SnailVM Specification

June 12, 2025

Contents

1 Overview

SnailVM is a stack-based virtual machine designed to execute programs written in the Snail programming language, a statically-typed language supporting functions, loops, conditionals, and arrays. The virtual machine processes a compact bytecode format that represents Snail programs efficiently. This specification details the bytecode file structure, instruction set, type system, error handling, and includes examples of compilation from Snail source code to bytecode.

2 Bytecode File Format

The SnailVM bytecode file is a binary format organized into distinct sections to facilitate program execution. Each section is described below with its specific format.

2.1 Header Format

Field	Size	Description
Magic Number	4 bytes	Fixed value 0x534E4131 (ASCII: SNA1) to identify the file.
Version	2 bytes	Bytecode format version in big-endian (e.g., 0x0001 for version 1).
Main Function Index	2 bytes	Index of the main function in the function table (big-endian). If main is absent, set to 0xFFFF (-1).

Table 1: Header Format

2.2 Constant Pool Format

Type ID	Type	Binary Format
0x01	i32	4-byte signed integer (big-endian).
0x02	usize	4-byte unsigned integer (big-endian).
0x03	string	2-byte length (unsigned, big-endian), followed by UTF-8 bytes.

Table 2: Constant Pool Entry Format

The pool starts with a 2-byte (big-endian) number indicating the number of entries.

2.3 Global Variables Format

Field	Description
Number of Variables	2 bytes (big-endian), number of variables.
Name Length	1 byte, length of the variable name (n).
Name	n bytes, UTF-8 encoded variable name.
Type ID	1 byte, type of the variable (see Table 11).
(Array only)	If Type ID = 0x04 (array): 1 byte for element type ID, 4 bytes (big-endian) for array size.

Table 3: Global Variable Entry Format

2.4 Function Table Format

Field	Description
Number of Functions	2 bytes (big-endian), number of functions.
Name Length	1 byte, length of the function name (n).
Name	n bytes, UTF-8 encoded function name.
Number of Parameters	1 byte, count of function parameters.
Return Type	1 byte, type ID (see Table 11).
Number of Local Variables	2 bytes (big-endian), count of local variables (including parameters).
Bytecode Length	4 bytes (big-endian), length of the function's bytecode (t).
Bytecode	t bytes, the function's executable bytecode.

Table 4: Function Table Entry Format

2.5 Function Call and Local Variable Pool

When a function is called, the caller pushes all arguments onto the stack in left-to-right order. Upon entering the function, the callee pops the arguments from the stack in reverse order (last argument first), assigning them to local variable slots 0..N-1, where N is the number of parameters. The local variable pool for a function thus starts with its parameters, followed by other local variables declared in the function body. All accesses to parameters and local variables use their respective indices in this pool.
Instruction PUSH_LOCAL is used to read values from local variable pool by their index, while STORE_LOCAL is used to write values to local variable pool. Both instructions take a 2-byte index (big-endian) that points to the variable's position in the local variable pool.

2.5.1 Control Flow

Control flow statements (conditionals, loops) evaluate conditions first. For if statements, if the condition is true, the body is executed, otherwise the else branch (if present). For loops, as long as the condition is true, the body is executed repeatedly.
For logical operators && (AND) and || (OR), short-circuit evaluation is implemented. In the case of AND, if the first operand evaluates to false, the second operand is not evaluated, and false is immediately returned. For OR, if the first operand evaluates to true, the second operand is not evaluated, and true is immediately returned. This behavior is implemented using conditional jumps in the bytecode.

2.6 Global Bytecode

This section contains bytecode executed before the main function, typically for initializing global variables. It starts with a 4-byte (big-endian) length, followed by the bytecode.

3 Bytecode Generation Architecture

3.1 General Approach

Bytecode generation in SnailVM is based on object-oriented design principles. Each node of the abstract syntax tree (AST) implements the emitBytecode method, which is responsible for generating its own bytecode. The central facade, BytecodeEmitter, only assembles the final file, delegating all instruction generation logic to the nodes themselves.
Instructions that control program flow, such as JMP, JMP_IF_FALSE, and JMP_IF_TRUE, use signed 16-bit offsets measured in bytes relative to the current position after the jump instruction itself. This allows for forward and backward jumps within code sections.

