wok::Interpreter
Written by Walter O. Krawec

Introduction

Why the name? In 2005 I started writing library routines that I commonly used, all encapsulated within the namespace “wok” (my initials). This naming scheme stuck with me and this Interpreter is the latest of those routines; unless I decide on a different name, I refer to it by its class name within the wok namespace, or wok::Interpreter. By itself it is a very minimal programming language only supporting the manipulation of integers and doubles, along with common commands like “while” loops or creating arrays. It is however capable of being easily extended to support other commands which is what makes it, in my opinion, very useful. I have begun using it extensively in my AI research, there is a module for a robot simulator, and it is also the foundation of the programming language Q-Prog.

This document only describes the base wok::Interpreter. For information on other modules, see the documentation included with those modules. We will first describe the syntax of this language (similar to Lisp/Scheme). Following this we will list all currently supported commands and their values; we will then mention how to extend its functionality (there are two methods: via statically linking a new command library, or, recommended, dynamically linking a command library).

Syntax

Every program in wok::Interpreter is of the form:

PROGRAM ::= CONSTANT | COMMAND (ARG1) (ARG2) … (ARG_n)

That is to say, a program is either a single word (translated to a constant) or a command followed by a list of arguments. Each argument is actually a program itself (that is, each argument is either a constant or a command followed by its own arguments). The "value of" a PROGRAM is simply the output of that program; it may be an integer, double, string, UserType, or NULL (meaning 'no value'). The value of a CONSTANT is that CONSTANT; the value of a COMMAND is the output of that COMMAND given the value of each ARG. (Note that one of the commands supported is the "begin" command which allows you to write a programs that run sequentially).

wok::Interpreter supports typed variables. Currently supported types are: integer, double, string, and UserType. Usually for the first three types, a CONSTANT is converted into one of these types (UserType types are complicated and are discussed later; they permit a user to define new types when writing a new command library). If a program consists of only a single word, the value of that program is an integer if it consists only of decimals (or a '-' in front); a double if it is only digits with a single '.' mark; else it is converted to a string.

Example: The value of the program "123" (without quotes) is 123; the value of "test" is the string "test"; the value of "test 1 2 3" is undefined since "test" is not a command.

When run, wok::Interpreter should present you with a prompt ">" from which you may write programs (type "exit" (without quotes) and press ENTER to quit). One of the supported commands is "+" (without the quotes). It's defined as follows:

Command: + (Value1 <INTEGER>) (Value 2<INTEGER>)
Description: Will add Value1 to Value 2
Value: Value1 + Value2

The above block tells us that the value of the program + (VALUE1) (VALUE2) is the integer VALUE1+VALUE2. It also tells us that VALUE1 and VALUE2 must be integer types. At the prompt ">", try the following program (type each line then press ENTER to run the program)

```
+ (1) (2)
```

You should get an integer output equal to "3". Note that, for a CONSTANT, there is no need to include the parenthesis. Hence we could write the above program as:

```
+ 1 2
```

With spaces between the "+" the "1" and the "2". Here are some other programs:

```
+ (+ 1 2) (+ 3 4)
```

```
+ (+ (+1 2) 3) 4
```

Both should evaluate to the integer "10".

One may also create new variables using the "new" command; these variables are manipulated using the "get" and "set" commands:

Command: new (type <STRING>) (name <STRING>)
Description: creates a new variable of the specified type and name.
Value: NULL

Command: get (name <STRING>)
Description: Will return the value of the variable "name"
Value: the value of the variable "name"

Command: set (name <STRING>) (Value <PROGRAM>)
Description: Will set the value of the variable "name" to the value of "Value". Returns an error if types don't match.
Value: NULL

Type may be "integer", "double", or "string" (without the quotes). Other types may be added later if needed. However, now we have an issue: if the program is only this single command, how do we use this new variable? We next introduce the "begin" command:

Command: begin (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)
Description: Will evaluate each argument in sequential order
Value: The value of ARG_n (the last run command)

