

An Introduction to the Yoix™ Interpreter

Richard Drechsler and John Mocenigo

AT&T Labs - Research

Florham Park, New Jersey

November 1, 2000

Abstract

The Yoix interpreter is built using 100% pure Java™ technology and its language includes the best bits and many familiar constructs from both Java and C plus a few twists of its own. Because it uses Java technology, it is cross-platform, GUI-capable and both network and thread friendly, yet Yoix users are spared more cumbersome and tricky parts of Java programming. The Yoix language includes pointers, addressing, declarations, and global and local variables. The Yoix interpreter supports the important data types and methods from the Java environment and also provides many familiar C library routines, letting you use them the way you would in C, including functions such as *fprintf* and *fscanf*, the latter of which uses pointers and addressing. The apparent handicap of being an interpreted language on top of an interpreted language is not an issue because of careful coding and the sorts of processors commonly available in the last few years, let alone now. Since January 1999, the Yoix interpreter and its prototype implementation have been successfully powering the Global Fraud Management System (GFMS), a 24/7 system with over 500 users in several locations.

Introduction

Its many features have made the Java language very appealing for application development, but there is no denying that cumbersome coding is sometimes required for even the simplest tasks. Consider what is required for the archetypal program "Hello World" and how much nicer it would be to still have the many nice features of Java, but in a language where:

```
stdout.nextline = "Hello, World.";
```

or

```
yoix.stdio.printf("Hello, World.\n");
```

is the complete "Hello World" program.

The Yoix interpreter is our attempt to provide a way of developing robust, cross-platform, possibly networked applications more rapidly and with less effort than through other available approaches. Because the Yoix interpreter is a Java¹ application, it provides most of the features that Java users have come to know and love such as URL connectivity, sockets, threads, process execution and GUI components. As an example of the power and ease of Yoix programming, the following script reads the AT&T home page and writes it to standard output:

```
URL att = yoix.io.open("http://www.att.com/", "r");  
while(line = att.nextline) stdout.nextline = line;
```

Moreover, because the Yoix interpreter contains many familiar C library functions, it is just as easy to perform more interesting tasks. For example, the following script extracts just the HTML directives and comments from the AT&T home page:

```
import yoix.io.open;
import yoix.stdio.*;
URL att = open("http://www.att.com/", "r");
String text[128,...];
int cnt = 0;
while(cnt >= 0) {
    if((cnt = fscanf(att, " <^[^>]>", text)) > 0)
        printf("<%s>\n", text);
    else
        cnt = fscanf(att, " %*s"); // discard
}
```

A powerful feature of the Yoix interpreter is that it can be directed to read its initial and subsequent input from a URL, thus it is possible to have applications downloaded from a server at runtime, as needed, much as a browser downloads and interprets HTML pages at runtime. Unlike browsers, however, the Yoix interpreter provides access to the equivalent of a robust, full-featured Java application and that application can be updated by simply changing compact, quick-to-download, ASCII files on a server. In other words, once the Yoix jar file is distributed to end-users (usually from a departmental file server), distribution of new Yoix applications is trivial (simply provide a one-line script that starts the interpreter directed to the URL of the new application) and updates are even simpler (add, replace or remove Yoix script files from a web server).

This paper is intended as a brief introduction to Yoix technology and its possibilities. The following sections highlight features of the Yoix language and interpreter. Short examples are sprinkled throughout the paper, which wraps up with a slightly longer example of a simple GUI chat application.

The Yoix Language

The Yoix language has good variety of basic data types and an extended set of secondary data types. In many instances, a type name corresponds directly to the Java class name used to implement it as an aid to Yoix users familiar with Java programming. Each data type maintains a dictionary of information about itself, the complexity of which depends on the complexity of the type. Users wishing to learn about the components of a particular type or the current values of those components can request a dump of a data type instance in any of several ways and most easily by applying the `toString()` built-in to the object and printing out the result. Table I on the next page lists the current data types.

The language supports declarations, pointers and addressing, but there is no way to say a variable is a pointer to a particular type, which means a declaration like:

```
int *ptr;
```

parses as a multiplication of variables named **int** and **ptr**. However, one can work with pointers in ways familiar to C programmers. For example:

```
String str = "This is a Java application.";
yoix.string.overlay(str+10,"Yoix");
*(str + yoix.string.strlen(str) - 1) = '!';
yoix.stdio.puts(str);
```

yields:

```
This is a Yoix application!
```

Thus, the Yoix language is for anyone who misses pointers, but not access violations.

