Navarint: A Simple Interpreter

This simple interpreter is written in C and uses a recursive-decent parser. This documentaion contains some of the internals of the language. Please note that everything that is documented here is not required for solving this challenge.

Please note that the interpreter is not fully complete and you might observe that some functionalities are not working, and I apologize for that. But this still can be pwned !

Credits

The initial idea for this interpreter and also many parts of the code come from this awesome tutorial on writting interpreters - https://github.com/lotabout/write-a-C-interpreter/blob/master/tutorial/en/0-Preface.md

Syntax

The grammer for this language can be found in the grammer.md file. Here are some of the essentials anyway :)

- **Math Expressions**:

```
In [1]: 1+2
Out[1]: 3
In [2]: 5*4-2/1*4
Out[2]: 12
```

- **Logical Operations**:

```
In [1]: 1 == 1
Out[1]: 1
In [2]: 100 > 1000
Out[2]: 0
```

- **Variable Assignment**:

```
In [1]: a1 = 0x10
In [2]: a2 = 1
In [3]: a1 + a2
Out[3]: 17
In [4]: str = "this is a string"
```

```
In [5]: str
Out[5]: "this is a string"
In [6]: list = [1, 2+2, 0x100, "list with string", [12, str]]
In [7]: list
Out[7]: [1, 4, 256, "list with string", [12, "this is a string"]]
```

- **Loop**:

```
In [1]: i=0
In [2]: sum=0
In [3]: while (i<10){sum=sum+i; i=i+1}
In [4]: sum
Out[4]: 45
```

- **Conditional Statement**:

```
In [1]: a=1234
In [2]: b=5678
In [3]: c=0
In [4]: if(a==b){c=1} else{c=100}
In [5]: c
Out[5]: 100
```

- **Function**: Not working as expected :(

File Hierachy

- **global.h**: Contains global variables, tokens, vm instructions and all the structures used.
- **tokens.h**: The tonkenizer
- **parser.h**: Holds the code for the parser
- **vm.h**: Just a wrapper for calling vm_operations which are defined in `vmop.h`
- **vmop.h**: Contains the code for all the VM operations.
- **interp.c**: The entry point of the interpreter.
- **grammer.md**: The grammer for navarint

Internals

Essentially, Navarint parses the input code into bytecode which is then executed by it's VM.

Type Representation

In Navarint, there is only one datatype - `val`. It has the following structure:

```
typedef struct val
{
  long type;
  long data;
}val;
```

`type` is the underlying primitive type of this val. `type` can take the following values -

- 0 - None Type - Default type
- 1 - Integer type
- 2 - Strings type
- 3 - List type
- 4 - Pointer type
- 5 - Function type

  The data pointer of the `val` structure points to the actual data of this val. If the type is Int, then the data field contains the integer itself. In all other cases, it points to the objects of their corresponding types. For more details, feel free to go through globals.h.

Stack

There are essentially 2 stacks - the VM's operation stack and the call stack.

The operations stack is used for all VM operations and operands are passed to operations via this stack. Each entry in the operations stack has the datatype `val`

The call stack is used to save the `pc` in case of a function call. It also keeps track of the current scope.

Registers

- sp: operations stack top pointer
- ax: auxilary register. has the type `val`
- cs_sp: call stack top pointer.
- cs_bp: call stack base pointer.

VM Opcodes

A brief description for each of the bytecodes.

- **LEA**: Not implemented
- **IMM**: Load the immediate value that follows into the `ax` register
- **JMP**: Jump (set the pc) to the address pointed to by pc
- **CALL**:
- **JZ**: Jump if `ax` =0
- **JNZ**: Jump if `ax` != 0
- **RET**: Return to the caller. Pop old scope and pc from call stack
- **SET**: Set `ax` to the content of the varible on stack top.
- **GET**: Put the content of a variable pointed to by `ax` into `ax` (dereference `ax`)
- **PUSH**: Push the contents of `ax` on to the stack
- **GETELEM**: Get the nth item from the string/list that is there on top of the stack. `n`, the index, is taken from `ax`
- **SETELEM**: Set the nth item of a string/list to value. `n` the index, is present on top of the stack. Second on stack top comes the list/string. The `value` is taken from `ax`
- **ADDELEM**: Add the element in `ax` to the list present on top of the stack.
- **EXIT**: Exit the VM
- **Math Operations**: All these take the first operand from stack top and second from `ax`. The result of the operation is also moved into `ax`: **OR**, **XOR**, **AND**, **EQ**, **NE**, **LT**, **GT**, **LE**, **GE**, **SHL**, **SHR**, **ADD**, **SUB**, **MUL**, **DIV**, **MOD**
- **Not Implemented**: **OPEN**, **READ**, **CLOSE**, **PRTF**, **MALC**, **MSET**, **MCMP**, **LI**, **LC**, **SI**, **SC**, **ADJ**, **LEV**, **ENT**

Tips

Best way - try random input and analyse a crash, instead of going through the code . If your random input does not crash the interpreter, I wonder if you can use the most basic of the basic and the most dumbest of the dumb fuzzers....