Also the print command we'll use in the next example:

Command: print (Value <PROGRAM>)
Description: Will print to the console screen, the value of "Value"
Value: NULL

Here is a simple program:

begin (new integer j) (new integer k) (set j 1) (set k 2) print (+ (get j) (get k))

The value of this program is NULL however the integer "3" is printed to the screen. Two new integer variables are created, one is set to "1", the other to "2". The print command will evaluate the program + (get j) (get k) and print it's value (which is 1+2 = 3). Here is another program:

begin (new integer j) (set j 1) (print j)

The value of this program is also NULL but the string "j" is printed to the screen. That is because the print command prints the value of its argument; its argument is "j" which is a constant whose value is the string "j". If we changed this command to (print (get j)) then the program would print the integer "1". One last example:

begin (new integer j) (new integer k) (set j 1) (set k 2) print (+ (get j) (get k)) (100)

The integer “3” is again printed to the screen however now the value of the program is the integer 100 (since the value of a “begin” command is the value of the last command).

One last point we’ll make is that wok::Interpreter may run in two modes: regular and global. Regular (which is default) will keep variables only within their scope (inside a begin block for example but once that begin block is finished the variable is freed). When running in global mode any variable created is added to a global store; this variable will exist even when the program is finished (so multiple programs may access the variable). Sometimes it might be useful to create variables in global mode in one program (a setup program), and then have a second program actually use them. Global mode may be accessed using the global command:

Command: global (ARG <PROGRAM>)
Description: Will evaluate the program “ARG” in global mode
Value: value of ARG

For example:

global (new integer j)

will create a new integer “j” on the global store. After running the above program, type “!store” (without the quotes) at the prompt. You should see (Integer, j) printed. Use “get” or “set” to manipulated this variable. To clear the global store, use the “!clear” command. Note that any command with a “!” mark is a “shell” command and cannot be used in a program.

We next look at the if command:

Command: if (Test <INTEGER>) (True <PROGRAM>) (False <PROGRAM>)
Description: Will first evaluate “Test”; if this is not “0”, will evaluate and return the value of “True”; otherwise will evaluate and return the value of “False”
Value: The value of “True” or “False” depending on the value of “Test”

Command: < (Value1 <PROGRAM>) (Value2 <PROGRAM>)
Description: Will evaluate Value1 and Value2 and compare these. If they are not comparable (e.g. a Value1 is a string, Value2 is an integer), an error is returned.
Value: 1 if value of Value1 < value of Value2; 0 otherwise

Here is an example:

```
begin
        (new integer j) (new integer k)
        (set j 0) (set k 1)
        (if (< (get j) (get k))
                (print (“j is less than k”))
                (print (“j is not less than k”))
        )
```

The value of the program is NULL (since print is NULL); however the string "j is less than k" should be printed to the screen. If you change the value of (set j 0) to (set j 1), the string "j is not less than k" should be printed. Note that you can include a begin command in the True/False arguments of the if command. Also, if you don't need one of the True/False arguments (for instance if you don't need to do anything on account of a True of False condition), you may use the NOP command (which is also useful for writing comments):

Command: NOP (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)
Description: Will ignore each ARG (they are not evaluated) and do nothing.
Value: NULL

Command: nop (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)
Description: Same as NOP just lower-case depending on user preferences
Value: NULL

```
begin
        (NOP
                This is a comment; anything between the two parentheses is ignored.
        )
        (if 1
                (begin
                        (print ("Always true, value of program is 1+2=3"))
                        (+ 1 2)
                )
                (NOP)
        )
```

Finally, you may type programs in a text editor and run them from the prompt using the "file" command:

Command: file (Filename <STRING>)
Description: Will load in the program located at Filename and evaluate it
Value: value of the program located at Filename

Note: You should place a space between every argument. Within a parenthesis however there should be no spaces between the '(' and your text nor between the ')' and your text. For example use (1) but not (1) or (1).