3.2 Variable Pools

Global variable pool is formed from all variables declared at the top level of the program. Each variable receives a unique index in the pool, which is used for generating PUSH_GLOBAL, STORE_GLOBAL, etc.
Local variable pool is formed separately for each function. It includes:

- Function parameters (indices 0..N-1)
- All variables declared inside the function body (indices continue after parameters)

Access to local variables is performed via PUSH_LOCAL, STORE_LOCAL instructions with the corresponding index.

3.3 Bytecode Generation Context

The BytecodeContext class stores tables of constants, global variables, functions, and local variables. It provides methods for obtaining indices and adding new elements. Each AST node uses this context for correct addressing during bytecode generation.

3.4 Example: Bytecode Generation for a Number Literal

Source Code

```
public class NumberLiteral extends PrimaryExpression {
  private final long value;
  // ...
  @Override
  public void emitBytecode(ByteArrayOutputStream out, BytecodeContext
     context) throws IOException {
  int constIndex = context.addConstant(value);
  out.write(BytecodeConstants.Opcode.PUSH_CONST);
  BytecodeUtils.writeU16(out, constIndex);
  }
  }
```

3.5 Advantages

- Easy to extend: adding new constructs only requires implementing the emitBytecode method in the new node.
- Clean architecture: the facade contains no instruction generation logic.
- Simplified maintenance and testing.

4 Instruction Set

SnailVM employs a stack-based architecture with single-byte opcodes.

4.1 Stack and Memory Operations

Opcode	Name	Description	Arguments	Stack Effect
0x01	PUSH_CONST	Pushes a constant from the constant pool.	2-byte index (big-endian)	[] → [value]
0x02	PUSH_LOCAL	Pushes a local variable's value onto the stack.	2-byte index (big-endian)	[] → [value]
0x03	PUSH_GLOBAL	Pushes a global variable's value onto the stack.	2-byte index (big-endian)	[] → [value]
0x04	STORE_LOCAL	Stores the top stack value into a local variable.	2-byte index (big-endian)	[value] → []
0x05	STORE_GLOBAL	Stores the top stack value into a global variable.	2-byte index (big-endian)	[value] → []
0x06	POP	Removes the top value from the stack.	None	[value] → []
0x07	DUP	Duplicates the top value on the stack.	None	[value] → [value, value]

Table 5: Stack and Memory Operations

4.2 Arithmetic and Logic Operations

Opcode	Name	Description	Arguments	Stack Effect
0x10	ADD	Adds the top two integers on the stack.	None	[a, b] $\rightarrow$ [a+b]
0x11	SUB	Subtracts the top integer from the second-top integer.	None	[a, b] $\rightarrow$ [a-b]
0x12	MUL	Multiplies the top two integers on the stack.	None	[a, b] $\rightarrow$ [a*b]
0x13	DIV	Divides the second-top integer by the top integer.	None	[a, b] $\rightarrow$ [a/b]
0x14	MOD	Computes the remainder of the division of the second-top integer by the top integer.	None	[a, b] $\rightarrow$ [a % b]
0x20	EQ	Checks if the top two values are equal.	None	[a, b] $\rightarrow$ [bool]
0x21	NEQ	Checks if the top two values are not equal.	None	[a, b] $\rightarrow$ [bool]
0x22	LT	Checks if the second-top integer is less than the top integer.	None	[a, b] $\rightarrow$ [bool]
0x23	LE	Checks if the second-top integer is less than or equal to the top integer.	None	[a, b] $\rightarrow$ [bool]
0x24	GT	Checks if the second-top integer is greater than the top integer.	None	[a, b] $\rightarrow$ [bool]
0x25	GTE	Checks if the second-top integer is greater than or equal to the top integer.	None	[a, b] $\rightarrow$ [bool]
0x26	AND	Performs a logical AND on the top two booleans.	None	[a, b] $\rightarrow$ [a && b]
0x27	OR	Performs a logical OR on the top two booleans.	None	[a, b] $\rightarrow$ [a || b]
0x28	NOT	Negates the top boolean value.	None	[a] $\rightarrow$ [!a]

