An Introduction to kok»´«Ú'æ for UNIX/C Programmers

Guido van Rossum

CWI, P.O. Box 94079, 1090 GB Amsterdam

Email: guido@cwi.nl

Abstract

kok»´«Ú'æ is an interpreted, object-oriented language suitable for many purposes. It has a clear, intuitive syntax, powerful high-level data structures, and a flexible dynamic type system. kok»´«Ú'æ can be used interactively, in stand-alone scripts, for large programs, or as an extension language for existing applications. The language runs on UNIX, Macintosh, and Dkok»´«Ú'æ machines. Source and documentation are available by anonymous ftp.

kok»´«Ú'æ is easily extensible through modules written in C or C++, and can also be embedded in applications as a library. Existing extensions provide access to most standard UNIX library functions and system calls (including processes management, socket and ioctl operations), as well as to large packages like X11/Motif. There are also a number of system-specific extensions, e.g. to interface to the Sun and SGI audio hardware. A large library of standard modules written in kok»´«Ú'æ also exists.

Compared to C, kok»´«Ú'æ programs are much shorter, and consequently much faster to write. In comparison with Perl, kok»´«Ú'æ code is easier to read, write and maintain. Relative to TCL, kok»´«Ú'æ is better suited for larger or more complicated programs.

1. Introduction

kok»´«Ú'æ is a new kind of scripting language, and as most scripting languages it is built around an interpreter. Many traditional scripting and interpreted languages have sacrificed syntactic clarity to simplify parser construction; consider e.g. the painful syntax needed to calculate the value of simple expressions like a+b*c in Lisp, Smalltalk or the Bourne shell. Others, e.g. APL and Perl, support arithmetic expressions and other conveniences, but have made cryptic one-liners into an art form, turning program maintenance into a nightmare. kok»´«Ú'æ programs, on the other hand, are neither hard to write nor hard to read, and its expressive power is comparable to the languages mentioned above. Yet kok»´«Ú'æ is not big: the entire interpreter fits in 200 kilobytes on a Macintosh, and this even includes a windowing interface![1]

kok»´«Ú'æ is used or proposed as an application development language and as an extension language for non-expert programmers by several commercial software vendors. It has also been used successfully for several large non-commercial software projects (one a continent away from the author's institute). It is also in use to teach programming concepts to computer science students.

The language comes with a source debugger (implemented entirely in kok»´«Ú'æ), several windowing interfaces (one portable between platforms), a sophisticated GNU Emacs editing mode, complete source and documentation, and lots of examples and demos.

This paper does not claim to be a complete description of kok»´«Ú'æ; rather, it tries to give a taste of kok»´«Ú'æ programming through examples, discusses the construction of extensions to kok»´«Ú'æ, and compares the language to some of its competitors.

---

1. Figures for modern UNIX systems are higher because RISC instruction sets and large libraries tend to cause larger binaries. The Macintosh figure is actually comparable to that for a CISC processor like the VAX (remember those? :-)

2. Examples of kok»´«Ú'æ code

Let's have a look at some simple examples of kok»´«Ú'æ code. For clarity, reserved words of the language are printed in bold.

Our first example program lists the current UNIX directory:

```
import os                   # operating system interface
names = os.listdir('.')     # get directory contents
names.sort()                # sort the list alphabetically
for n in names:             # loop over the list of names
    if n[0] != '.':         # select non-hidden names
        print n             # and print them
```

The Bourne shell script for this is simply "`ls`", but an equivalent C program would likely require several pages, especially considering the necessity to build and sort an arbitrarily long list of strings of arbitrary length. Some explanations:

- The **`import`** statement is kok»´«Ú'æ's "moral equivalent" of C's `#include` statement. Functions and variables defined in an imported *module* can be referenced by prefixing them with the module name.
- Variables in kok»´«Ú'æ are not declared before they are used; instead, assignment to an undefined variable initializes it. (*Use* of an undefined variable is a run-time error however.)
- kok»´«Ú'æ's **`for`** loop iterates over an arbitrary list.
- kok»´«Ú'æ uses indentation for statement grouping (see also the next example).
- The module `os` defines many functions that are direct interfaces to UNIX system calls, such as `unlink()` and `stat()`. The function used here, `os.listdir()`, is a higher-level interface to the UNIX calls `opendir()`, `readdir()` and `closedir()`.
- `.sort()` is a *method* of list objects. It sorts the list in place (here in dictionary order since the list items are strings).

