Developer's Guide to ConteX

Rex Ford

April 18, 2005

Contents

Chapter 1

Introduction

1.1 What is ConteX?

ConteX is a prefix-syntax programming language, and it is commonly called ’CX’. ConteX was designed and created and recreated (a few times) by Rex Ford. The current implementation is made in Java, but the plans are to convert it to C for a native runtime. CX sports prototype-based object oriented programming with an original syntax. (table of aspects)

1.2 Organization of this Document

This document is divided up into 4 sections, and if you’ve read everything up to this point, you’re practically done with the first section.

The second section teaches CX. It will provide all of the information to run the interpreter, and it will lead you through how to program in CX. It has subsections for each aspect of CX. For example, there is a section all about lists. Once you’ve gone through that, you really know CX. You can pretty much just refer to CX API’s from there and build whatever you want, but reading the third section will ensure that you know how to code properly styled, properly-commented code.

The third section of this document is a case study that will teach some of the commonly used foundation library functions as will as good programming style in CX. Each part of the case study simply explains what I had to do to make the code, dissecting how the program was made from scratch. Reading this will definitely help you become faster and more efficient in CX. In this section, I unleash the facts to fast-reference of API’s in CX, keeping your code efficient and maintainable, and providing proper standard documentation from within your code, leading to a project bound for success.

The fourth section of this book describes different CX libraries, contributed or appended to the foundation. This way, you have a brief idea of what’s available (so that your don’t reinvent the CX wheel) and you know where it is available. Some of the libraries are unfinished or planned and open for contribution.

Chapter 2

Programming in ConteX

2.1 What you'll need

It's best to try out the code as you learn CX, so you'll need the ConteX interpreter up and running on your computer as you read this to get the most out of this. If you haven't already, you can get the latest release of CX from `http://contex.sf.net` . The download comes as a zip file, containing a README.txt file explaining the rest of the directory structure and how to get the repl started. Once you have the repl started, you're ready to learn CX by going through the rest of this document and following along by testing variations of the code samples given here.

2.2 Getting Started

REPL stands for 'Read, Evaluate, Print Loop' and, in my head, I think of it as being pronounced: "repple". You might have used a repl before and you just don't know it. You type a set of code in at the repl and press enter, and that's when this loop starts: the code you've typed is read, evaluated, and the results (if any) will be printed, and you are prompted again. Even calculators use this, that a calculator should be familiar to you. Therefore it would help you understand the CX repl by doing some simple math problems. Since you should have the repl running by now, you should see a prompt that says "(cx:)". I'll use that prompt when showing code examples so that you know you can type that code right into your own repl and hopefully get the same results!

```
(cx:) + [2 3]
5
```

That's how to add 2 and 3. I'm sure you've already noticed the odd ordering of the math problem. CX aims to be consistent and so + should be treated no different from any other function: the function is stated first, then the arguments that are applied to it. You can also try out -,*, and /, which each also take a list of numbers. In case you are wondering what order you want the numbers in, just imagine sliding the operator between the two numbers in the list. +, *, and - can take variable amounts of numbers, and they will give the sum, product, or trailing difference of all of the numbers, respectively. You don't need to add a space before the function and the arguments. CX will separate lists from function calls without a space between them. You could relate this syntax to *f(x)* in mathematics.

All functions that take input in CX can only take one input. It is imperative that you understand that + is receiving one thing: a list of 2 numbers. The list started at the '[' and ended at the ']'. There are unary functions that don't take lists. An example of one of these functions is the factorial function:

```
(cx:) ! 4
24
```

Neat? You don't have to do ![4] , and you can't because '!' expects one number, not a list of one number. Try ![4] and you'll get an error. Now you may be thinking "when will I be using a list and when will I not?". Well, you must think about how many things that function will need. If the function needs more than one peice of information, you'll be sending a list, but if the function is like '!', where it only needs one peice of information (a number in the factorial's case), you'll just give the argument bare. Rarely (or never) would you call a function and pass a list of one object like '[5]'. This should become natural to you very fast. There are great advantages to this symantic property of CX. For example, you can chain functions without the overhead of lists:

```
(cx:) ! ! 3
720
```

Ok, don't get too excited with the factorial thing- you can easily find the limitations of java's mathematics. This function chaining is used all the time in CX, just not with the same word repeated. Here is a cool line that uses chaining:

```
(cx:) eval @ parse "+ [2 3]"
```

The string is parsed, applied to '@' (that's explained later) and then evaluated. This line is a single statement, where the last thing on the line is either a function that takes no args, but produces output, or a literal itself. The information falls through the functions from the right to left, where the leftmost word was the action we wanted to take all along. It is important to be able to tell what a statement is because there are no semicolons in CX (like from java or C). With no semicolons, it is your responsibility to return lines after each statement in cx, though the code would work fine if you wrote it all on one line. Styling details are explained in Appendix A of this manual.