Table I: Yoix Data Types

Basic Types:

```
Object
Callable
  Builtin
  Function
Number
  int
  double
Pointer
Array
Dictionary
Matrix
Stream
String
```

Secondary Types:

Miscellaneous Objects :

```
Calendar, Color, Dimension,
Displacement, Font, Insets,
Locale, Menu, Point, Process,
Rectangle, Regexp,
SecurityManager, ServerSocket,
Socket, Subexp, Thread, TimeZone,
Tree
```

Stream Objects:

```
File, StringStream, URL
```

LayoutManager Objects:

```
BorderLayout, CardLayout,
CustomLayout, FlowLayout,
GridBagLayout, GridLayout
```

Component Objects:

```
Button, Canvas, Checkbox,
CheckboxGroup, Choice, Dialog,
FileDialog, Frame, Label, List,
MenuBar, Panel, PopupMenu, Scrollbar,
ScrollPane, TableColumn,
TableManager, TextArea, TextCanvas,
TextField, TextTerm, Window
```

Event Objects:

```
ActionEvent, AdjustmentEvent,
ComponentEvent, FocusEvent,
ItemEvent, KeyEvent, MouseEvent,
PaintEvent, TextEvent, WindowEvent
```

There are also occasions for using an ampersand to indicate a pointer to an object. For example:

```
Dictionary title = {
    String a;
    int b;
    String c[2,...];
};
int year;
yoix.stdio.sscanf("The 39 Steps (1935)", "%s %d %s (%d)",
    &title.a, &title.b, title.c, &year);
yoix.stdio.printf("a='%s', b='%d', c='%s', year='%d'\n",
    title.a, title.b, title.c, year);
yoix.stdio.printf("a='%s', b='%d', c='%s', year='%d'\n",
    title[0], title[1], title[2], year);
```

yields:

```
a='The', b='39', c='Steps', year='1935'
a='The', b='39', c='Steps', year='1935'
```

The above example illustrates several points that are worth discussing. Firstly, we see a *Dictionary* being declared and used. Dictionaries are widely used by the Yoix interpreter. For example, the **sscanf** and **printf** built-ins are each elements in the dictionary **stdio**, which in turn an element in the global dictionary **yoix**². Elements in a dictionary can be accessed by name or by position. Also of interest are the *String* declarations. Element **a** is declared with no size

specified and so is initially just a *NULL* of type *String*. Element *c* is declared to be of size 2 and the ellipses indicate that it is allowed to grow as needed. In this case, it needed to grow to size 5. A declaration of "*c*[2]" would have fixed *c* at size 2 and the call to **sscanf** would have failed with a **rangecheck** error. A declaration of "*c*[2,8]" would have set the size of *c* to 2 and allowed it to grow to size 8, if needed. The ampersands before the dictionary elements *a* and *b* in the **sscanf** call allow the function to assign the appropriate objects from its string conversion to those variables. The ampersand is absent from variable **title.c** to indicate to **sscanf** that it should write the appropriate conversion into the existing storage of that variable. No **rangecheck** error occurs in this case, as it would have if the ampersand had been absent from the *NULL* variable **title.a**, because **title.c** has storage space available to accept content.

Programmers familiar with languages such as C or Java should feel rather comfortable with the Yoix language. All the familiar C and Java language directives, like *for*, *while*, *switch* and so on, are there. Moreover, a *try* and *catch* construct is available, though return value checking can often obviate the need for them, if desired. As mentioned, the equivalent of many familiar C functions are available, including regular expressions, and many Java method equivalents are provided as well. In addition, users can easily extend the language by adding their own built-in routines.

The Yoix Interpreter

The Yoix interpreter consists of 110 Java class files that, when optimized, fit into a *jar* file of nearly 465,000 bytes. The language parser was written using Version 1.1 of *JavaCC*, a product of Sun Microsystems that is now distributed and supported by Metamata. The interpreter is invoked using the *java* command or the *jre* command. An example using the Java Runtime Environment³ in a UNIX-like environment is:

```
CLASSPATH=$JAVAHOME/lib/classes.zip:$YOIXHOME/lib/yoix.jar
jre att.research.yoix.YoixMain [options] [script_path]
```

A discussion of the interpreters command line *options* goes beyond the scope of this paper. The *script_path*, as noted earlier, can indicate a file, a URL or, if absent, the standard input.