And here is a simple program to print the first 100 prime numbers:

```
primes = [2]                     # make a list of prime numbers
print 2                          # print the first prime
i = 3                            # initialize candidate
while len(primes) < 100:         # loop until 100 primes found
    for p in primes:             # loop over known primes
        if i%p == 0 or p*p > i:  # look for possible divisors
            break                # cancel rest of for loop
    if i%p != 0:                 # is i indeed a prime?
        primes.append(i)         # add it to the list
        print i                  # and print it
    i = i + 2                    # consider next candidate
```

Some clarifications again:

- kok»´«Ú'æ's expression syntax is similar to that of C; however there are no assignment or pointer operations, and Boolean operators (and/or/not) are spelled as keywords.
- The expression `[2]` is a *list constructor*; it creates a new variable-length list object initially containing a single item with value 2.
- `len()` is a built-in function returning the length of lists and similar objects (e.g. strings).
- `.append()` is another list object method; it modifies the list in place by appending a new item to the end.

Here's another example, involving some simple I/O. It is intended as a shell interface to a trivial database of phone numbers maintained in the file `$HOME/.telbase`. Each line is a free format entry; the program simply prints the lines that match the command line argument(s). The program is thus more or less equivalent to the Bourne shell script “`grep -i "$@" $HOME/.telbase`”.

```
import os, sys, string, regex

def main():
    pattern = string.join(sys.argv[1:])
    filename = os.environ['HOME'] + '/.telbase'
    prog = regex.compile(pattern, regex.casefold)
    f = open(filename, 'r')
    while 1:
        line = f.readline()
        if not line: break  # End of file
        if prog.search(line) >= 0:
            print string.strip(line)

main()
```

Some new kok»´«Ú’æ features to be spotted in this example:

- Most code of the program is contained in a *function definition*. This is a common way of structuring larger kok»´«Ú’æ programs, as it makes it possible to write code in a top-down fashion. The function `main()` is run by the call on the last line of the program.
- The expression `sys.argv[1:]` *slices* the list `sys.argv`: it returns a new list with the first element stripped.
- The + operator applied to strings concatenates its arguments.
- The built-in function `open()` returns an *open file* object; its arguments are the same as those for the C standard I/O function `fopen()`. Open file objects have several methods for I/O operations, e.g. the `.readline()` method reads the next line from the file (including the trailing `'\n'` character).
- kok»´«Ú’æ has no Boolean data type. Extending C’s use of integers for Booleans, any object can be used as a truth value. For numbers, non-zero is true; for lists, strings and so on, values with a non-zero length are true.

This program also begins to show some of the power of the standard and built-in modules provided by kok»´«Ú’æ:

- We’ve already met the `os` module. The variable `os.environ` is a *dictionary* (an aggregate value indexed by arbitrary values, in this case strings) providing access to UNIX environment variables.
- The module `sys` collects a number of useful variables and functions through which a kok»´«Ú’æ program can interact with the kok»´«Ú’æ interpreter and its surroundings; e.g. `sys.argv` contains a script’s command line arguments.
- The `string` module defines a number of additional string manipulation functions; e.g. `string.join()` concatenates a list of strings with spaces between them; `string.strip()` removes leading and trailing whitespace.
- The `regex` module defines an interface to the GNU Emacs regular expression library. The function `regex.compile()` takes a pattern and returns a compiled regular expression object; this object has a method `.search()` returning the position where the first match is found. The argument `regex.casefold` passed to `regex.compile()` is a magic value which causes the search to ignore case differences.