2.3 Slots, Quotes, and Functions

A slot is a conceptual space represented by a name. The space can hold a value, and that value may be changed, and the slot can be created from nothing, and destroyed. Slots are always identified by their name. The only time that two slots may have the same name is if they are in two different contexts. For now, we program in one context, so we'll just play along as though all slots must have a different name. When programs get large, it is best to separate the program in to chunks called contexts. Contexts are explained in a later section in this document.

To make a slot, you use the 'make' function. Pass a list with a string, and any value, respectively, and a slot will be made with the name correlating with the given string.

```
(cx:) make["x" 5]
(cx:) x
5
```

It's possible to make a slot with a string that has a space in it and still use it. To do this , you use the 'get' function. This is rarely, or perhaps never used.

```
(cx:) make[" y " "howdy"]
(cx:) get " y "
"howdy"
```

Now it's time to learn about quotes. Quotes are a literal in CX. They are representations of CX code in CX. They start with '' and end with '', and all the code inside of them is parsed, but not evaluated. Words that don't exist may be used in quotes, and quotes always print back with even spacing between words.

```
(cx:) {hah this is cx code}
{ hah this is cx code }
(cx:) make["my-quote" { some code here }]
(cx:) my-quote
{ some code here }
```

You might be thinking 'Why call it a slot? Why not just call it a variable?'. A slot might be a variable, but a slot can be a function too. If the slot is a function, when that slot is referred to in CX, the function is called. In this way, function calls and variable reference has the exact same syntax.

To make a function, we use the 'make' function like normal, but we set the slot to a special type of value. You should have noticed that setting a variable to a quote doesn't make a function. In the example above, 'my-quote' simply returned the quote. Making a function requires the use of the '@' function, which takes a quote as input, and returns that same quote, tagged as 'live'. When a live quote is referred to by slot name, it executes.

```
(cx:) make["say3" @{ 3 }]
(cx:) say3
3
(cx:) make["some-code" { +[2 3] }]
(cx:) some-code
{ + [ 2 3 ] }
(cx:) make["add2and3" @ some-code]
(cx:) add2and3
5
```

2.4 Lists

There is alot more to CX's lists than displayed in the earlier section. Lists in CX are like the parameter lists in many other languages. You can nest statements in lists, and the resulting list will have the returns of the statements:

```
(cx:) [ *[2 3]  -[9 4] ]
[ 6 5 ]
(cx:) *[ *[2 3] -[9 4] ]
30
```

There are ways to access the seperate elements of a list. CX uses cons cells to create these lists, so lisp developers will be familar with this part of CX. Cons cells are objects made up of two other objects. An illustration best describes how cons cells link. The following is an illustration of the list [1 2 3], as cons cells.

As you can see, cons cells are kind of recursive in nature. When this linked list of cons cells are printed out, a ’[’ is initially printed, then all of the red numbers, seen from left to right, then a ’]’ when null is found as the cdr of a cons cell. If a cons cell is the ’car’ of another cons cell, a ’[’ is printed. Remember: the car of a cons cell may be any object- the cdr must be another cons cell or null. When the cdr of a cons cell is null, a ’]’ is printed.

Once you understand the structure of linked lists, the words car and cdr should be easy to understand. car returns the car of a cons cell. cdr returns the cdr of a cons cell:

```
(cx:) car [ 1 2 3 ]
1
(cx:) cdr [ 1 2 3 ]
[ 2 3 ]
```

If it helps, you may think of car as being ’the first element of the list’ and cdr would be ’the rest of the list after the first element’. Then you may wonder: So what happens when we ask for the cdr of an empty list? cdr will return an empty list.

```
(cx:) cdr [ 1 ]
[ ]
(cx:) cdr [ ]
[ ]
```

The same thing happens when you ask for the car of an empty list:

```
(cx:) car [ ]
[ ]
```

Now, we can apply chaining to cons cells and be able to access any element of a list!

```
(cx:) car cdr  [ 1 2 3 ]
2
(cx:) car cdr cdr [ 1 2 3 ]
3
```

This isn’t convenient, is it? So we have another option:

```
(cx:) list-nth[[1 2] 1]
1
```

2.5 Collections

2.6 Contexts

Everything is a context in CX.

Chapter 3

Case Study:

Chapter 4

Libraries