Appendix E
Definition of the CPRL Virtual Machine

E.1 Specification.

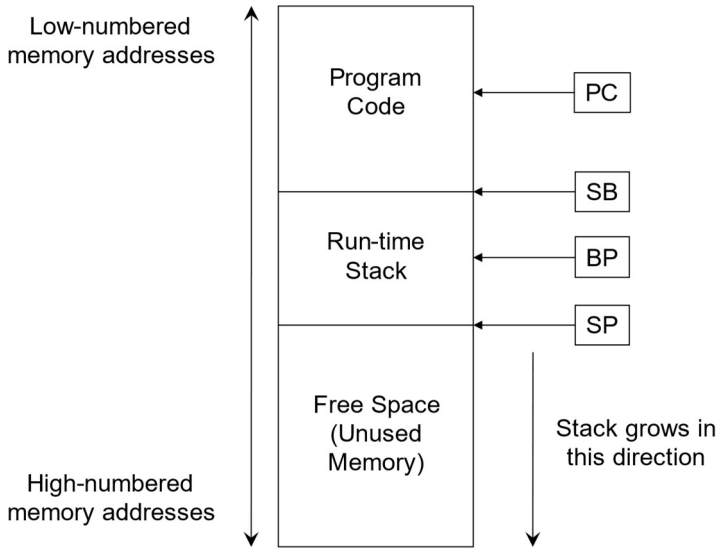
CVM (CPRL Virtual Machine) is a hypothetical computer designed to simplify the code generation phase of a compiler for CPRL (the Compiler PRoject Language). CVM has a stack architecture; i.e., most instructions either expect operands on the stack, place results on the stack, or both. Memory is organized into 8-bit bytes, and each byte is directly addressable. A word is a logical grouping of 4 consecutive bytes in memory. The address of a word is the address of its first (low) byte. `Boolean` values are represented in a single byte, character values use 2 bytes (Unicode Basic Multilingual Plane or Plane 0, code points from `U+0000` to `U+FFFF`), and integer values use a word (four bytes).

CVM has four 32-bit internal registers that are usually manipulated indirectly as a result of program execution. There are no general-purpose registers for computation. The names and functions of the internal registers are as follows.

- `PC` (program counter); a.k.a. instruction pointer: holds the address of the next instruction to be executed.
- `SP` (stack pointer): holds the address of the top of the stack. The stack grows from low-numbered memory addresses to high-numbered memory addresses. When the stack is empty, `SP` has a value of the address immediately before the first free byte in memory.
- `SB` (stack base): holds the address of the bottom of the stack. When a program is loaded, `SB` is initialized to the address of the first free byte in memory.
- `BP` (base pointer): holds the base address of the current activation record (a.k.a., frame); i.e., the base address for the subprogram currently being executed.

Each CVM instruction operation code (opcode) occupies one byte of memory. Some instructions take an immediate operand, which is always located immediately following the instruction in memory. Depending on the opcode, an immediate operand can be a single byte, two bytes (e.g., for a char), four bytes (e.g. for an integer or a memory address), or multiple bytes (e.g., for a string literal). The complete instruction set for CVM is given in the next section. Most instructions get their operands from the run-time stack. In general, the operands are removed from the stack whenever the instruction is executed, and any results are left on the top of the stack. With respect to boolean values, zero means false and any nonzero value is interpreted as true.

The following diagram illustrates a program is loaded into memory.



Variable Addressing

Each time that a subprogram is called, CVM saves the current value of `BP` and sets `BP` to point to the new activation record (a.k.a., frame). When the subprogram returns, CVM restores `BP` back to the saved value.

A variable has an absolute address in memory (the address of the first byte), but variables are more commonly addressed relative to a register. A local variable is addressed relative to register `BP`, and a global variable is addressed relative to register `SB`.

Various load and store operations move data between memory and the run-time stack.

E.2 Implementation

CVM is implemented by three objects in package `edu.citadel.cvm`.

