Interp Internals

Version 01.01.00

June 12, 2008

1. Overview.

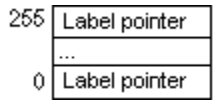
Interp isn't your ordinary byte-code interpreter. The byte-codes (also called tokens and operators) are ASCII characters. The c-string (zero-terminated) is the unit of interpretation. Execution of the string contents proceeds from left (low memory) to right (high memory.) There isn't any concept of lines, or whitespace to the inner interpreter (interp.c.) The virtual machine has a stack-based, operator-follows-operand (RPN) architecture. Numeric literals are ASCII, not binary. When you see whitespace, think NO-OP. Typically they (NO-OPs) are used to terminate numeric literals, but are also frequently used to keep everything from running together.

1.1. Operands and Operators.

As numeric literals are encountered, they are converted to unsigned 32-bit values and pushed on the data stack. (The length of the number is determined by deduction. If the next character is not a valid digit in the current number base, that's the end of it.) When operators are executed, they take their arguments from the data stack, perform their computation or transformation and push their result (if there is one) on the data stack.

1.2. Vector Table.

During execution of an individual token, its ASCII value (0-255) is used as an index into an array of addresses (the vector table.) Each table entry is the starting address of the code that implements that token's behavior (functionality.) Sound familiar? It's indirect-threaded code.



1.3. Sub-Operators.

Sometimes, the operator is followed by one or more characters that qualify it. For example, 'S' is the stack operator. It is qualified by one of these: 'C', 'D', 'c', 'd', 'n', 'o', 'p', 'r', or 's'. That second letter is often called a sub-operator. For example, “Ss” is STACK-SWAP.

1.4. Input Concatenation.

On top of this very primitive machine sits the outer interpreter (imain.c.) It reads lines of input from STDIN and passes each line, as a c-string, to the interp() function. The outer interpreter limits line length to 80 characters. As previously mentioned, the inner interpreter (interp.c) executes strings of characters. This presents an interface difficulty: lines of related input must be concatenated into a single, 64Kbyte, string (by imain(), in imain.c) before execution can commence (via interp(), in interp.c.) This concatenation action is user-controlled through the outer interpreter directives “#BUFFER” (begin concatenation with the next input line) and “#EXECUTE” (stop concatenation,

with the previous input line, and give the completed string to interp().) This solution also completely eliminates a related problem: logical structures (functions, if/then/else and loop/while) cannot span strings. A side effect of buffering is prettiness. You can write indented code (that looks vaguely like c.)

1.5. Macro Expansion.

In addition to input buffering, the outer interpreter performs one more function that is actually its most important action – input transformation. As each line is read, outer interpreter directives are processed, in-line comments are removed, and the rest of the line is passed through a substitution-based macro processor. After all substitutions are made, the fully processed line is executed or buffered.

1.6. Design Goals.

The focus of this design is avoidance: 1) no formal syntax (grammar), and 2) no parser. The macro processor is the center piece. It gives back important features that were lost by eliminating the parser. You can have a more traditional looking language with “if”, “else”, “loop”, and so on. Plus you get something you can't have with a parser – customization. You can do whatever you want with the standard macros – discard them, rename them, or add more, and without editing source code.

2. Inner Interpreter.

Interp() resides in interp.c. It takes two arguments, a pointer (*cp*) to the control block, and a pointer (*str*) to the string of tokens to be interpreted. The inner interpreter is written to support multiple interpreter instances within a single execution unit. This is why *cp* is passed in. The caller (e.g. main()) changes instances just by changing the control block pointer. The control block contains all the information that is specific to a single interpreter instance: data stack (and other stacks), output number base, global and local function definitions, global and local variables, system constants, vector table, I/O buffers, and error information.

Data Stack pointer (sp)	0
Loop Control Stack pointer (lcsp)	1
Function Return Stack pointer (frsp)	2
Output Number Base (obase)	3
Global Function Table pointer (gfp)	4
Local Function Table pointer (lfp)	5
Global Variable Pool pointer (vp)	6
Local Variabler Pool pointer (lp)	7
System Constants Table Pointer (kp)	8
Command Vector Table pointer (vt)	9
Input Message Buffer pointer (ibuf)	10
Output message Buffer pointer (ebuf)	11
Bad Opcode Pointer (bop)	12
Error Code (err)	13

All this instance specific data is created by calling interp_new(), which returns a pointer to the newly created, and freshly initialized control block. Initialization of the vector table is deferred until the first time interp() is called with the new control block pointer.

