expression are read and evaluated.

```
(set 'var 123)
(define (foo x y)
    (println var " " x " " y))
```

In the previous code snippet two top-level expressions are translated and evaluated resulting in the creation of the three symbols: `MAIN:var`, `MAIN:foo`, `MAIN:x`, `MAIN:y` and `MAIN:CTX`.

The symbol `MAIN:var` will contain the number `123` and the `MAIN:foo` symbols will contain the lambda expression `(lambda (x y) (println var " " x " " y))`. The symbols `MAIN:x` and `MAIN:y` both will contain `nil`. The `var` inside the definition of `foo` is the same as the `var` previously set to `123` and will be printed as `123` during execution of `foo`.

In detail the following steps are happening:

1. current context is `MAIN`
2. read the opening top-level parenthesis `I` and create a Lisp cell of type `EXPRESSION`.
3. read and lookup `set` in `MAIN` and find it to be a built-in primitive in `MAIN`, translate it to the address of this primitive function in memory. Create a Lisp cell of type `PRIMITIVE` containing the functions address in its contents slot.
4. read the quote `'` and create a Lisp cell of type `QUOTE`.
5. read and lookup lookup `var` in `MAIN`, it is not found in `MAIN`, create it in `MAIN` and translate it to the address in the binary symbol tree. Create a Lisp cell of type `SYMBOL` containing the symbols address in its contents slot. The previously created quote cell serves as an envelop for the symbols cell.

6. read `123` and create a Lisp cell of type `INTEGER` with `123` in its contents slot.
7. read the closing top-level parenthesis finish the following list structure in memory:

```
[ ]                          ; cell of type: EXPRESSION
  \
 [MAIN:set]  → [']  → [123]  ; three cells of type: PRIMITIVE, QUOTE, INTEGER
                 \
               [MAIN:var]    ; cell of type: SYMBOL
```

   The above list diagram shows the five Lisp cells, which are created. List and quote cells are envelope cells containing a list or a quoted expression.

   The statement `(set 'var 123)` is not executed yet, but symbol translation and creation have finished and the statement exists as a list structure in memory. The whole list structure can be referenced with one memory address, the address of the first created cell of type `EXPRESSION`.

8. Evaluate the statement

In similar fashion newLISP will read and translate the next top-level expression, which is the function definition of `foo`. Evaluating this top-level expression will result in an assignment of a lambda expression to the `foo` symbol.

In the above code snippet both instances of `var` refer to `MAIN:var`. The `(context 'CTX)` statement only changes the context, namespaces for newly created symbols. The symbol `var` was created during loading translating the `foo` function. By the time `foo` is called and executed `var` already exists inside the `foo` function as `MAIN:var`. The `(context 'CTX)` statement doesn't have any effect of the subsequent execution of `(println var)`.

Context statements like `(context 'CTX)` above, change the current context for symbol creation during the loading and translation phase. The current context defines under which branch in the symbol tree new symbols are created. This affects only the the functions `sym`, `eval-string` and a special syntax of `context` to create symbols. Once a symbol belongs to a context it stays there.

Namespace context switching

Previous chapters showed how to use context switching on the top-level of a newLISP source file to influence symbol creation and translation during the source loading process. Once different namespaces exist,

calling a function which belongs to different context, will cause a context switch to the namespace the called lambda function resides in. If the called function doesn't execute any `sym` or `eval-string` statements, then these context switches don't have any effect. Even the `load` command will always start file loading relative to context `MAIN` unless a different context is specified as a special parameter in `load`. Inside the file loaded context switches will have an effect of symbol creation during the load process as explained previously.

What causes the context switch is the symbol holding the lambda function. In the following code examples bold face is used for output generated by `println` statements:

```
(context 'Foo)
(set 'var 123)
(define (func)
    (println "current context: " (context))
    (println "var: " var))
(context 'MAIN)

(Foo:func)
current context: Foo
var: 123

(set 'aFunc Foo:func)
(set 'var 999)

(aFunc)
current context: MAIN
var: 123
```

Note that the call to `aFunc` causes the current context to be shown as `MAIN`, because the symbol `aFunc` belongs to context `MAIN`. In both cases `var` is printed as `123`. The symbol `var` was put into the `Foo` namespace during translation of `func` and will always stay there, even if a copy of the lambda function is made and assigned to a symbol in a different context.

This context switching behaviour follows the same rules when applying or mapping functions:

```
(apply 'Foo:func)
current context: Foo
var: 123

(apply Foo:func)
current context: MAIN
var: 123
```

The first time `Foo:func` is applied as a symbol – quoted, the second time the lambda function contained in `Foo:func` is applied directly, because `apply` evaluates it's first argument first.

Namespaces and the default functor

In newLISP a symbol is a *default functor* if it has the same name as the context it belongs too, e.g. `Foo:Foo` is the default functor symbol in the context `Foo`. In newLISP when using a context symbol in the functor position, it is taken as the default functor:

```
(define (double:double x) (* 2 x))
(double 3)  → 6

(set 'my-list:my-list '(a b c d e f))
(my-list 3)  → d
```

The second example combines implicit indexing with usage of a default functor.

Default functors can be applied and mapped using `apply` and `map` like any other function or functor symbol:

```
(map my-list '(3 2 1 2))  → (d c b c)

(apply double '(10))  → 20
```

Default functors are a convenient way in newLISP to pass lists or other big data objects by reference:

```
(set 'my-list:my-list '(a b c d e f))

(define (set-last ctx val)
        (setf (ctx -1) val))

(set-last my-list 99)  → f

my-list:my-list  → (a b c d e 99)
```

Default functors are also a convenient way to define functions with a closed state-full name space:

```
(context 'accumulator)
(define (accumulator:accumulator x)
        (if (not value)
                (set 'value x)
                (inc 'value x)))
(context MAIN)

(accumulator 10)  → 10
(accumulator 2)  → 12
(accumulator 3)  → 15
```

Note that the symbols `x` and `value` both belong to the namespace `accumulator`. Because `(context 'accumulator)` is at the top level, the

translation of following function definition for `accumulator:accumulator` happens inside the current namespace `accumulator`.

Namespaces in newLISP can be passed by reference and can be used to create state-full lexical closures.

The default functor used as a pseudo hash function

A default functor containing `nil` and in operator position will work similar to a hash function for building dictionaries with associative key → value access:

```
(define aHash:aHash) ; create namespace and default functor containing nil

(aHash "var" 123) ; create and set a key "var" to 123

(aHash "var")  → 123 ; retrieve value from key
```

References

[1] *Lutz Mueller*, 2004-2015
Automatic Memory Management in newLISP.

[2] *John McCarthy*, 1960
Recursive Functions of Symbolic Expressions and their Computation by Machine.