Table 6: Arithmetic and Logic Operations

4.3 Control Flow Operations

Opcode	Name	Description	Arguments	Stack Effect
0x30	JMP	Unconditional jump to the specified offset in bytes relative to the current position after this instruction.	2-byte signed offset (big-endian)	$[] \rightarrow []$
0x31	JMP_IF_FALSE	Jump if the top value is 0 (false) to the specified offset in bytes relative to the current position after this instruction.	2-byte signed offset (big-endian)	$[bool] \rightarrow []$
0x35	JMP_IF_TRUE	Jump if the top value is not 0 (true) to the specified offset in bytes relative to the current position after this instruction.	2-byte signed offset (big-endian)	$[bool] \rightarrow []$
0x32	CALL	Calls a function at the specified index, passing arguments from the stack.	2-byte function index (big-endian)	$[args] \rightarrow [ret]$
0x33	RET	Returns from a function with the top stack value as the return value.	None	$[value] \rightarrow []$
0x34	HALT	Stops the execution of the virtual machine.	None	$[] \rightarrow []$

Table 7: Control Flow Operations

4.4 Array Operations

Opcode	Name	Description	Arguments	Stack Effect
0x40	NEW_ARRAY	Creates a new uninitialized array of the specified size and type, pushing its reference onto the stack.	2-byte size, 1-byte type ID	[] → [array]
0x41	GET_ARRAY	Retrieves the element at the specified index from the array.	None	[index, array] → [value]
0x42	SET_ARRAY	Sets the element at the specified index in the array.	None	[index, value, array] → []
0x43	INIT_ARRAY	Initializes an array with a specified number of elements from the stack. The elements are popped from the stack and placed into the array.	2-byte size (N)	[elem1, ..., elemN, array] → [array]

Table 8: Array Operations

4.5 Intrinsic Instructions

Opcode	Name	Description	Arguments	Stack Effect
0x50	INTRINSIC_CALL	Calls a built-in intrinsic function identified by its index in the Intrinsic Table.	2-byte index (big-endian)	[args] → [ret]

Table 9: Intrinsic Instructions

4.6 Intrinsic Functions

Name	Parameters	Return Type	Description	Stack Effect
println	1 (any type convertible to string)	void (0x00)	Outputs the argument to the console followed by a newline and discards the top stack value.	[value] → []

Table 10: Intrinsic Functions

Note: The Intrinsic Table can be extended with more built-in functions as needed. Each intrinsic is identified by its index in the table.

5 Type System

SnailVM supports i32, usize, string, void, and array types.

5.1 Type Identifiers

Type ID	Type
0x00	void
0x01	i32 (also used for bool)
0x02	usize
0x03	string
0x04	array

Table 11: Type Identifiers

Note: The type bool is represented as i32 (0 for false, 1 for true) in the bytecode and type tables.

6 Error Diagnostics

The SnailL compiler, upon encountering an error, outputs a detailed message including:

- The relevant fragment of the source code
- A pointer line with the $character under the error location The error type$
- A human-readable error description

Example: Type mismatch

Source Code

```
let x: i32 = "abc";
```

```
ERROR:
let x: i32 = "abc";
^^^
TYPE_MISMATCH
=====================================
Type mismatch: cannot assign string to i32
=====================================
```

Example: Unknown variable

Source Code

```
y = 5;
```

```
ERROR:
y = 5;
^
UNKNOWN_VARIABLE
=====================================
Unknown variable: y
=====================================
```

Example: Unknown operator

Source Code

```
let x: i32 = 1 %% 2;
```

ERROR:
let x: i32 = 1 %% 2;
^
UNKNOWN_OPERATOR
==================================
Unknown operator: %%
==================================

Example: Dead code

Source Code

```
fn f() {
  return 1;
  let x = 2; // dead code
  }
```

ERROR:
return 1;
let x = 2;
DEAD_CODE
==================================
Code after return is unreachable
==================================

Note: All errors always include the source line and the error position using the *character*.

bool type

The bool type is represented as i32 (0 — false, 1 — true) in all type tables and in bytecode.

