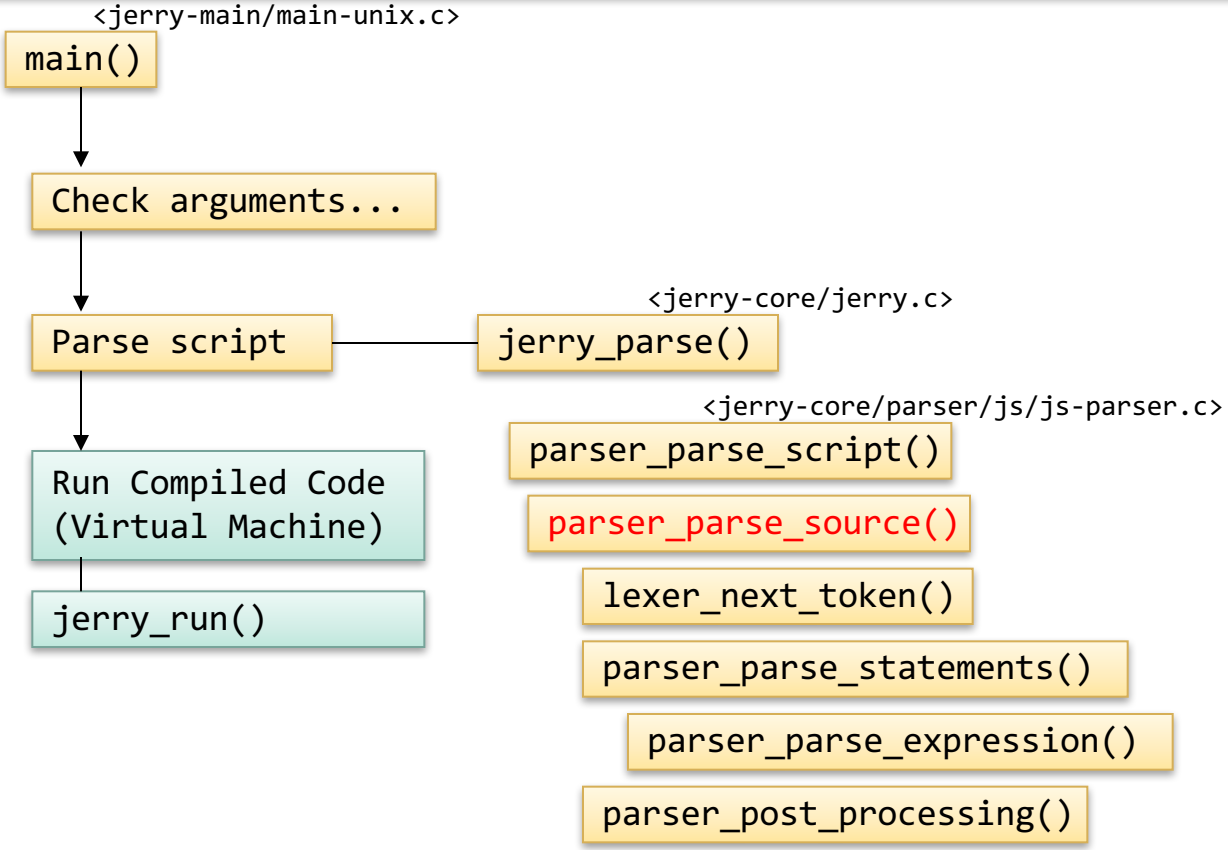
Tizen/Artik IoT Lecture Chapter 3. JerryScript Parser & VM