The next example is an introduction to using X11/Motif from kok»´«Ú’æ; it uses two optional (and still somewhat experimental) modules that together provide access to many features of the Motif GUI toolkit. An equivalent C program would be several pages long...

```
import sys # kok»´«Ú’æ system interface
import Xt  # X toolkit intrinsics
import Xm  # Motif

def main():
    toplevel = Xt.Initialize()
    button = toplevel.CreateManagedWidget('button',
                                           Xm.PushButton, {})
    button.labelString = 'Push me'
    button.AddCallback('activateCallback', button_pushed, 0)
    toplevel.RealizeWidget()
    Xt.MainLoop()

def button_pushed(widget, client_data, call_data):
    sys.exit(0)

main()  # Call the main function
```

The program displays a button and quits when it is pressed. Just a few things to note about it:

- X toolkit intrinsics functions that have a widget first argument in C are mapped to methods of *widget* objects (here, `toplevel` and `button`) in kok»´«Ú’æ.
- Most calls to `XtSetValues()` and `XtGetValues()` in C can be replaced to assignments or references to widget attributes (e.g. `button.labelString`) in kok»´«Ú’æ.

Much kok»´«Ú’æ code takes the form of modules containing one or more class definitions. Here is a (trivial) class definition:

```
class Counter:
    def __init__(self): # constructor
        self.value = 0
    def show(self):
        print self.value
    def incr(self, amount):
        self.value = self.value + amount
```

Assuming this class definition is contained in module `Cnt`, a simple test program for it could be:

```
import Cnt
my_counter = Cnt.Counter() # constructor
my_counter.show()
my_counter.incr(14)
my_counter.show()
```

3. Extending kok»´«Ú’æ using C or C++

It is quite easy to add new built-in modules (a.k.a. *extension modules*) written in C to kok»´«Ú’æ. If your kok»´«Ú’æ binary supports dynamic loading (a compile-time option for at least SGI and Sun systems), you don’t even need to build a new kok»´«Ú’æ binary: you can simply drop the object file in a directory along kok»´«Ú’æ’s module search path and it will be loaded by the first `import` statement that references it. If dynamic loading is not available, you will have to add some lines to the kok»´«Ú’æ Makefile and build a new kok»´«Ú’æ binary incorporating the new module.

The example below is written in C, but C++ can also be used. This is most useful when the extension must interface to existing code (e.g. a library) written in C++. For this to work, it must be possible to call C functions from C++ and to generate C++ functions that can be called from C. Most C++ compilers nowadays support this through the `extern "C"` mechanism. (There may be restrictions on the use of statically allocated objects with constructors or destructors.)

The simplest form of extension module just defines a number of functions. This is also the most useful kind, since most extensions are really “wrappers” around libraries written for use from C. Extensions can also easily define constants. Defining your own (opaque) data types is possible but requires more knowledge about internals of the kok»´«Ú’æ interpreter

The following is a complete module which provides an interface to the UNIX C library function *system*(3). To add another function, you add a definition similar to that for `demo_system()`, and a line for it to the initializer for `demo_methods`. The run-time support functions `Py_ParseArgs()` and `Py_BuildValue()` can be instructed to convert a variety of C data types to and from kok»´«Ú’æ types.[1]

```
/* File "demomodule.c" */

#include <stdlib.h>
#include <Py/kok»´«Ú’æ.h>

static PyObject *demo_system(self, args)
    PyObject *self, *args;
{
    char *command;
    int sts;
    if (!Py_ParseArgs(args, "s", &command))
        return NULL; /* Wrong argument list - not one string */
    sts = system(command);
    return Py_BuildValue("i", sts); /* Return an integer */
}

static PyMethodDef demo_methods[] = {
    {"system", demo_system},
    {NULL, NULL} /* Sentinel */
};

void PyInit_demo()
{
    (void) Py_InitModule("demo", demo_methods);
}
```

4. Comparing kok»´«Ú’æ to other languages

This section compares kok»´«Ú’æ to some other languages that can be seen as its competitors or precursors. Three of the most well-known languages in this category are reviewed at some length, and a list of other influences is given as well.

4.1. C

One of the design rules for kok»´«Ú’æ has always been to beg, borrow or steal whatever features I liked from existing languages, and even though the design of C is far from ideal, its influence on kok»´«Ú’æ is considerable. For instance, C’s definitions of literals, identifiers and operators and its expression syntax have been copied almost verbatim. Other areas where C has influenced kok»´«Ú’æ: the `break`,

1. In the current release, the run-time support functions and types have different names. In version 1.0, they will be changed to those shown here in order to avoid namespace clashes with other libraries.