7 Compilation Example

7.1 Source Code

Source Code

```
let counter: i32 = 0;
  let data: [i32; 5] = [10, 20, 30, 40, 50];

  fn computeSum(a: i32, b: i32) -> i32 {
  let sum: i32 = a + b;
  let offset: i32 = 5;
  return sum + offset;
  }

  fn main() -> void {
  let i: i32 = 0;
  while (i < 5) {
  let value: i32 = data[i];
  if (value > 25) {
  data[i] = value * 2;
  } else {
  data[i] = value - 5;
  }
  i = i + 1;
  }
  let sum: i32 = computeSum(data[0], data[1]);
  if (sum >= 100) {
  counter = sum / 2;
  } else {
  counter = sum;
  }
  let flag: i32 = 0;
  if (counter < 50 && data[2] > 50) {
  flag = 1;
  }
  println(sum); // Example usage of println
  }
```

7.2 Bytecode Output

=== HEADER SECTION ===
[HEADER] Magic: SNA1
[HEADER] Version: 1
[HEADER] Main function index: 1

=== CONSTANTS SECTION ===
[CONSTANTS] Count: 11
  [CONST] i32: 0
  [CONST] i32: 10
  [CONST] i32: 20
  [CONST] i32: 30
  [CONST] i32: 40
  [CONST] i32: 50

```
  [CONST] i32: 5
  [CONST] i32: 25
  [CONST] i32: 2
  [CONST] i32: 1
  [CONST] i32: 100

=== GLOBALS SECTION ===
[GLOBALS] Count: 2
  [GLOBAL] counter : i32
  [GLOBAL] data : array (elemType: i32, size: 5)

=== FUNCTIONS SECTION ===
[FUNCTIONS] Count: 2
  [FUNC] computeSum (params: 2, return: i32, locals: 4, code length: 30 bytes, offset: 0x0000006C)
    0000: STORE_LOCAL 1 (параметр #1)
    0003: STORE_LOCAL 0 (параметр #0)
    0006: PUSH_LOCAL 0 (параметр #0)
    0009: PUSH_LOCAL 1 (параметр #1)
    000C: ADD
    000D: STORE_LOCAL 2 (локальная переменная #0)
    0010: PUSH_CONST 6 (i32: 5)
    0013: STORE_LOCAL 3 (локальная переменная #1)
    0016: PUSH_LOCAL 2 (локальная переменная #0)
    0019: PUSH_LOCAL 3 (локальная переменная #1)
    001C: ADD
    001D: RET
  [FUNC] main [main] (params: 0, return: void, locals: 4, code length: 181 bytes, offset: 0x00000097)
    0000: PUSH_CONST 0 (i32: 0)
    0003: STORE_LOCAL 0 (локальная переменная #0)
    0006: PUSH_LOCAL 0 (локальная переменная #0)
    0009: PUSH_CONST 6 (i32: 5)
    000C: LT
    000D: JMP_IF_FALSE 65 [to 0x0051]
    0010: PUSH_GLOBAL 1 (data, array of i32[5])
    0013: PUSH_LOCAL 0 (локальная переменная #0)
    0016: GET_ARRAY
    0017: STORE_LOCAL 1 (локальная переменная #1)
    001A: PUSH_LOCAL 1 (локальная переменная #1)
    001D: PUSH_CONST 7 (i32: 25)
    0020: GT
    0021: JMP_IF_FALSE 17 [to 0x0035]
    0024: PUSH_LOCAL 1 (локальная переменная #1)
    0027: PUSH_CONST 8 (i32: 2)
    002A: MUL
    002B: PUSH_GLOBAL 1 (data, array of i32[5])
    002E: PUSH_LOCAL 0 (локальная переменная #0)
    0031: SET_ARRAY
    0032: JMP 14 [to 0x0043]
    0035: PUSH_LOCAL 1 (локальная переменная #1)
    0038: PUSH_CONST 6 (i32: 5)
    003B: SUB
    003C: PUSH_GLOBAL 1 (data, array of i32[5])
    003F: PUSH_LOCAL 0 (локальная переменная #0)
```

```
0042: SET_ARRAY