Sungkyunkwan University

Contents

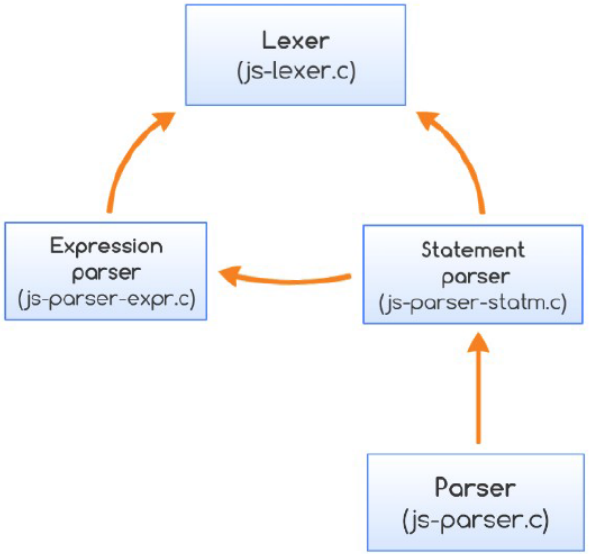
- **JerryScript Execution Flow**
- **JerryScript Parser**
  - Execution Flow
  - Lexing
  - Parsing
  - Compact Bytecode (CBC)
- **JerryScript VM(Virtual Machine)**
  - Virtual Machine
  - Execution Flow
  - vm_run(), vm_execute(), vm_loop()

JerryScript Parser: Execution Flow



JerryScript Parser

- **Implemented as a recursive descent parser**
- **The parser converts the JavaScript source code directly into byte-code.**
- **Components**
  - Lexer
  - Scanner
  - Parser
    - Statement Parser
    - Expression Parser



JerryScript Parser: Components

- **Lexer**
  - Tokenize a source into the meaningful words
- **Scanner**
  - Some tokens have multiple meanings dependent to following tokens, and scanner scans following tokens in order to specify current token
  - e.g. ***for*** could be for-in statement or general for statement
    '\' could be a start of regular expression or a division operator
- **Predictive Top-down Parser**
  - Make a semantic from the **statements** to the possible **terminals**
    - Possible terminals: tokens predictable by current token
      e.g. ***for*** of for-in statement is followed by *(in) statement*

Lexing (1/2)

`parser_parse_source()`

`lexer_next_token()`

1. Read a word from current `source_p`
2. Identify the word as a certain token; identifier, number, string, or operators
3. Fill the token information in `context_p (parser_context_t)` as an output

```
if (context_p->source_p[0] >= LIT_CHAR_0 && context_p->source_p[0] <= LIT_CHAR_9)
{
  lexer_parse_number (context_p);
  return;
}
```

```
typedef struct
{
  uint8_t type;
  uint8_t literal_is_reserved;

  uint8_t extra_value;
  uint8_t was_newline;
  parser_line_counter_t line;
  parser_line_counter_t column;
  lexer_lit_location_t lit_location;
} lexer_token_t;
```

```
context_p->token.type = LEXER_LITERAL;
context_p->token.literal_is_reserved = PARSER_FALSE;
context_p->token.extra_value = LEXER_NUMBER_DECIMAL;
context_p->token.lit_location.char_p = source_p;
context_p->token.lit_location.type = LEXER_NUMBER_LITERAL;
context_p->token.lit_location.has_escape = PARSER_FALSE;
```

literal has ecape character (\ )?

Lexing (2/2): String

`lexer_next_token()`

`lexer_parse_string()`

1. Calculate character size by character encoding type
2. Store current `source_p` to `string_start_p`
3. Move `source_p` to the end of string
4. Fill literal information.

```
context_p->token.type = LEXER_LITERAL;

/* Fill literal data. */
context_p->token.lit_location.char_p = string_start_p;
context_p->token.lit_location.length = (uint16_t) length;
context_p->token.lit_location.type = LEXER_STRING_LITERAL;
context_p->token.lit_location.has_escape = has_escape;

context_p->source_p = source_p + 1;
context_p->line = line;
context_p->column = (parser_line_counter_t) (column + 1);
```

Parsing (1/2)