Object `Constants` defines the number of bytes for primitive types plus the number of bytes for the context part of an activation record.

```
object Constants
  {
    const val BYTES_PER_OPCODE  = 1
    const val BYTES_PER_INTEGER = 4
    const val BYTES_PER_ADDRESS = 4
    const val BYTES_PER_CHAR    = 2
    const val BYTES_PER_BOOLEAN = 1
    const val BYTES_PER_CONTEXT = 2*BYTES_PER_ADDRESS
  }
```

Object `OpCode` defines the numeric values for each CVM opcode. In addition, this object defines a `toString()` function that returns a string representation for a numeric value.

```
object OpCode
  {
    // halt opcode
    const val HALT    : Byte = 0

    // load opcodes (move data from memory to top of stack)
    const val LOAD    : Byte =  9
    const val LOADB   : Byte = 10
    const val LOAD2B  : Byte = 11
    const val LOADW   : Byte = 12
    ...

    // arithmetic opcodes
    const val ADD     : Byte = 70
    const val SUB     : Byte = 71
    const val MUL     : Byte = 72
    const val DIV     : Byte = 73
    const val MOD     : Byte = 74
    const val NEG     : Byte = 75
    const val INC     : Byte = 76
    const val DEC     : Byte = 77

    // I/O opcodes
    const val GETCH   : Byte = 80
    const val GETINT  : Byte = 81
    const val GETSTR  : Byte = 82
    const val PUTBYTE : Byte = 83
    const val PUTCH   : Byte = 84
    const val PUTINT  : Byte = 85
    const val PUTEOL  : Byte = 86
    const val PUTSTR  : Byte = 87

    // program/procedure opcodes
    const val PROGRAM : Byte = 90
    const val PROC    : Byte = 91
    const val CALL    : Byte = 92
    const val RET     : Byte = 93
    ...

    fun toString(n : Byte) : String
      {
        ...
      }
  }
```

Object `CVM` is the primary component of the implementation for the virtual machine. In addition to several helper methods, every opcode is implemented by a method. Method `run()` provides the basic control logic for the virtual machine using a large "`when`" statement to dispatch numeric opcodes to their corresponding method calls.

```
fun run()
  {
    var opCode : Byte
    running = true
    pc = 0

    while (running)
      {
        opCode = fetchByte()

        when (opCode)
          {
            OpCode.ADD     -> add()
            OpCode.ALLOC   -> allocate()
            OpCode.BR      -> branch()
            OpCode.BE      -> branchEqual()
            OpCode.BNE     -> branchNotEqual()
            OpCode.BG      -> branchGreater()
            OpCode.BGE     -> branchGreaterOrEqual()
            OpCode.BL      -> branchLess()
            OpCode.BLE     -> branchLessOrEqual()
            OpCode.BZ      -> branchZero()
            OpCode.BNZ     -> branchNonZero()
            OpCode.CALL    -> call()
            OpCode.DEC     -> decrement()
            OpCode.DIV     -> divide()
            OpCode.GETCH   -> getCh()
            OpCode.GETINT  -> getInt()
            OpCode.GETSTR  -> getString()
            ...
            OpCode.SHL     -> shiftLeft()
            OpCode.SHR     -> shiftRight()
            OpCode.STORE   -> store()
            OpCode.STOREB  -> storeByte()
            OpCode.STORE2B -> store2Bytes()
            OpCode.STOREW  -> storeWord()
            OpCode.STOREST -> storeString()
            OpCode.SUB     -> subtract()
            else           -> error("invalid machine instruction")
          }
      }
  }
```

E.3 CVM Instruction Set Architecture

Mnemonic	Short Description	Stack before after	Definition
Arithmetic Opcodes			
ADD	Add: Pop two integers from the stack and push their sum back onto the stack.	n1 n2 n1 + n2	n2 ← popInt() n1 ← popInt() pushInt(n1 + n2)
SUB	Subtract: Pop two integers from the stack and push their difference back onto the stack.	n1 n2 n1 - n2	n2 ← popInt() n1 ← popInt() pushInt(n1 - n2)
MUL	Multiply: Pop two integers from the stack and push their product back onto the stack.	n1 n2 n1*n2	n2 ← popInt() n1 ← popInt() pushInt(n1*n2)
DIV	Divide: Pop two integers from the stack and push their quotient back onto the stack.	n1 n2 n1/n2	n2 ← popInt() n1 ← popInt() pushInt(n1/n2)
MOD	Modulo: Pop two integers from the stack, divide them, and push the remainder back onto the stack.	n1 n2 n1 % n2	n2 ← popInt() n1 ← popInt() pushInt(n1 % n2)
NEG	Negate: Pop an integer from the stack, negate it, and push the result back onto the stack.	n -n	n ← popInt() pushInt(-n)
INC	Increment: Pop an integer from the stack, add 1, and push the result back onto the stack.	n n + 1	n ← popInt() pushInt(n + 1)
DEC	Decrement: Pop an integer from the stack, subtract 1, and push the result back onto the stack.	n n - 1	n ← popInt() pushInt(n - 1)