The Yoix interpreter reads and executes one statement at a time, very much like a Unix shell. The parse tree for each statement is built by *JavaCC* and *JITree* and executed by code that knows how to traverse the tree. When statement execution ends, the tree disappears and the interpreter moves on to the next statement as you would expect from a scripting language. The simple Yoix grammar, however, means the parser only catches obvious syntax errors and it is left to the strong, runtime type checking capabilities of the interpreter to catch any other errors. When runtime errors occur, the Yoix interpreter's response is robust: it unwinds the stack to the start of the offending operation, provides diagnostics or triggers a *catch* response, as appropriate, and moves on.

The interpreter's dynamic checking is about as far from Java as you can get, particularly because there is no Yoix equivalent of the Java compiler to rigorously screen the code for errors. The difference is significant, but there is no best approach. Java programs have not replaced shell and perl scripts. The Yoix language does have within it, though, the seeds to bridge this difference: the *Tree* type provides access to Yoix parse trees. That ability will allow us to write, as a Yoix script, a program checker to find potential errors in Yoix scripts prior to runtime.

Finally we should note that the Yoix interpreter supports threads, but uses a static parser from *JavaCC*, so access to that parser has to be synchronized and carefully controlled by the interpreter. Threads that execute Yoix code get their own private stack that is used for expression evaluation, scoping, error control, and more. The approach is obvious, but coming up with an efficient, thread-safe Java implementation was a challenging task.

What Distinguishes Yoix Technology?

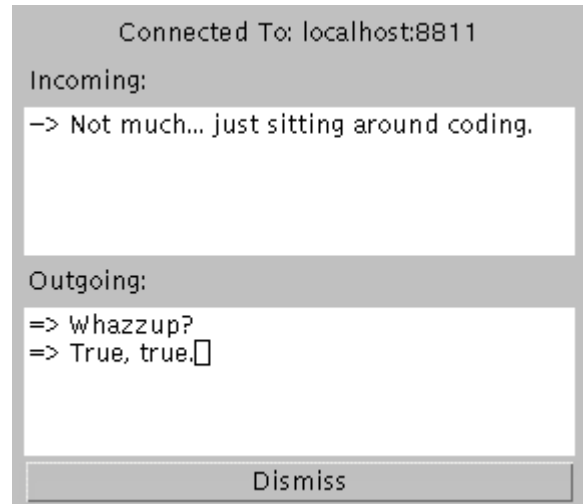
With so many useful scripting languages already available, why introduce a Yoix variant into the mix? We believe the Yoix feature set not only allows rapid development of easy-to-maintain, networked, cross-platform applications, but also that there is no scripting language currently available that does what the Yoix interpreter does as well as the Yoix interpreter does it. Why?

Firstly, Yoix technology is 100% pure Java, which distinguishes it from valuable and familiar scripting languages such as Korn shell, perl and tcl, which have been ported to a variety of platforms, but are not intrinsically cross-platform in the manner of Java applications. A carefully written Java application, such as the Yoix interpreter, can run unmodified wherever the Java Virtual Machine (JVM) runs. Moreover, there are a variety of resources committed to making sure that the JVM runs just about everywhere and, by being a Java application, the Yoix interpreter directly reaps the benefits of those efforts.