2.1. Stacks in Interp.

All stacks behave the same, without regard to their use or contents (much as hardware-supported stacks in most computers.) Interp uses three stacks: 1) the data stack, 2) the loop control stack, and 3) the function return stack. Each data stack item exactly fills one stack position. Loop control stack items also occupy one stack position. Function return entries (or frames) occupy three stack positions. These three items capture the execution environment at the time the function call was made, and when restored, allow interp to continue on after the function finishes execution (and returns.) From the point of view of the inner interpreter, functions are simply independent strings of opcodes.

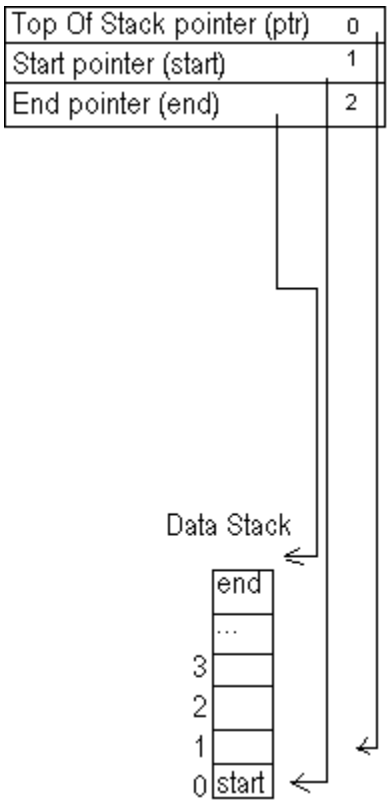
The information that must be stored on the function return stack (*cp->frsp*) and later retrieved from it (in the order it is pushed on) is: 1) the address of the name of the function being called, 2) the address of the start of the local variables, and 3) the address of the local function table. Item 1 is just the return address. Items 2 and 3 are the execution context of the calling (parent) function.

2.2. Stack Operations.

Stack operations are handled by two functions (Stack_new, and Stack_free) and a comprehensive set of sixteen macros (defined in istack.h.) Because there is more than one stack, all the macros need at least one argument: a pointer to the stack of interest. The most commonly used stack macros are Stack_depth_M and Stack_empty_M. The former produces an integer number and the latter a 0 (FALSE) for not-empty or a non-zero (TRUE) if that stack is empty. The names of the stack macros were chosen because they communicate the intent. "_push", "_pop", and "_drop", write (add to), read (take from), or discard the top of the stack. "_get" and "_put", overwrite or make a copy of the top item on the stack without disturbing the stack contents. "_under", references the item that is underneath the top item on the stack without disturbing the stack contents. "_under_under" references the item that is two positions down from the top item on the stack without disturbing the stack contents.

2.3. Stack Architecture.

Stacks consist of a list head and a one-dimensional array of 32-bit integers. The list head contains three elements: 1) the address of the item currently on the top of the stack (commonly called "the stack pointer"), 2) the address of the item at the beginning of the stack (usually called "the base address", and 3) the address of the item at the end of the stack (called "the end of the stack.")



Because the stack grows toward higher memory addresses, the base address is always less than the end of the stack. When the stack is empty, the stack pointer is less than the base address. When the stack is full, the stack pointer is greater than or equal to the end of the stack. A stack push increments the stack pointer and stores the item. A stack pop reads the item and decrements the stack pointer.

2.4. Opcode Implementation.

When control passes to interp(), four local variables are created: 1) *opcode* points to the first character in the input string (takes on the value of *str*), 2) *vtptr* points directly to the base address of the current instance's vector table (takes on the value of *cp->vt*), 3) *datap* points directly to the data stack list head (takes on the value of *cp->sp*), and 4) *funcp* points directly to the function return stack list head (takes on the value of *cp->frsp*.)

Features common to all opcodes (implementation features):

1. They start with a unique label followed by a new context (left brace.)
2. If local variables are needed they are defined next.
3. The opcode-specific instructions are next.
    a. When errors are detected, an error message is written into *cp->ebuf* (the error and output buffer), and a non-zero error code is issued via the “return” statement (to interp's caller.)
    b. At the end of the opcode-specific instructions, *opcode* points to the last character in the input string that was processed as a result of executing the opcode.
4. Then the right brace signals the end of the context (local variables are discarded.)
5. Then the NEXT_M macro advances to the next token.

2.5. START_M and NEXT_M.

Recall that one of the arguments passed to interp(), *str*, points to the first (left-most) character in the string of tokens that are to be interpreted (executed.) START_M is a computed got. It uses the current value of *opcode* to index the vector table and retrieve the start address of the opcode's instruction sequence, then it jumps to it. This is often called “token execution.” START_M is used to execute the first token from the input string, or the first token from a function. NEXT_M, used everywhere else, assumes that *opcode* points to the last character processed and increments *opcode* before using it to index the vector table, retrieve the start address for the opcode's instruction sequence, and then jump to it.