Logical Opcodes			

`NOT`	Logical Not: Pop a byte from the stack and push its logical negation back onto the stack.	`b`
		`!b`
			`b ← popByte()`
			`if b = 0`
			`  pushByte(1)`
			`else`
			`  pushByte(0)`

Shift Opcodes			
`SHL b`	Shift Left: Pop an integer from the stack, shift the bits left by the specified amount using zero fill, and push the result back onto the stack. Note: Only the right most five bits of the argument are used for the shift.	`n` `n << b`	`n ← popInt()` `pushInt(n << b)`
`SHR b`	Shift Right: Pop an integer from the stack, shift it right by the specified amount using sign extend, and push the result back onto the stack. Note: Only the right most five bits of the argument are used for the shift.	`n` `n >> b`	`n ← popInt()` `pushInt(n >> b)`
Branch Opcodes			
`BR displ`	Branch: Branch unconditionally according to displacement argument (may be positive or negative).		`pc ← pc + displ`

`BE displ`	Branch Equal: Pop two integers from the stack and compare them. If they are equal, then branch according to displacement argument (may be positive or negative); otherwise continue with the next instruction.	`n1`
		`n2`
		———
			`n2 ← popInt()`
			`n1 ← popInt()`
			`if n1 == n2`
			`  pc ← pc + displ`

`BNE displ`	Branch Not Equal: Pop two integers from the stack and compare them. If they are not equal, then branch according to displacement argument (may be positive or negative); otherwise continue with the next instruction.	`n1`
		`n2`
		———
			`n2 ← popInt()`
			`n1 ← popInt()`
			`if n1 != n2`
			`  pc ← pc + displ`

`BG displ`	Branch Greater: Pop two integers from the stack and compare them. If the second integer is greater than the first, then branch according to displacement argument (may be positive or negative); otherwise continue with the next instruction.	`n1`
		`n2`
		———
			`n2 ← popInt()`
			`n1 ← popInt()`
			`if n1 > n2`
			`  pc ← pc + displ`

`BGE displ`	Branch Greater or Equal: Pop two integers from the stack and compare them. If the second integer is greater than or equal to the first, then branch according to displacement argument (may be positive or negative); otherwise continue with the next instruction.	`n1`
		`n2`
		———
			`n2 ← popInt()`
			`n1 ← popInt()`
			`if n1 >= n2`
			`  pc ← pc + displ`

`BL displ`	Branch Less: Pop two integers from the stack and compare them. If the second integer is less than the first, then branch according to displacement argument (may be positive or negative); otherwise continue with the next instruction.	`n1`
		`n2`
		———
			`n2 ← popInt()`
			`n1 ← popInt()`
			`if n1 < n2`
			`  pc ← pc + displ`

`BLE displ`	Branch Less or Equal: Pop two integers from the stack and compare them. If the second integer is less than or equal to the first, then branch according to displacement argument (may be positive or negative); otherwise continue with the next instruction.	`n1`
		`n2`
		———
			`n2 ← popInt()`
			`n1 ← popInt()`
			`if n1 <= n2`
			`  pc ← pc + displ`

`BZ displ`	Branch if Zero: Pop one byte from the stack. If it is zero, then branch according to displacement argument (may be positive or negative); otherwise continue with the next instruction.	`b`
		———
			`b ← popByte()`
			`if b = 0`
			`  pc ← pc + displ`

`BNZ displ`	Branch if Nonzero: Pop one byte from the stack. If it is nonzero then branch according to displacement argument (may be positive or negative); otherwise continue with the next instruction.	`b`
		———
			`b ← popByte()`
			`if b ≠ 0`
			`  pc ← pc + displ`

Load/Store Opcodes			

`LOAD n`	Load multiple bytes onto the stack: The number of bytes to move is part of the instruction. Pop an address from the stack and push n bytes starting at that address onto the stack.	`addr`
		———
		`b1`
		`b2`
		`...`
		`bn`
			`addr ← popInt();`
			`for i ← 0..n-1 loop`
			`  pushByte(mem[addr + i])`

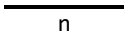
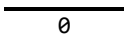
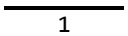
`LOADB`	Load Byte: Load (push) a single byte onto the stack. The address of the byte is obtained by popping it off the stack.	`addr`
		———
		`b`
			`addr ← popInt()`
			`b ← mem[addr]`
			`pushByte(b)`