Of course, these days, being a Java based programming language is not a rarity. One site on the web⁴ catalogues over 130 programming languages using Java technology with a little over a dozen of those being considered scripting languages. What does the Yoix scripting language bring to this party? For one thing, it is a language with a well-developed grammar which in many instances uses familiar C-style syntax. Secondly, it is an industrial strength tool written to perform robustly in real-life, large-scale applications. As a simple illustration of its cross-platform reliability, consider the screen resolution problem. On Microsoft platforms, for reasons beyond the control of the JVM, the *java.awt.Toolkit* `getScreenResolution()` method does not return a meaningful number. Though the Yoix interpreter cannot fix this limitation on its own, it does provide a solution. When the diagonal measurement of the monitor screen is provided as a command line argument, the interpreter sets up a reliable 72 dots per inch coordinate system. Additionally, when accessing underlying Java capabilities such as GUI components, sockets, threads and so on, the interpreter does not simply pass-through directives, but rather provides value-added processing that includes a consistent interface and avoidance of common Java pitfalls, of which, even for fans of the language, there are an annoying number. For example, in a Java program, if you request a default *GridLayout*, namely call the constructor with no arguments, and then want to specify a column-only layout, you must specify the non-zero column value before specifying the zero row value to avoid an exception. Specifically, the order in which you call the *GridLayout* `setColumns()` and `setRows()` methods is important in certain instances. The Yoix interpreter protects users from this capricious exception-generating pitfall; other pass-through languages, such as *JPython*, do not. Similarly, some Java components specify layout using **LEFT**, **CENTER**, **RIGHT**, while others use **WEST**, **CENTER**, **EAST**. The Yoix interpreter accepts either and does not require that one remembers where **LEFT** or whatever is specifically defined, but rather defines the terms globally. Although these examples may seem unimportant, these and similar problems can make the *rapid* development of a *robust* cross-platform application impossible.

Example: A Simple, Two-Party Chat Application

We like to trumpet Yoix technology as great for rapid application development and, certainly, that was our experience when we used it for the GUI of the Global Fraud Management System (GFMS). In an attempt to impart a sense of this technology's value to the reader in the context of a brief paper, we cobbled together a simple, two-party chat application to illustrate some of its GUI, socket and thread capabilities in a terse, one-page script. An image of the running application is shown to the right. The complete, line-numbered script can be found in the Appendix. Note that the other end of the chat need not be this same chat application. A standard telnet program, for example, will work just as well.



When started, the program waits, listening on a socket port specified on the command line, until a connection is requested, at which point the connection is moved to another port. The program then pops up the GUI window and also starts a thread that reads from the new socket connection and writes to the upper text region in the GUI. Meanwhile, the original thread waits for the user to type into the lower text region in the GUI and, as each new-line is entered, writes the text to the socket connection. When the user presses the "Dismiss" button or the remote-side disconnects, the thread is killed and the program exits.

Working through the code in the Appendix, we see that lines 1 - 6 specify `GridBagConstraints` parameters that are needed repeatedly in what follows and so are specified one time in the `GBC` variable. Throughout the GUI portions of the language, data types are named so as to clearly refer to their Java counterparts, thereby making it easier for those familiar with the Java AWT to know what those data types represent and how they function. In this case, we are specifying `GridBagConstraints` used for `GridBagLayout` purposes. Line 7 begins specification of the main display `Frame` shown above. Its title, size and type of layout manager are specified (lines 8 - 10) and then an array of six components to be laid out within the `Frame` begins at line 11. Each component specification is followed by the `GridBagConstraints` that apply to it (lines 16, 18, 25, 27, 35 and 43), which in this case are the same for all the components since the layout is a straightforward top-to-bottom arrangement. The `Label` specifications (lines 12 - 15, 17 and 26) need no further description and the `Button` specification (lines 36 - 42) is only interesting in that it shows how easily event handling can be specified (lines 38 - 41). The `TextTerm` specifications refer to a GUI component supplied by the interpreter⁵ and not part of the Java AWT package. The `TextTerm` component is meant for situations where terminal-like behavior in a text region is needed. Setting `edit` to `FALSE` (line 22) in the *incoming* `TextTerm` (lines 19 - 24) makes it read-only. Specifying a body for the newline function (line 33) in the *outgoing* `TextTerm` (lines 28 - 34) tells the component what to do each time a new-line is entered, which in this case causes the text to be written to the socket stream. That is all there is to the GUI specification, which ends at line 45. After two lines of declarations (lines 46 and 47), socket handling occurs based on the

command line arguments (lines 48 - 58). When only a port number is supplied, the script is in server mode, waiting for a client to connect for a chat. In that case, the socket set-up requires two lines of code: setting the port number of the server (line 49) and waiting to accept a client request (line 50). The `accept` function will block until a client connects, at which point it returns the new connection. When both a host and port are supplied to the script, it is then in client mode and three lines are required to make the connection: two lines to set the address and port of the server (lines 52 and 53) and one line to enable the connection (line 54). If the wrong number of arguments is supplied, an error message is printed and the script exits (lines 56 and 57). Note that `argv[0]` contains the script source name. Now that the socket connection is made, the label text can be set and the frame made visible (lines 59 and 60). The script wraps up with some thread code. The input from the socket connection needs to be read by a separate thread to keep read-blocking out of the main thread. A thread is defined (lines 61 - 67) with its `run` function set to read one line at a time from the socket (line 63) and write the line to the *incoming* `TextTerm` component (line 64). When socket input ends, the thread exits (line 65). The only thing left to do in the script is to start the thread running (line 68).