`parser_parse_source()`

`lexer_next_token()`

`parser_parse_statements()` `parser_parse_statements()`

1. Parsing the tokens in a statement with a **parser stack**
2. Compile the statement to an opcode
3. Emit the opcodes except for last opcode
   Last opcode remain for morphing based on next token

e.g. `this.a = ...` 2 morphings happen

`this` → `CBC_PUSH_THIS`

`this.a` → `CBC_PUSH_PROP_THIS_LITERAL`

`this.a =` → `CBC_ASSIGN_PROP_THIS_LITERAL`

```
else if (context_p->last_cbc_opcode == CBC_PUSH_THIS)
{
  context_p->last_cbc_opcode = PARSER_CBC_UNAVAILABLE;
  parser_emit_cbc_literal_from_token (context_p, CBC_PUSH_PROP_THIS_LITERAL);
}
else
{
  parser_emit_cbc_literal_from_token (context_p, CBC_PUSH_PROP_LITERAL);
}
```

Parsing (2/2)

- **Parsing process**

Stack depth OPCODE Arguments

```
[  1] CBC_PUSH_NUMBER_0
[  0] CBC_ASSIGN_SET_IDENT idx:0->ident(i)
[  0] CBC_JUMP_FORWARD
[  1] CBC_CREATE_ARRAY
[  0] CBC_ASSIGN_SET_IDENT_BLOCK idx:1->ident(a)
[  1] CBC_PUSH_NUMBER_0
[  0] CBC_ASSIGN_SET_IDENT idx:2->ident(j)
[  0] CBC_JUMP_FORWARD
[  3] CBC_PUSH_THREE_LITERALS idx:1->ident(a) idx:2->ident(j) idx:2->ident(j)
[  0] CBC_ASSIGN_BLOCK
[  0] CBC_POST_INCR_IDENT idx:2->ident(j)
[  1] CBC_LESS_TWO_LITERALS idx:2->ident(j) idx:3->number(5000)
[  0] CBC_BRANCH_IF_TRUE_BACKWARD
[  0] CBC_POST_INCR_IDENT idx:0->ident(i)
[  1] CBC_PUSH_LITERAL idx:0->ident(i)
[  2] CBC_PUSH_NUMBER_POS_BYTE number:3
[  1] CBC_LESS
[  0] CBC_BRANCH_IF_TRUE_BACKWARD
```

Example Code

```
for (i = 0; i < 3; i++)
{
  a = [];

  for (j = 0; j < 5000; j++)
  {
    a[j] = j;
  }
}
```

Literal Object in Parsing

- **When literal object is constructed during parsing,**

```
if (!context_p->token.was_newline
    || LEXER_IS_BINARY_OP_TOKEN (context_p->token.type)
    || context_p->token.type == LEXER_LEFT_PAREN
    || context_p->token.type == LEXER_LEFT_SQUARE
    || context_p->token.type == LEXER_DOT)
{
  /* The string is part of an expression statement. */
  context_p->status_flags = status_flags;

  lexer_construct_literal_object (context_p, &lit_location, LEXER_STRING_LITERAL);
  parser_emit_cbc_literal_from_token (context_p, CBC_PUSH_LITERAL);
  /* The extra_value is used for saving the token. */
  context_p->token.extra_value = context_p->token.type;
  context_p->token.type = LEXER_EXPRESSION_START;
  break;
}
```

- **Parser puts the literal object into literal storage if not exist**

```
literal_p = (lexer_literal_t *) parser_list_append (context_p, &context_p->literal_pool);
literal_p->prop.length = (uint16_t) length;
literal_p->type = literal_type;
literal_p->status_flags = has_escape ? 0 : LEXER_FLAG_SOURCE_PTR;
```

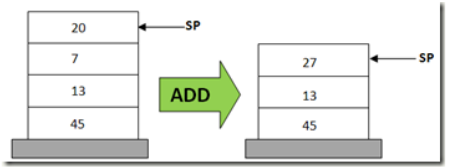
Compact Bytecode (CBC)

- **CBC is a unique variable length byte code with lightweight data compression**
- **Currently 306 opcodes are defined**
  - Majority of the opcodes are variants of the same operation
  - e.g. “this.name” is a frequent expression in JavaScript, so an opcode is defined to resolve this expression