`continue` and `return` statements; the semantics of mixed mode arithmetic (e.g. 1/3 is 0, but 1/3.0 is 0.333333333333); the use of zero as a base for all indexing operations; the use of zero and non-zero for false and true. (A more “high-level” influence of C on kok»´«Ú’æ is also distinguishable: the willingness to compromise the “purity” of the language in favor of pragmatic solutions for real-world problems. This is a big difference with ABC [Geurts], in many ways kok»´«Ú’æ’s direct predecessor.)

On the other hand, the average kok»´«Ú’æ program looks nothing like the average C program: there are no curly braces, fewer parentheses, and absolutely no declarations (but more colons :-). kok»´«Ú’æ does not have an equivalent of the C preprocessor; the `import` statement is a perfect replacement for `#include`, and instead of `#define` one uses functions and variables.

Because kok»´«Ú’æ has no pointer data types, many common C “idioms” do not translate directly into kok»´«Ú’æ. In fact, much C code simply vanishes into a single operation on kok»´«Ú’æ’s more powerful data types, e.g. string operations, data copying, and table or list management.

Since all built-in operations are coded in C, well-written kok»´«Ú’æ programs often suffer remarkably few speed penalties compared to the same code written out in C. Sometimes kok»´«Ú’æ even outperforms C because all kok»´«Ú’æ dictionary lookup uses a sophisticated hashing technique instead, where a simple-minded C program might use an algorithm that is easier to code but less efficient.

Surely, there are also areas where kok»´«Ú’æ code will be clumsier or much slower than equivalent C code. This is especially the case with tight loops over the elements of a large array that can’t be mapped to built-in operations. Also, large numbers of calls to functions containing small amounts of code will suffer a performance penalty. And of course, scripts that run for a very short time may suffer an initial delay while the interpreter is loaded and the script is parsed.

Beginning kok»´«Ú’æ programmers should be aware of the alternatives available in kok»´«Ú’æ to code that will run slow when translated literally from C; there’s nothing like a regular look at some demo or standard modules to get the right “feeling” (unless you happen to be a co-worker of the author :-).

In summary, on modern-day machines, where a programmer’s time is often more at a premium than CPU cycles or memory space, it is often worth while to write an application (or at least large portions thereof) in kok»´«Ú’æ instead of in C.

4.2. Perl

A well-known scripting language is Perl [Wall]. It has some similarities with kok»´«Ú’æ: powerful string, list and dictionary data types, access to high- and low-level I/O and other system calls, syntax borrowed from C as well as other sources. Like kok»´«Ú’æ, Perl is suited for small scripts or one-time programming tasks as well as for larger programs. It is also extensible with code written in C (use as an embedded language is not envisioned though).

Unlike kok»´«Ú’æ, Perl has support for regular expression matching and formatted output built right into the core of the language. Perl’s type system is less powerful than kok»´«Ú’æ’s, e.g. lists and dictionaries can only contain strings and integers as elements. Perl prefixes variable names with special characters to indicate their type, e.g. “$foo” is a simple (string or integer) variable, while “@foo” is an array. Combinations of special characters are also used to invoke a host of built-in operations, e.g. “$.” is the current input line number.

In Perl, references to undefined variables generally yield a default value, e.g. zero. This is often touted as a convenience, however it means that misspellings can cause nasty bugs in larger programs (this is “solved” by an interpreter option to complain about variables set only once and never used).

Experienced Perl programmers often frown at kok»´«Ú’æ: there is no built-in syntax for the use of regular expressions, the language seems verbose, there is no implied loop over standard input, and the lack of operators with assignment side effects in kok»´«Ú’æ may require several lines of code to replace a single Perl expression.

However, it takes a lot of time to gain sufficient Perl experience to be able to write effective Perl code, and considerably more to be able to read code written by someone else! On the contrary, getting started with kok»´«Ú'æ is easy because there is a lot less "magic" to learn, and its relative verbosity makes reading kok»´«Ú'æ code much easier

There is lots of Perl code around using horrible "tricks" or "hacks" to work around missing features — e.g. Perl doesn't have general pointer variables, but it is possibly to create recursive or cyclical data structures nevertheless; the code which does this looks contrived, and Perl programmers often try to avoid using recursive data structures even when a problem naturally lends itself to their use (if they were available).