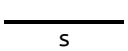
`LOAD2B`	Load Two Bytes: Load (push) two consecutive bytes onto the stack. The address of the first byte is obtained by popping it off the stack.	`addr`
		———
		`b0`
		`b1`
			`addr ← popInt()`
			`b0 ← mem[addr + 0]`
			`b1 ← mem[addr + 1]`
			`pushByte(b0)`
			`pushByte(b1)`

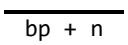
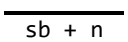
`LOADW`	Load Word: Load (push) a word (four consecutive bytes) onto the stack. The address of the word is obtained by popping it off the stack.	`addr`
		———
		`w`
			`addr ← popInt()`
			`w ← getWord(addr)`
			`pushInt(w)`

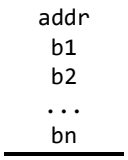
`LOADSTR`	Load String: Load (push) a string (length plus characters) onto the stack in reverse order. The address of the string to load is obtained by popping it off the stack.	`addr`
		———
		`s`
		`(reversed)`
			`addr ← popInt()`
			`strLen ← getInt(addr)`
			`addr ← addr + 4`
			`for i ← 0..strLen-1 loop`
			`  chars[i] ← mem[addr]`
			`  addr ← addr + 2`
			`for i ← strLen-1..0 loop`
			`  pushChar(chars[i])`
			`pushInt(strLen)`

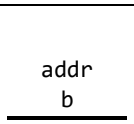
`LDCB b`	Load Constant Byte: Fetch the byte immediately following the opcode and push it onto the stack.	——— `b`	`pushByte(b)`
`LDCB0`	Load Constant Byte `0`: Optimized version of `LDCB 0`.	——— `0`	`pushByte(0)`
`LDCB1`	Load Constant Byte `1`: Optimized version of `LDCB 1`.	——— `1`	`pushByte(1)`
`LDCCH c`	Load Constant Character: Fetch the character immediately following the opcode and push it onto the stack.	——— `c`	`pushChar(c)`

`LDCINT n`	Load Constant Integer: Fetch the integer immediately following the opcode and push it onto the stack.		`pushInt(n)`
`LDCINT0`	Load Constant Integer `0`: Optimized version of `LDCINT 0`.		`pushInt(0)`
`LDCINT1`	Load Constant Integer `1`: Optimized version of `LDCINT 1`.		`pushInt(1)`

`LDCSTR s`	Load Constant String: The string (length plus characters) immediately follows the opcode. Push the string (length and characters) onto the stack.		`n ← fetchInt()`
			`pushInt(n)`
			`for i ← 0..n-1 loop`
			`  c ← fetchChar()`
			`  pushChar(c)`

`LDLADDR n`	Load Local Address: Compute the absolute address of a local variable from its relative address `n` and push the absolute address onto the stack.		`pushInt(bp + n)`
`LDGADDR n`	Load Global Address: Compute the absolute address of a global (program level) variable from its relative address `n` and push the absolute address onto the stack.		`pushInt(sb + n)`

`STORE n`	Store n bytes: Remove `n` bytes from the stack followed by an absolute address and copy the `n` bytes to the location starting at the absolute address.		`for i ← n-1..0 loop`
			`  data[i] ← popByte()`
			`addr ← popInt()`
			`for i ← 0..n-1 loop`
			`  mem[addr + i] ← data[i]`

`STOREB`	Store Byte: Store a single byte at a specified memory location. The byte to be stored and the address where it is to be stored are popped from the stack.		`b ← popByte()`
			`addr ← popInt()`
			`memory[addr] ← b`

STORE2B	Store Two Bytes: Store two bytes at a specified memory location. The bytes to be stored and the address where they are to be stored are popped from the stack.	addr b0 b1	b1 ← popByte() b0 ← popByte() addr ← popInt() mem[addr + 0] ← b0 mem[addr + 1] ← b1
STOREW	Store Word: Store a word (4 bytes) at a specified memory location. The word to be stored and the address where it is to be stored are popped from the stack.	addr w	w ← popWord() addr ← popInt() putWord(w, addr)

STOREST	Store String: Store string at a specified memory location. The string to be stored and the address where it is to be stored are popped from the stack. String characters and string length were pushed in reverse order.	addr
		s
			strLen ← popInt()
			for i ← 0..len-1 loop
			  chars[i] ← popChar()
			addr ← popInt()
			putInt(strLen, addr)
			addr ← addr + 4
			for i ← 0..len-1 loop
			  putChar(chars[i], addr)
			  addr ← addr + 2