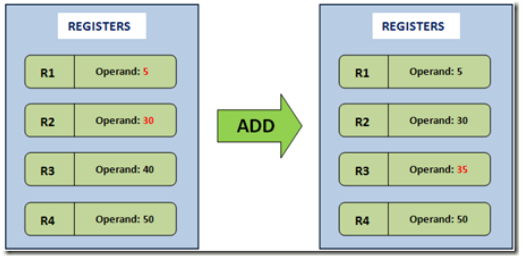
```
CBC_ASSIGN_PROP_THIS_LITERAL
CBC_EXT_FOR_IN_GET_NEXT
CBC_PUSH_NUMBER_0
CBC_PUSH_TRUE
CBC_PUSH_FALSE
```

Virtual Machine

- **Virtual machine executes an instruction stream in software**
- **Stack-based VMs**
  - Use a stack to store intermediate results, variables, etc.
  - Operands are located on stack
  - No need to specify location of operands
  - No need to load operand locations
- **Register-based VMs**
  - Use a limited set of registers for that purpose, like a real CPU
  - Fewer VM instructions needed
  - Each VM instruction is more expensive



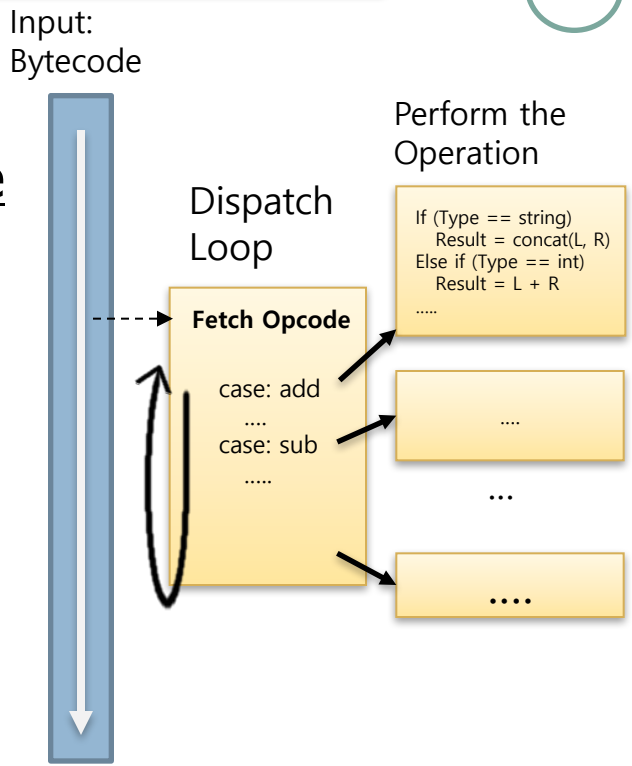
1. POP 20
2. POP 7
3. ADD 20, 7, result
4. PUSH result