0043: PUSH_LOCAL 0 (локальная переменная #0)
0046: PUSH_CONST 9 (i32: 1)
0049: ADD
004A: STORE_LOCAL 0 (локальная переменная #0)
004D: POP
004E: JMP -75 [to 0x0006]
0051: PUSH_GLOBAL 1 (data, array of i32[5])
0054: PUSH_CONST 0 (i32: 0)
0057: GET_ARRAY
0058: PUSH_GLOBAL 1 (data, array of i32[5])
005B: PUSH_CONST 9 (i32: 1)
005E: GET_ARRAY
005F: CALL 0 (computeSum, 2 параметра)
0062: STORE_LOCAL 2 (локальная переменная #2)
0065: PUSH_LOCAL 2 (локальная переменная #2)
0068: PUSH_CONST 10 (i32: 100)
006B: GTE
006C: JMP_IF_FALSE 14 [to 0x007D]
006F: PUSH_LOCAL 2 (локальная переменная #2)
0072: PUSH_CONST 8 (i32: 2)
0075: DIV
0076: STORE_GLOBAL 0 (counter, i32)
0079: POP
007A: JMP 7 [to 0x0084]
007D: PUSH_LOCAL 2 (локальная переменная #2)
0080: STORE_GLOBAL 0 (counter, i32)
0083: POP
0084: PUSH_CONST 0 (i32: 0)
0087: STORE_LOCAL 3 (локальная переменная #3)
008A: PUSH_GLOBAL 0 (counter, i32)
008D: PUSH_CONST 5 (i32: 50)
0090: LT
0091: JMP_IF_FALSE 12 [to 0x00A0]
0094: POP
0095: PUSH_GLOBAL 1 (data, array of i32[5])
0098: PUSH_CONST 8 (i32: 2)
009B: GET_ARRAY
009C: PUSH_CONST 5 (i32: 50)
009F: GT
00A0: JMP_IF_FALSE 7 [to 0x00AA]
00A3: PUSH_CONST 9 (i32: 1)
00A6: STORE_LOCAL 3 (локальная переменная #3)
00A9: POP
00AA: PUSH_LOCAL 3 (локальная переменная #3)
00AD: INTRINSIC_CALL 0
00B0: POP
00B1: PUSH_CONST 0 (i32: 0)
00B4: RET

=== INTRINSICS SECTION ===
[INTRINSICS] Count: 1
  [INTRINSIC] println (params: 1, return: void)
```

```
=== GLOBAL CODE SECTION ===
[GLOBAL CODE] Length: 34
  [Найден вызов main в глобальном коде на смещении 0x001F]
  0000: PUSH_CONST 0 (i32: 0)
  0003: STORE_GLOBAL 0 (counter, i32)
  0006: PUSH_CONST 5 (i32: 50)
  0009: PUSH_CONST 4 (i32: 40)
  000C: PUSH_CONST 3 (i32: 30)
  000F: PUSH_CONST 2 (i32: 20)
  0012: PUSH_CONST 1 (i32: 10)
  0015: NEW_ARRAY 5 i32
  0019: INIT_ARRAY 5
  001C: STORE_GLOBAL 1 (data, array of i32[5])
  001F: CALL 1 (main)

=== BYTECODE STATISTICS ===
  [Общий размер] 382 байт (0x17E)
  [Секции] Заголовок: 8 байт, Константы: 2 байт (11 записей), Функции: 211 байт кода (2 функций), Глобальный код: 161 байт
  [Глобальные переменные] 2 переменных
```

7.3 Execution Flow

- Header: Points to main (index 1).
- Constant Pool: Contains 0, 5, 10, 20, 30, 40, 50, 25, 2, 1, 100.
- Function Table: - computeSum: Computes and returns a + b + 5. - main: Executes loop, calls computeSum, updates counter, sets flag, and prints sum with println.
- Intrinsic Table: Contains println with 1 parameter and void return type.
- Global Variables: counter and data.
- Global Bytecode: Initializes counter and data, calls main.

8 Compilation Process

The Snail compiler parses the source code, generates an abstract syntax tree (AST), and emits bytecode based on the instruction set.

9 Conclusion

SnailVM provides an efficient and robust bytecode format for executing Snail programs.