Commands

We now list the supported commands in alphabetical order:

Command: + (Value1 <INTEGER>) (Value 2<INTEGER>)
Description: Will add Value1 to Value 2
Value: Value1 + Value2

Command: - (Value1 <INTEGER>) (Value 2<INTEGER>)
Description: Will subtract Value2 from Value1
Value: Value1 - Value2

Command: * (Value1 <INTEGER>) (Value 2<INTEGER>)
Description: Will multiply Value1 to Value 2
Value: Value1 * Value2

Command: / (Value1 <INTEGER>) (Value 2<INTEGER>)
Description: Will compute Value1 / Value2
Value: Value1 / Value2

Command: == (Value1 <PROGRAM>) (Value2 <PROGRAM>)
Description: Will evaluate Value1 and Value2 and compare these. If they are not comparable (e.g. a Value1 is a string, Value2 is an integer), an error is returned.
Value: 1 if value of Value1 is equal to the value of Value2; 0 otherwise

Command: < (Value1 <PROGRAM>) (Value2 <PROGRAM>)
Description: Will evaluate Value1 and Value2 and compare these. If they are not comparable (e.g. a Value1 is a string, Value2 is an integer), an error is returned.
Value: 1 if value of Value1 < value of Value2; 0 otherwise

Command: > (Value1 <PROGRAM>) (Value2 <PROGRAM>)
Description: Will evaluate Value1 and Value2 and compare these. If they are not comparable (e.g. a Value1 is a string, Value2 is an integer), an error is returned.
Value: 1 if value of Value1 > value of Value2; 0 otherwise

Command: and (Arg1 <INTEGER>) (Arg2 <INTEGER>) … (Arg_n <INTEGER>)
Description: Will compute Arg1 & Arg2 & … & Arg_n (integer "and" operation)
Value: the integer Arg1 & Arg2 & … & Arg_n

Command: begin (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)
Description: Will evaluate each argument in sequential order
Value: The value of ARG_n (the last run command)

Command: case (Test <PROGRAM>) (Case1 <PROGRAM>) (Prog1 <PROGRAM>) … (Case_n <PROGRAM>) (Prog_n <PROGRAM>)
Description: Will evaluate Test and compare it to the value of each "Case" in order; as soon as a match is found, the corresponding Prog is evaluated. If the value of Case_n is the string "default", then Prog_n is always evaluated if none of the other Cases match.
Value: The value of Prog_i if Case_i matches Test; NULL otherwise

Command: cond (Test1 <INTEGER>) (Prog1 <PROGRAM>) … (Test_n <INTEGER>) (Prog_n <PROGRAM>)
Description: Will evaluate each "Test" in order; if the value of one of the "Test" is not "0" then the corresponding program is evaluated (all other Tests and Progs are ignored).
Value: The value of Proc_i where Test_i is not 0; NULL otherwise

Command: double (Value <STRING>)
Description: Constructs a new double from the given string argument. Note "Value" is not evaluated, it is taken directly as it is.
Value: A double equal to the given string (0 if the string is ill-formed)

Command: file (Filename <STRING>)
Description: Will load in the program located at Filename and evaluate it
Value: value of the program located at Filename

Command: get (name <STRING>)
Description: Will return the value of the variable "name"
Value: the value of the variable "name"

Command: global (ARG <PROGRAM>)
Description: Will evaluate the program "ARG" in global mode
Value: value of ARG

Command: if (Test <INTEGER>) (True <PROGRAM>) (False <PROGRAM>)
Description: Will first evaluate "Test"; if this is not "0", will evaluate and return the value of "True"; otherwise will evaluate and return the value of "False"
Value: The value of "True" or "False" depending on the value of "Test"

Command: index (Arg1 <STRING or INTEGER>) (Arg2 <STRING or INTEGER>) … (Arg_n <STRING or INTEGER>)
Description: Will evaluate each Arg in order and return the string equal to "Arg1.Arg2. … .Arg_n" (without quotes). Used for array indexing.
Value: The string "Arg1.Arg2. … .Arg_n" (without quotes).