1. **ADD R1, R2, R3 ;** # Add contents of R1 and R2, store result in R3

Virtual Machine

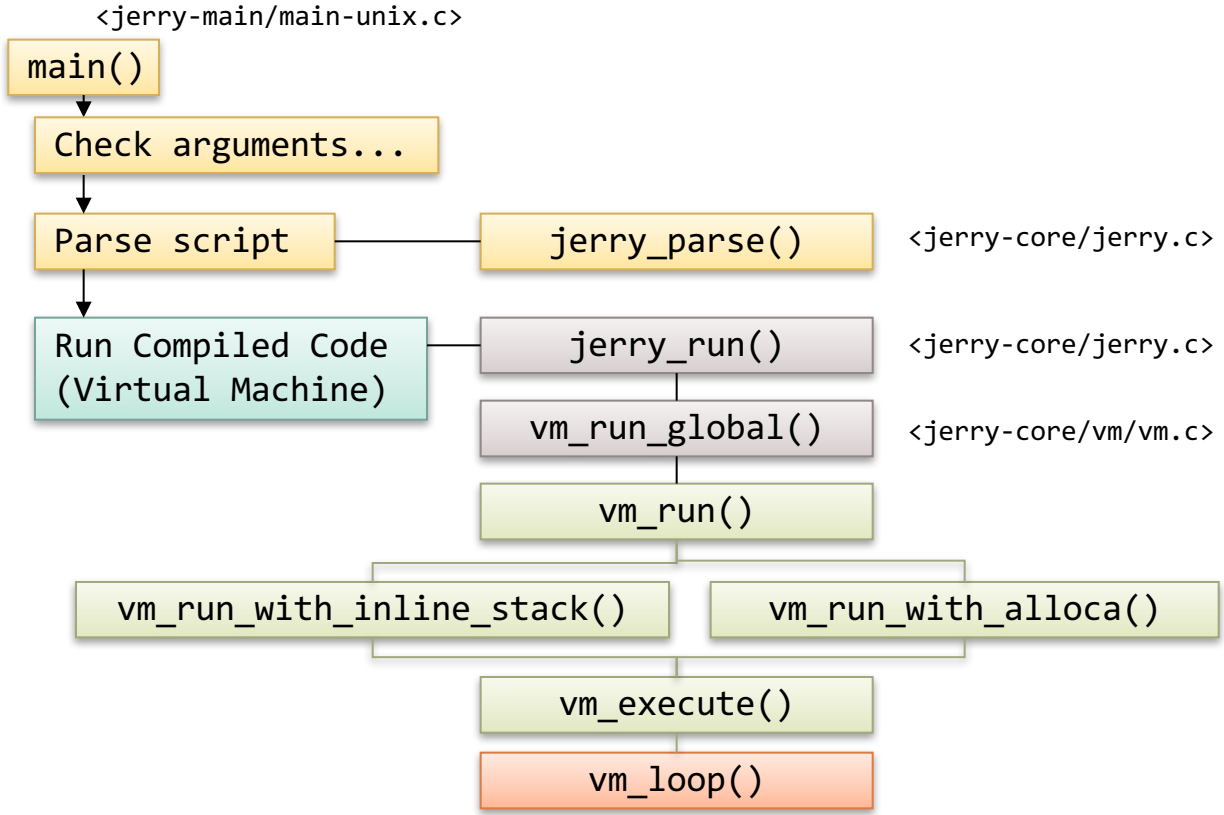
- **Input**: a sequence of instructions
  - Each instruction is identified by its opcode
- **Dispatch**
  - Fetch opcode & jump to implementation
  - Most expensive part of execution
- **Fetch Operands**
- **Perform the Operation**
  - Often cheapest part of executions

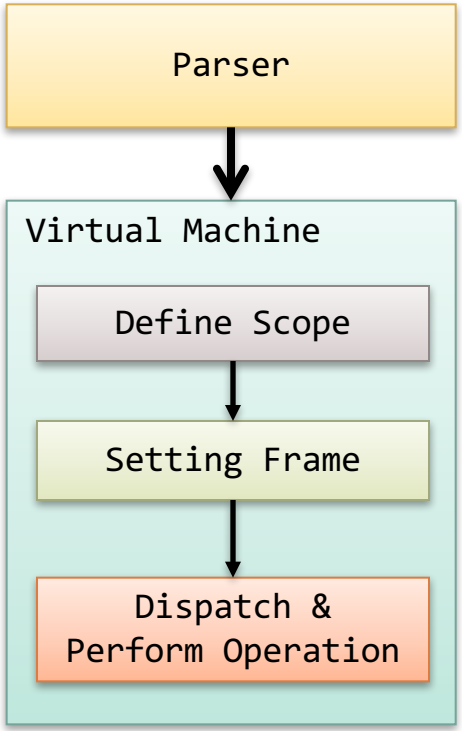


JerryScript VM

- The interpreter is a combination of **register and stack machines**
  - **Stack** is used for computing temporary values
  - **Registers** are used for storing local variables
- The main loop is non-recursive to reduce stack usage
- Byte-code decompression
  - Byte codes are decoded into a maximum of three atomic opcodes and these opcodes are executed