ALLOC n	Allocate: Allocate space on the stack for future use.		sp ← sp + n
Program/Procedure Opcodes			
PROGRAM n	Program: Initialize base pointer and allocate space on the stack for the program's local variables.		bp ← sb sp ← bp + n – 1
PROC n	Procedure: Allocate space on the stack for a subprogram's local variables.		sp ← sp + n
CALL disp	Call: Call a subprogram, pushing current values for BP and PC onto the stack.	bp pc	pushInt(bp) pushInt(pc) bp ← sp – 7 pc ← pc + disp

`RET n`	Return: Return from a subprogram, restoring the old value for BP plus space on stack previously allocated for the subprogram's local variables and parameters.		`bpSave ← bp`
			`sp ← bpSave - n - 1`
			`bp ← getInt(bpSave)`
			`pc ← getInt(bpSave + 4)`

`RET0`	Optimized version of `RET 0`.		`bpSave ← bp`
			`sp ← bpSave - 1`
			`bp ← getInt(bpSave)`
			`pc ← getInt(bpSave + 4)`

`RET4`	Optimized version of `RET 4`.		`bpSave ← bp`
			`sp ← bpSave - 5`
			`bp ← getInt(bpSave)`
			`pc ← getInt(bpSave + 4)`

`HALT`	Halt: Stop the virtual machine.		`halt`
I/O Opcodes			

`GETINT`	Get Integer: Read digits from standard input, convert them to an integer, and store the integer at the address on top of stack.	`addr`	`addr ← popInt()`
			`n ← readInt()`
			`putInt(n, addr)`

`GETCH`	Get Character: Read character from standard input and store it at the address on top of stack.	`addr`	`addr ← popInt()`
			`c ← readChar()`
			`putChar(c, addr)`

`GETSTR n`	Get String: Read string from standard input and store it at the address on top of stack.	`addr`	`addr ← popInt()`
			`s ← readStr()`
			`strLen = min(s.length, n)`
			`putInt(n, addr)`
			`for i ← 0..strLen-1 loop`
			`  putChar(s[i], addr)`
			`  addr ← addr + 2`

`PUTBYTE`	Put Byte: Pop byte from top of the stack and write its value to standard output.	`b`	`b ← popByte()` `writeByte(b)`
`PUTINT`	Put Integer: Pop integer from top of the stack and write its value to standard output.	`n`	`n ← popInt()` `writeInt(n)`

`PUTCH`	Put Character: Pop character from top of stack and write its value to standard output.	`c`	`c ← popChar()` `writeChar(c)`

`PUTSTR n`	Put String: Write a string of n characters to standard output. The string (length plus characters) was previously pushed onto the stack.	`s`	`nBytes ← 4 + 2*n`
			`addr ← sp – nBytes + 1`
			`strLen ← getInt(addr)`
			`for i ← 0..strLen–1 loop`
			`  writeChar(mem[2*i])`
			`sp ← sp - n`

`PUTEOL`	Put End-of-Line: Write a line terminator to standard output.		`write(EOL)`

Appendix F
Searching for Reserved Words

An important task of the scanner is to distinguish between programmer-defined identifiers and reserved words. Since identifiers and reserved words account for a large percentage of the symbols in a typical program, this task needs to be implemented efficiently. Indeed, an analysis of the collection of correct CPRL test programs shows that almost half the symbols fall into one of these two categories.

The basic idea adopted by many hand-implemented scanners is to accumulate the characters for an identifier or reserved word into a string and then use a look-up mechanism to determine if the string is one of the reserved words. We encapsulate the details of the look-up mechanism in a method with the following signature.

```
fun getIdentifierSymbol(idString : String) : Symbol
```

The method will return either `Symbol.identifier` or one of the reserved word symbols such a `Symbol.IntegerRW`, `Symbol.ifRW`, `Symbol.loopRW`, `Symbol.varRW`, etc.

This appendix explores several algorithms for implementing this method, and it includes benchmark results for the performance of each algorithm. Other than the sequential algorithms, most of the algorithms discussed have either been used or suggested by compiler writers in the past. Details of the implementations and benchmark analyses are given in the remainder of this appendix, but the results displayed in the following table show that the performance differences can be substantial. Pairwise comparison of the results shows statistically significant differences, even between algorithms where the performances are similar.

Search Algorithm	Benchmark Time (in seconds)	Standard Deviation
Sequential 1	16.258	0.1689
Sequential 2	13.496	0.0179
Sequential 3	3.457	0.0069
Binary	8.316	0.0089
By Length	5.252	0.0173
By First Character	4.568	0.0181
Gperf Hash	3.955	0.0983
When Expression	3.462	0.0082
HashMap	3.096	0.0088