Command: integer (Value <STRING>)
Description: Constructs a new integer from the given string argument. Note "Value" is not evaluated, it is taken directly as it is.
Value: An integer equal to the given string (0 if the string is ill-formed)

Command: load_module (Filename <STRING>)
Description: Will load the command library located at Filename
Value: NULL

Command: new (type <STRING>) (name <STRING>)
Description: creates a new variable of the specified type and name.
Value: NULL

Command: new (type="array" <STRING>) (name <STRING>) (element_type <STRING>) (size <INTEGER>)

Description: creates a new array of the specified name and size. Each entry is of type "element_type". Use the "index" command to index in the array.
Value: NULL

Command: NOP (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)
Description: Will ignore each ARG (they are not evaluated) and do nothing.
Value: NULL

Command: nop (ARG1 <PROGRAM>) (ARG2 <PROGRAM>) … (ARG_n <PROGRAM>)
Description: Same as NOP just lower-case depending on user preferences
Value: NULL

Command: not (Arg <INTEGER>)
Description: Will "not" the value of Arg
Value: 1 if value of Arg is 0; 0 otherwise

Command: or (Arg1 <INTEGER>) (Arg2 <INTEGER>) … (Arg_n <INTEGER>)
Description: Will compute Arg1 | Arg2 | … | Arg_n (integer "or" operation)
Value: the integer Arg1| Arg2 | … | Arg_n

Command: path (Path <STRING>)
Description: Will add "Path" to the list of directories to search for files and modules. Use the shell command "!path" to see the current path list.
Value: NULL

Command: print (Value <PROGRAM>)
Description: Will print to the console screen, the value of "Value"
Value: NULL

Command: set (name <STRING>) (Value <PROGRAM>)
Description: Will set the value of the variable "name" to the value of "Value". Returns an error if types don't match.
Value: NULL

Command: string (Value <STRING>)
Description: Constructs a new string from the given string argument. Note "Value" is not evaluated, it is taken directly as it is.
Value: A string equal to the given string.

Command: user.input (Type <STRING>)
Description: Will request input from the user. Type may be "integer" (an integer is returned), "string" (a string is returned), "char" (Windows only, an integer is returned representing the ASCII key pressed), or "kbhit" (Windows only, non-blocking; returns 0 if no key is pressed, otherwise the ASCII code of the pressed key)
Value: Depends on Type and the user input.

Command: while (Test <STRING>) (Prog1 <PROGRAM>) (Prog2 <PROGRAM>) … (Prog_n <PROGRAM>)

Description: Will evaluate Test; if this is 0, the loop terminates. Otherwise each Prog is evaluated in order. If one of the Prog evaluates to the string “break”, the loop terminates, if one of the Prog evaluates to the string “continue” the while loop starts over (evaluating “Test” and running Prog1, etc.). After Prog_n has evaluated, the loop repeats by re-evaluating “Test”.
Value: NULL

Extending the Language

One may write new commands and data types to extend the functionality of wok::Interpreter. There are two ways to do this: first you may statically link the interpreter to your project and create a command library this way. Secondly you may write a dynamic library containing your language extension. Then in any instance of wok::Interpreter (even ones that have been statically linked to other libraries), you may use the “load_module” command to access your library.

Despite the method you choose, you will need to create your commands. This is done in C++ by inheriting the “CommandLibrary” class and overwriting the virtual functions: evaluate, newUserType, and requiredVersion. See TestModule for an example.

To incorporate your new CommandLibrary (call your class userCL which inherits CommandLibrary), use the following code:

```
wok::interp::Interpreter
std::vector <wok::interp::CommandLibrary*> library;
library.push_back(new userCL);
interpreter.shell(“Your CL Extension”, library);
```

To create a DLL file, the same exact CommandLibrary class is used. The only difference is that you must add the following code (assuming Windows):

```
extern "C"
__declspec (dllexport) wok::interp::CommandLibrary* __cdecl LoadCommandLibrary()
{
	return new userCL;
}
```