JerryScript VM: Execution Flow

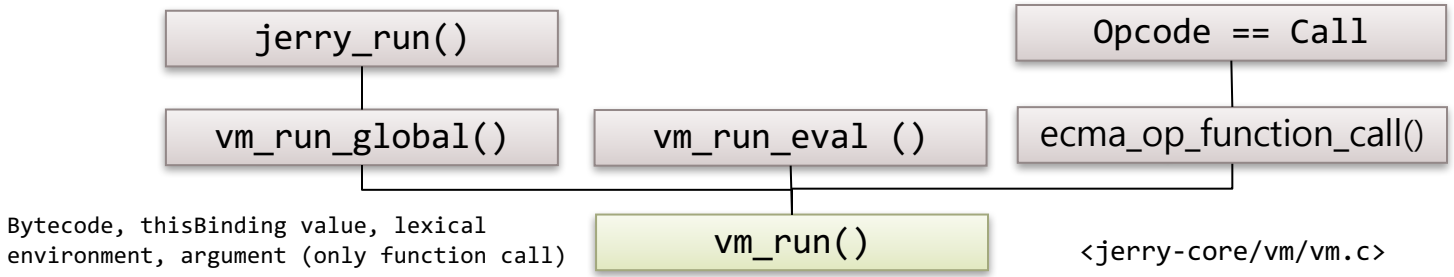




Define Scope

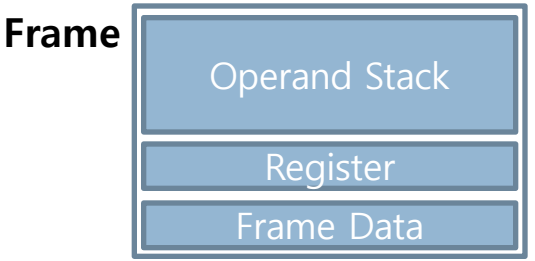
- **vm_run_global():** Global scope
- **vm_run_eval()**
  - Direct call = Scope of Top Context (Inheritance)
  - Indirect call: Global Scope
- **ecma_op_function_call()**
  - Function Scope (Local Scope)

Scope
- ThisBinding Value
- Lexical environment

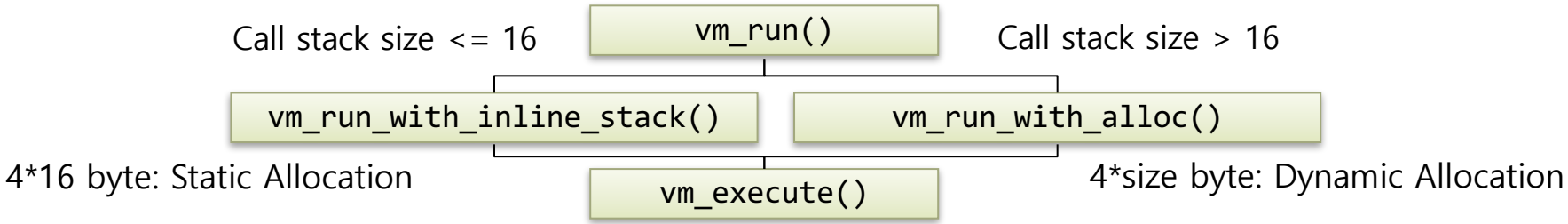


Setting Frame

- **Setting Frame**
  - Frame: execution context of interpreter
  - Bytecode pointer / register start pointer / stack top pointer / current lexical environment / this binding / literal list start pointer

