VM Actions

This document describes the Virtual Machine (VM) actions for the code that is generated by the parser. The companion document, parserActions.pdf, describes the actions the parser performs to generate this code.

Data structures used by the VM to execute the program.

Instruction memory: This contains the instructions of the program. It should be a vector or array of smart pointers to *Stmt* objects, with the object being pointed to being a concrete object that extends Stmt. By pointing to objects, each statement takes the same amount of space (1) in the instruction memory, and simplifies your programming effort.

pc: the location of the next instruction in the instruction memory to be executed.

Runtime stack: the runtime stack holds the values operated on by the stack oriented instruction (various flavors of pop, push, add, div, etc.).

Data memory: the data memory contains storage for variables. As a subroutine is entered, the data memory expands in size to hold those values by the amount specified by the *gosublabel* operand. As a subroutine is exited, the data memory shrinks to the previous size, popping those variables off the stack. This is similar to the stack used by C and C++ to hold data for subroutines.

String table: This holds the statements used by prints statements.

The operation of instructions in the .out file

Unless otherwise specified, all instructions increment the pc by one.

declscal *var* : no code was generated and is irrelevant at runtime

declarr *var length* : no code was generated and is irrelevant at runtime

label *label* : no code was generated and is irrelevant at runtime

OP_ENTER_SUBROUTINE *opnd* : Add a stack frame of size *opnd* to the data memory to hold the variables used by the subroutine. Increment the pc to cause the next statement in instruction memory to be executed next.

This instruction was created by the parser from the *gosublabel label* statement.

OP_START_PROGRAM *opnd***:** Set up storage in the data memory to hold the outer scope variables. The size of the stack frame is given by *opnd*. This is basically setting up the stack frame for the main procedure.

This instruction was created by the parser from the ***start*** statement.

end: no code was generated and this statement is irrelevant at runtime

OP_EXIT : exit the program, i.e., terminates execution.

This instruction was created by the parser from the ***exit*** statement.

OP_JUMP *opnd*: Set the pc to the address specified by *opnd.* This instruction was created by the parser from the ***jump label*** statement.

OP_JUMPZERO *opnd*: Remove the top element from the runtime stack. If it is zero, set the pc to the address specified by *opnd*, otherwise increment the pc to execute the next statement.

This instruction was created by the parser from the ***jumpzero label*** statement.

OP_JUMPNZERO *opnd* : Remove the top element from the runtime stack. If it is non-zero, set the pc to the address specified by *opnd*, otherwise increment the pc to execute the next statement.

This instruction was created by the parser from the ***jumpnzero label*** statement.

OP_GOSUB *opnd* : Make a subroutine call. The address of the next statement (pc+1) is stored into a runtime stack of return addresses. The pc is set to the value of *opnd*, which is the address of the subroutine.

This instruction was created by the parser from the ***gosub label*** statement.

OP_RETURN : return from a subroutine call. Set the pc to the last return address saved by an OP_GOSUB instruction. Pop the current subroutine stack frame from the data memory.

This instruction was created by the parser from the ***return*** statement.

OP_PUSHSCALAR *opnd*: The value at the location in data memory given by *opnd* is pushed onto the runtime stack. This should be the location of a scalar variable.

This instruction was created by the parser from the ***pushscal var*** statement.

OP_PUSHARRAY *opnd*: The value of *opnd* is added to the value at the top of the runtime stack, giving a location *e* in the data memory of an array element (*opnd* is the start of the array and the value at the top of the runtime stack is the index into the array). The element at the top of the runtime stack is discarded, and the value of the data memory at location *e* in the data memory is pushed onto the runtime stack.

This instruction was created by the parser from the ***pusharr var*** statement.

OP_PUSHI *opnd* : The value of *opnd* is pushed onto the runtime stack.

This instruction was created by the parser from the ***pushi*** statement.

OP_POP: The value at the top of the runtime stack is discarded.

This instruction was created by the parser from the ***pop*** statement.

OP_POP_SCALAR *opnd* : The value at the top of the runtime stack is removed and stored into the data memory at location *opnd*.

This instruction was created by the parser from the ***popscal var*** statement.

OP_POP_ARRAY *opnd*: The value at the top of the runtime stack and the value of *opnd* are added to give a value *e*. The value at the top of the runtime stack is then discarded. The value now at the top of the runtime stack is removed and placed into the data memory at location *e*.

This instruction was created by the parser from the ***poparr var*** statement.

OP_DUP : The value at the top of the runtime stack is duplicated. Thus, if 3 4 were the top two elements on the runtime stack before executing an OP_DUP, with 4 being at the top of the runtime stack, after executing the OP_DUP the runtime stack will hold 3 4 4.

This instruction was created by the parser from the ***dup*** statement.

OP_SWAP : The top two values value in the runtime stack are swapped. Thus, if 3 4 were the top two elements on the runtime stack before executing an OP_SWAP, after executing the OP_SWAP the runtime stack will hold 4 3.

This instruction was created by the parser from the ***swap*** statement.

OP_ADD : Let 3 4 5 be the values at the top of the runtime stack, with 5 being at the top. After the add is executed, the runtime stack will contain 3 9. Thus, the top two elements are removed, added, with the sum pushed onto the runtime stack.

This instruction was created by the parser from the ***add*** statement.

OP_NEGATE : Let 3 4 be at the top of the runtime stack, with 4 being at the top. After executing OP_NEGATE the runtime stack will be 3 -4. Thus, the value at the top of the runtime stack is negated.

This instruction was created by the parser from the ***negate*** statement.

OP_MUL: Let 3 4 5 be at the top of the runtime stack, with 5 being at the top. After executing OP_MUL the runtime stack will be 3 20. Thus the top two elements of the runtime stack are popped off and multiplied, with the result being pushed onto the stack.

This instruction was created by the parser from the ***mul*** statement.

OP_DIV: Let 3 4 8 be at the top of the runtime stack, with 8 being at the top. After executing OP_DIV the runtime stack will be 3 2. Thus the top element is popped and divided by the new top element, which is also popped. The result of the division is pushed onto the runtime stack.

This instruction was created by the parser from the ***div*** statement.

OP_PRINTS : The string at location *opnd* in the string buffer is printed to standard out.

This instruction was created by the parser from the ***prints string*** statement.

OP_PRINTTOS : The value at the top of the runtime stack is popped and printed.

This instruction was created by the parser from the ***printtos*** statement.

Actions of the VM

The file <fn>.out is opened for reading. The strings in the string buffer are read first, and added to the string buffer. Instructions are then read, and an instruction factory is used to create objects that represent the instruction, where the instruction object is derived from an abstract Stmt object.

Once the instruction buffer is created, the pc is set to the address of the first instruction in the buffer (this should be 0) and the instruction pointed to by the pc is executed until an exit instruction executes and the program is finished.