2.6. End-of-string (EOS) Processing.

The zero ('\0') character at the end of the interp input string is just another token. This saves time by eliminating a special case for START_M and NEXT_M to be aware of.

2.7. Opcode Implementation Families.

There are specific implementation patterns (templates) recommended for each category or family of opcodes. If you need to create a new opcode, please copy an existing one that has the same stack usage as your new opcode will. This will accelerate your development in two ways: you are modifying existing code (that works), and avoiding the pitfalls that come with the large number of stack operators, whose correct usage isn't always obvious. Here are the most obvious categories:

1. Transformation – stack values change, but the stack depth does not change:
   '@', 'K', 'L', 'M?', 'Ma', 'Ml', 'N', 'Sp', 'Sr', 'Ss', 'V', 'a', 'b@', 'bL', 'bV', 'h@', 'hL', 'hV', '~'
2. 2-to-1 – two stack operands are transformed into one result:
   '%', '&', '*', '+', '-', '/', '<', '<<', '<=', '<>', '=', '>', '>=', '>>', '?s', 'Mm', '^', '|'
3. Creation – no stack operands are used, but a new value is pushed onto the stack:
   ',', ':', '?', 'Il', 'Ir', 'Sn', 'Te', 'l', 'qg', 'ql'
4. Consumption – one or more operands are consumed without producing a result for the stack:
   '!', '.', 'M”', 'Mf', 'SD', 'Sd', '[', '\', 'b!', 'h!', 'M@', 'p', 'w'
5. Operator – The token executes one of a class of related operations, based on the value of the next character (sub-operator):
   '<', '>', '?', 'I', 'M', 'P', 'S', 'T', 'X', 'b', 'd', 'h', 'm', 'q'
6. Other – the operators that don't fit any of the preceding categories:
   '$', '”', '(', ')', ';', 'F', 'MC', 'MR', 'MW', 'Mc', 'Ms', 'Pd', 'Po', 'Px', 'SC', 'Sc', 'So', 'Ts', 'Xd', 'Xg', ']','`', 'bC', 'bR', 'bW', 'dG', “dL', 'dg', 'dl', 'f', 'hC', 'hR', 'hW', 'mF', 'mf', 'v', '{', '}'

2.7.1. Transformation.

Opcodes that process the item on the top of the stack into some type of result may take advantage of Stack_get_M and Stack_put_M and eliminate two unnecessary stack adjustments (one pop followed by one push.) In some cases, like 'Sp' (STACK-PICK), a stack macro (Stack_pick_M) is created for a single use. This also happened with 'Sr' (STACK-ROT), only this time two unique operations were required: Stack_put_under_under_M, and Stack_get_under_under_M.

2.7.2. 2-to-1.

This category includes the comparison, bit-shift, and logical operators, and nearly all the math operators. As a family, the most interesting characteristic of these operators is the fact that each opcode

actually has a unique, although usually simple algorithm as it core. An exception is the I/O opcode '?s' (READ-STRING.)

2.7.3. Creation.
Some of this category's members are I/O opcodes, like ':' (READ-CHAR), '?' (READ-NUM), and ',' (PRINT-CHAR.) Others return information about something that cannot be expressed by a stack argument, like the stack or functions. This includes 'Sn' (STACK-NUM), 'qg' (QUERY-GLOBAL), and 'ql' (QUERY-LOCAL.)

2.7.4. Consumption.
This category's members are split between storage, (destructive) stack, and decision (branching) opcodes. Generally, every opcode in this category performs an action: writes data to memory, waits for time to pass, branches conditionally (flag value on stack), or takes data off the stack. If you are creating an opcode that exhibits this type of behavior, use one from this group as the basis or model for your work.

2.7.5. Operator.
This category extends the interpretation of “token” beyond one character. When a operator token is executed, it evaluates the next character, as a sub-operator, and branches to the proper opcode implementation. “Interp_stack_operator:” is a prime example of an operator opcode that absolutely requires a valid sub-operator. A second example, “Interp_lt_operator:”, can properly recognize any of these cases: '<<', '<=', '<>', or '<'. It's that last case, when no valid sub-operator is found, that requires this operator token to have a “default” action.