Concluding Remarks

We have tried to give some indication in this limited space of the power and simplicity of the Yoix interpreter, and its suitability for production system development. The Yoix language and interpreter provide a familiar syntax in a cross-platform tool whose wealth of features include GUI components, sockets, threads, URL connectivity and much more. The Yoix distribution is available at <http://www.research.att.com/sw/tools/yoix>.

Footnotes

- 1 The initial release of the Yoix interpreter is written entirely with JDK1.1, though it compiles and runs using JDK1.2 and 1.3. Future releases of the interpreter will seek to take advantage of features in the later releases of the JDK, such as Swing and Java2D.
- 2 As an earlier example showed without fanfare, it is possible to import explicit dictionary elements (e.g., `import yoix.io.open;`) or all elements in a dictionary (e.g., `import yoix.stdio.*;`) into a script to reduce typing.
- 3 In practice the `jre "-classpath"` option would be used to specify the `CLASSPATH`, but its use would have resulted in too long a line for the example in this document.
- 4 The site, compiled and maintained by Robert Tolksdorf, is located at:
<http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html>
- 5 In addition to `TextTerm`, Yoix provides a few other custom GUI components: `TableColumn` and `TableManager` for displaying tables and `TextCanvas` for flexibly displaying text messages.

Yoix is a trademark of AT&T Corp.

Java is a trademark of Sun Microsystems.

Appendix: Two-Party GUI Chat Example (Complete Yoix Script)

```
1: final GridBagConstraints GBC = {
2:     double      weightx = 1;
3:     int          fill = yoix.awt.HORIZONTAL;
4:     int          gridwidth = yoix.awt.REMAINDER;
5:     Insets      insets = { int left = 6, right = 6, top = 1, bottom = 1; };
6: };
7: Frame chatscreen = {
8:     String      title = "YOIX-CHAT";
9:     Dimension   size = { int width = 325, height = 275; };
10:    GridBagLayout layoutmanager;
11:    Array layout = {
12:        new Label {
13:            String      componentname = "label";
14:            int          alignment = yoix.awt.CENTER;
15:        },
16:        GBC,
17:        new Label { String text = "Incoming:"; },
18:        GBC,
19:        new TextTerm {
20:            String      componentname = "incoming";
21:            Color       background = yoix.awt.Color.white;
22:            int         edit = FALSE;
23:            int         rows = 5;
24:        },
25:        GBC,
26:        new Label { String text = "Outgoing:"; },
27:        GBC,
28:        new TextTerm {
29:            String      componentname = "outgoing";
30:            Color       background = yoix.awt.Color.white;
31:            String      prompt = "=> ";
32:            int         rows = 5;
33:            newline(text) { socket.output.nextline = text; }
34:        },
35:        GBC,
36:        new Button {
37:            String      text = "Dismiss";
38:            actionPerformed(e) {
39:                outputthread.alive = FALSE;
40:                exit(0);
41:            }
42:        },
43:        GBC,
44:    };
45: };
46: ServerSocket  server;
47: Socket        socket;
48: if (argc == 2) { // we're the server
49:     server.port = yoix.string.atoi(argv[1]);
50:     socket = yoix.net.accept(server);
51: } else if (argc == 3) {
52:     socket.address = argv[1];
53:     socket.port = yoix.string.atoi(argv[2]);
54:     socket.alive = TRUE;
55: } else {
56:     yoix.stdio.fprintf(stderr, "ERROR(%s): arguments: [host] port\n", argv[0]);
57:     exit(1);
58: }
59: chatscreen.components.label.text = "Connected To: " + socket.name;
60: chatscreen.visible = TRUE;
61: Thread  outputthread = {
62:     run() {
63:         while ((line = socket.input.nextline) != NULL)
64:             yoix.awt.appendText("-> " + line, chatscreen.components.incoming);
65:         exit(0);
66:     }
67: };
68: outputthread.run();
```