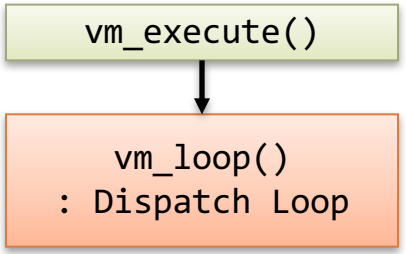


- **Call Stack Size**



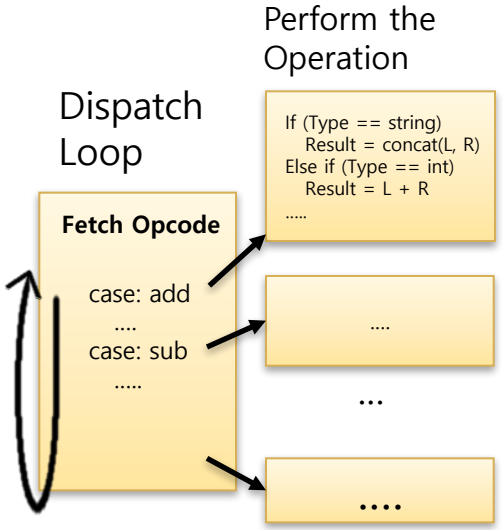
VM Execute Function

1. Set stack top, registers in frame
2. **while(true)**
    1. Call vm_loop(): Dispatch Loop
    2. Check return value
        1. No more execute operation ➔ Break
        2. Invoke a function ➔ opfunc_call() → … → vm_run()
        3. Constructor method ➔ opfunc_construct()
3. Free registers



VM Loop: Dispatch Loop

- **Outer loop for opcode's result handling**
- **Internal loop for byte code execution**
  - Fetch operands
  - Dispatch
    - Fetch opcode & jump to implementation
    - Most expensive part of execution
  - Perform the operation
    - Often cheapest part of executions



Internal Loop in vm_loop()

- **Internal loop for byte code execution**
  - Get operand
    - Setting Left value & right value
    - Branch & Literal: bytecode
    - Stack: Top stack of frame

Literal
- Value of Register Or String

```
typedef enum
{
  VM_OC_GET_NONE = VM_OC_GET_ARGS_CREATE_INDEX (0),          /**< do nothing */
  VM_OC_GET_BRANCH = VM_OC_GET_ARGS_CREATE_INDEX (1),         /**< branch argument */
  VM_OC_GET_STACK = VM_OC_GET_ARGS_CREATE_INDEX (2),         /**< pop one element from the stack */
  VM_OC_GET_STACK_STACK = VM_OC_GET_ARGS_CREATE_INDEX (3),     /**< pop two elements from the stack */

  VM_OC_GET_LITERAL = VM_OC_GET_ARGS_CREATE_INDEX (4),       /**< resolve literal */
  VM_OC_GET_LITERAL_LITERAL = VM_OC_GET_ARGS_CREATE_INDEX (5), /**< resolve two literals */
  VM_OC_GET_STACK_LITERAL = VM_OC_GET_ARGS_CREATE_INDEX (6),   /**< pop one element from the stack
                                                                *   and resolve a literal */
  VM_OC_GET_THIS_LITERAL = VM_OC_GET_ARGS_CREATE_INDEX (7),    /**< get this and resolve a literal */
} vm_oc_get_types;
```

  - Dispatch: Fetch opcode & jump to implementation

Example: Perform the Operation

- **Push**
  - Push operand value to stack
  - Continue to next bytecode (continue;)

```
case VM_OC_PUSH:
{
  *stack_top_p++ = left_value;
  continue;
}
```

- **Assign**
  - Assigning result value
  - Result handling (break)

```
case VM_OC_ASSIGN:
{
  result = left_value;
  left_value = ecma_make_simple_value (ECMA_SIMPLE_VALUE_UNDEFINED);
  break;
}
```

- **Add**
  - Type Checking & Add
    - <int, int>, <int, float>, <float, int>, <float, float>, <string, string>
  - Result handling (break)

Outer Loop in vm_loop()

- **Outer loop for exception handling**
  - Flags for result handling
    - **IDENT Flag**: Assign result to register or string
      - **STACK Flag** By bytecodes
        - » Copy result to stack
    - **Reference Flag**
      - Popping property and object from stack
      - Setting Result (object.property = result)
    - **STACK Flag**
      - For branch