2.8. The zero opcode ('\0').
It is at the end of every string of opcodes, including functions. Which means that it serves two purposes. It terminates the input string passed to interp, and at the end of a function it causes the function to stop executing and return to the caller (where execution will resume.) In interp.c, find the label “Stop_Interpreting:” and follow along in the code. If the function return stack (*funcp*) is empty, we can safely assume we have reached the end of the string that was passed to interp, and return success to whomever called interp(). Otherwise, we are attempting to return from one of the three types of functions interp supports: 1) global functions, 2) local functions, and 3) macros. As mentioned earlier, when a function is called, three items are pushed (in order) on the function return stack: 1) the return address (a copy of *opcode*), 2) the local variables pointer (a copy of *cp->lp*), and 3) the local function table pointer (a copy of *cp->lfp*.) In “Stop_Interpreting:” they are popped off the function stack in reverse order. If they are non-NULL, then storage must be released and they are written into their former position in the control block. If they were NULL, there's no storage to release and nothing to restore (that pointer wasn't changed for the function call.)

Global functions have their own local variables and their own local functions (items 2 and three are non-NULL.) Local functions have their own local variables but share the same local function table as their parent (item 2 is non-NULL, but item 3 is NULL.) Macros use their caller's environment (items 2 and 3 are NULL.)

2.9. Return Caller Context.
The L-BASE (“l”) opcode returns the base address of the caller's local variable table. This might be useful when the parent function is sharing data from it's local variable table with a local function it

calls. Once your local function has that address, it can read or write that data. Of course there are other ways to achieve the same affect such as pass the address on the stack, or pass an offset (from the base address of the parent's local variable table.) Find the label “Interp_return_caller_context:” in interp.c, and follow along. Once it has been determined that at least one frame is on the function return stack, the local variable, which is the second item in the frame, is copied and pushed onto the data stack. If this function has a zero value for the local variable pointer, then it was called as a macro. In this case, instead of failing, the proper value is copied from system constant 15. The point of doing this additional check is to make calling methods as transparent as possible. In other words, whether the function using this opcode is a global function, a local function or is called as a macro, it just works. This is an important tip to make use of when creating your own opcodes.

2.10. If.

Find the “Interp_if:” label in interp.c and re-read the description of this opcode in the comments: “[(*flag* --) IF: if *flag* is TRUE execute the commands following the [up to the matching ; or]. If *flag* is FALSE, skip to the matching ; or] and begin executing at that point.” This tells you exactly how the implementation code works, but it isn't clear enough. Half of the secret is in this phrase: “execute the commands following the [up to the matching ; or].” It is telling you when *flag* is TRUE, this opcode is effectively a no-op. Look at the code. After the top of stack value (*flag*) is popped into the variable *number*, there's an if statement that prevents the entire body of this opcode from executing when *number* (the variable) is not zero (TRUE.) So it returns without doing anything other than consuming *flag*, and the opcodes that follow the '[' are executed.

The other half of the secret is in this phrase: “skip to the matching ; or] and begin executing at that point.” This is a full description of what is happening inside the if statement described in the previous paragraph (when *flag* is FALSE.) All the difficulties in this implementation come from trying to “skip” opcodes (including QUOTE and nested if statements), while looking for ';' or ']' in a syntax-less language. The bug warnings are there because this code can be easily broken by valid opcode sequences that cannot be detected without simultaneously killing execution speed and giving the code knowledge of the opcode values. Documenting the bugs was the most execution-efficient solution.

It may not be totally clear, but when the “skipping” is over, *opcode* is pointing at the token that was found (';' or ']'.) This is significant. When NEXT_M executes (at the very end of this opcode implementation), the token that was found is skipped without ever having a chance to execute. It has to be this way because ';' is just a branch opcode (see the following paragraph for details), and ']' isn't even an opcode and cannot execute.

2.11. Else.

As stated in the previous section, else is a branch statement. When *flag* is TRUE, and the body of the if statement executes, the else token is that last opcode within the body of the if that gets to execute. It begins searching for the matching ']' while dealing with quoted strings (QUOTE) and nested if statements. It suffers from the same class of bugs that afflict the if opcode.

2.12. Loop.

As with the if statement, the description of loop tells the short story of its execution: “((--) LOOP: marks the beginning of a loop see '\' (WHILE) and ')' (END-LOOP.)” The value of *opcode* (the address of the '(' in the string being executed) is pushed onto the loop control stack (*cp->lcsp*.)

2.13. While.

There is no set rule (no syntax), but the while ('\') is usually placed shortly after the beginning of the loop, marked by '('. The code just before the while computes the value of *flag*. If *flag* is TRUE, while behaves like a NO-OP and returns without doing anything except consuming *flag*. On the other hand, when flag is TRUE, this opcode skips to the matching ')' and execution resumes with whatever follows it (with thanks to NEXT_M.) As with other opcodes, the difficulties come from trying to "skip" opcodes (including QUOTE and nested loops), while looking for ')' in a syntax-less language.