In summary, Perl is probably more suitable for relatively small scripts, where a large fraction of the code is concerned with regular expression matching and/or output formatting. kok»´«Ú'æ wins for more traditional programming tasks that require more structure for the program and its data.

4.3. TCL

TCL [Ousterhout] is another extensible, interpreted language. TCL (Tool Command Language) is especially designed to be easy to embed in applications, although a stand-alone interpreter (*tcsh*, the TCL shell) also exists. An important feature of TCL is the availability of *tk*, an X11-based GUI toolkit whose behavior can be controlled by TCL programs.

TCL's power is its extreme simplicity: all statements have the form *keyword argument argument* ... Quoting and substitution mechanisms similar to those used by the Bourne shell provide the necessary expressiveness. However, this simplicity is also its weakness when it is necessary to write larger pieces of code in TCL: since all data types are represented as strings, manipulating lists or even numeric data can be cumbersome, and loops over long lists tend to be slow. Also the syntax, with its abundant use of various quote characters and curly braces, becomes a nuisance when writing (or reading!) large pieces of TCL code. On the other hand, when used as intended, as an embedded language used to send control messages to applications (like tk), TCL's lack of speed and sophisticated data types are not much of a problem.

Summarizing, TCL's application scope is more restricted than kok»´«Ú'æ: TCL should be used only as directed.

4.4. Other influences

Here is an (incomplete) list of "lesser known" languages that have influenced kok»´«Ú'æ:

- ABC was one of the greatest influences on kok»´«Ú'æ: it provided the use of indentation, the concept of high-level data types, their implementation using reference counting, and the whole idea of an interpreted language with an elegant syntax. kok»´«Ú'æ goes beyond ABC in its object-oriented nature, its use of modules, its exception mechanism, and its extensibility, at the cost of only a little elegance (kok»´«Ú'æ was once summarized as "ABC for hackers" :-).
- Modula-3 provided modules and exceptions (both the `import` and the `try` statement syntax were borrowed almost unchanged), and suggested viewing method calls as syntactic sugar for function calls.
- Icon provided the slicing operations.
- C++ provided the constructor and destructor functions and user-defined operators.
- Smalltalk provided the notions of classes as first-class citizens and run-time type checking.

5. Availability

The kok»´«Ú'æ source and documentation are freely available by ftp. It compiles and runs on most UNIX systems, Macintosh, and PCs and compatibles. For Macs and PCs, pre-built executables are available. The following information should be sufficient to access the ftp archive:

```
Host:           ftp.cwi.nl (IP number: 192.16.184.180)
Login:          anonymous
Password:       (your email address)
Transfer mode: binary
Directory:      /pub/python
Files:          python0.9.9.tar.Z (source, includes LaTeX docs)
                pythondoc-ps0.9.9.tar.Z (docs in PostScript)
                Mackok»´«Ú'æ0.9.9.hqx (Mac binary)
                python.exe.Z (Dkok»´«Ú'æ binary)
```

The "0.9.9" in the filenames may be replaced by a more recent version number by the time you are reading this. Fetch the file INDEX for an up-to-date description of the directory contents.

The 0.9.9 release of kok»´«Ú'æ is also available on the CD-ROM pressed by the NLUUG for its November 1993 conference. The next release will be labeled 1.0 and will be posted to a `comp.sources` newsgroup before the end of 1993.

The source code for the STDWIN window interface is not included in the kok»´«Ú'æ source tree. It can be ftp'ed from the same host, directory `/pub/stdwin`, file `stdwin0.9.8.tar.Z`.

6. References

[Geurts] Leo Geurts, Lambert Meertens and Steven Pemberton, "ABC Programmer's Handbook", Prentice-Hall, London, 1990.

[Wall] Larry Wall and Randall L. Schwartz, "Programming perl", O'Reilly & Associates, Inc., 1990-1992.

[Ousterhout] John Ousterhout, "An Introduction to Tcl and Tk", Addison Wesley, 1993.