2.14. LOOP-END.

The end of the loop opcode ')', pops the loop control stack and assigns that address as the value for *opcode*. This will be the address of the beginning of the loop, as indicated by the matching '('. That's all there is to it. As with other opcodes, NEXT_M completes the puzzle by incrementing *opcode*. Then the next token executes.

2.15. Define Function (Global and Local).

The biggest obstacle to overcome with evaluating function definitions is finding the end of the function in a syntax-less language. As with if/then/else and loop/while/end, the process, while fast, is not perfect. But the problem is simpler for this case – only quoted strings (QUOTE) must be dealt with. Once the length of the definition is known, the opcode ('{' or "`"), the function name (character following the opcode), and the terminal character ('}' or "`") are omitted from the length, and memory is allocated to hold the function body. Then the body is copied into the allocated memory. If there is already a definition in the global or local function table with the same name, that definition is freed. Then the memory pointer for the new definition is stored in the global or local function table (at the offset determined by the ASCII value of the function name.)

2.16. Execute Function (Global and Local).

Function execution is relatively straight-forward. The opcode tells you if this is a global function ("F"), or a local function ("f".) Then the entry in the appropriate function table is checked for validity (non-NULL.) If all is well, the stack frame is created on the function return stack by pushing three values in this order: 1) the value of *opcode* (it points to the character following the opcode – the name), 2) the base address of the local variable table (*cp->lp*), and 3) the base address of the local function table (*cp->lfp)*, if it is a global function, or zero if it is a local function. Then create the execution context for this function: allocate the local variable table, and the local function table (but only for global functions.) Then transfer the value from the appropriate function table entry (by function name) into opcode and use START_M to initiate function execution.

The only tricky aspect of this code comes from the fact that memory has to be allocated for the execution context. If the allocations fail, the execution context must be restored from the stack frame. Once the machine state is restored, the error can be reported (and optional information displayed.)

2.17. Execute Macro (Global and Local).

Do not be confused. This is not a third type of function (in addition to global and local functions.) It's just another, more economical way to call an existing function. In other words, you can call a global function with 'F' or 'mF', and you can call a local function with 'f' or 'mf.' The advantage to calling a function as a macro is being able to avoid creating the function's executing context. In other words, whether you call a global function or a local function as a macro, when called this way, the function uses the parent's (caller's) local variables, and local function table. As long as private data, private local functions, or access to the parent's context are not required by the function, there will be no difference in its execution behaviour.

The only difference between the global macro ('mF') operator and the local macro ('mf') operator is which function table is the function address drawn from. The entry in the appropriate function table is checked for validity (non-NULL.) If all is well, the stack frame is created on the function return stack by pushing three values in this order: 1) the value of *opcode* (it points to the character following the opcode – the name), 2) a zero for the base address of the local variable table, and 3) a zero for the base address of the local function table. Then transfer the value from the appropriate function table entry (by function name) into opcode and use START_M to initiate function execution.

2.18. Push Number (numeric literals).

At first glance, the opcode implementation for numeric literals ("Interp_push_number:") seems too trivial to talk about – convert the (c-style) unsigned digit string into a binary number and push it on the stack. The interesting part of this opcode, isn't here. It is the recognition of numeric literals. Unlike all the other opcodes, numbers do not all look the same: "23", "0777", "0", "0xd00d", and "0xACE" are all valid numbers, and none of them look alike. Where is the recognizer for numeric literals? There isn't one. Instead, there are 10 vector table entries dedicated to processing numeric literals: entries '0' thru '9'. All ten entries point to this opcode implemetation. The string of digits is viewed as an opcode ('0' thru '9') followed by zero or more additional digits (in the specified number base.) This design decision makes START_M and NEXT_M function as the recognizer for numeric literals. The processing overhead is the conversion from a digit string to a binary value, which is done every time a number is executed, even in loops.

2.19. Exit.

A brief study of "Interp_exit:" shows how much interp's concept of the execution context relies on memory allocation. When '$' is executed, the execution context, which contains this token, is destroyed. This opcode is not permitted in a function or macro context. The first structure released is the data stack. Then the global function table is searched for non-NULL entries. Each one is released. Once all global functions are gone, the global function table is released. The same two-step process is done for the local function table. Then the loop control and function return stacks are released. Then the global variable data area and the local variable data area are released. Then the system constants data area, command vector table, input message buffer, and output/error message buffer are all released. If the user has allocated memory and forgotten to free it before executing this opcode, it is not freed, but its location may have been lost when the global and local variables were released.