50.051 Group 4 Project Final Report

Team members

Student ID	Name	Contributions
1006934	Ho Atsadet	Interpreter frontend
1007107	Mohammad Saif Zia	Interpreter backend
1007150	James Bryan Budiono	Interpreter frontend
1007176	Tan Meng Teck	Interpreter backend

Github repository - https://github.com/mDSaifZia/interpreter

Table of Contents

Ratsnake v1.0

Get Started

Features	Example
Declare var	`var x = 5`
Assign var	`x = "hello"`
For loop	`loop i from(1,5){...}`
While loop	`while (condition) {...}`
Bitwise OR, AND, XOR, LSHIFT, RSHIFT	`|, &, ^, <<, >>`
Compare	`==, !=, >=, <=, >, <`
Logical	`||, &&, !`
Function definition	`fn f(args) {}`
Function call	`f(args)`
Comments	`// This is a comment`
Dynamic typing	`var x = "Hello" ; x = 5;`
Declare block	`{}`
Delimit the next instruction	`;`
Print	`print(value)`
Input	`var x = input(message)`

Type conversion	`int("5");  //converts to int`
	`float(5);  //converts to float`
	`str(5);  //converts to string`
	`bool("true");//converts to bool`

Example program in Ratsnake

Let's run through a simple Fibonacci program in Ratsnake using the above language features.

Create a *fib.rtsk* file.

```
// fib.rtsk

fn fib(n) {
      if (n <= 1) {
      return n;
      } else {
      return fib(n-2) + fib(n-1);
      }
}
var test = fib(5);
print("fib(5): ");
print(test);

// Expected output on screen -
// fib(5): 5
```

Compilation in Seven Easy Steps

1. Clone the repository - link to interpreter repository
2. Ensure necessary system packages have been installed - system packages
3. To compile the Ratsnake virtual machine on Linux and Windows, just run the following command on the terminal in the root directory of the project -

```
make
```

4. This will output a binary file *ratsnake.exe* for Windows or *ratsnake* for Linux
5. To run Ratsnake from anywhere within your terminal environment, add this binary file to your path. (optional)
6. Place the *fib.rtsk* in the same directory as the Ratsnake executable. Run the program using the following command and view the output in your terminal (stdout) -

```
> ./ratsnake fib.rtsk
fib(5):
5
```

Implementation of Ratsnake

Introduction

Ratsnake is a stack-based, dynamically typed programming language designed to explore virtual machine (VM) architecture, abstract syntax trees (ASTs), bytecode generation and interpretation. The Ratsnake was implemented in C and Python, featuring a dynamically resizable global variable table (hashmap-based), a stack for evaluation (Array-based), a function table (hashmap-based) for function lookups and finally a separate stackframe struct to manage function calls and local variables. This design allows for function recursion, lexical scoping, and strict stack-frame isolation.

The vm does not directly interpret and traverse the AST to run Ratsnake programs. Instead, programs written in Ratsnake are transpiled via an intermediate representation compiler (IR compiler) into a custom binary file (source_file.rtskbin). This binary features a 64-byte header that stores the location of function and class definitions, though the latter has not been implemented. The rest of the binary is reserved for function, class, and execution sections delineated by values from the header. It contains a one-to-one conversion of the bytecode into binary representation, where each opcode is translated directly into a unique 1-byte value.

Our language uses a “Primitive” object and an “Advanced” object struct pointer system. All primitive types—integers, floats, booleans, strings, and nulls—belong to a PrimitiveObject struct. This makes different data types all “inherit” from a base struct, allowing us to reference and manipulate them via a struct pointer, effectively allowing us to have dynamic typing. The PrimitiveObjects have “virtual methods” that take the form of function pointers, enabling operator overloading and flexible behaviour between types.

While minimalistic by design, its architecture is modular and open-ended, offering a sandbox for exploring more features such as garbage collection, custom opcodes, and even JIT compilation in future extensions.

Ratsnake virtual machine architecture (Backend)

Backend Overview

The backend of Ratsnake handles the actual execution of the bytecode produced by the frontend. It does this by first reading and then compiling a *.bytecode* file into a smaller and more streamlined *.rtskbin* file, which can then be read by the VM to process the instructions and perform the relevant actions.

The backend of Ratsnake is mainly composed of the following components:

- **IR_compiler** - compiles the *.bytecode* file into *.rtskbin* for VM execution
- **Virtual Machine (VM)** - Takes in *.rtskbin* file, reads and executes binary instructions
- **Hashmap** - A core data structure used as a global and function table
- **PrimitiveObjects** - The foundational datatype of all primitive types – int, str, float, NULL

IR_Compiler

The IR_Compiler serves as a crucial bridge between the frontend and backend of the Ratsnake VM architecture. It transforms the human-readable *bytecode*, generated by the frontend, into a binary format, *.rtskbin,* that can be executed by the VM.

File Handling and Processing Flow

The main function that drives this process is defined in IR_compiler.c -

```
int compile_ir(const char *input_path, const char *output_path);
```

1.) The compiler begins by opening the input file and output files using fopen() in binary read mode and write mode, respectively.
2.) Before processing the bytecode content, a placeholder header is written to reserve space at the beginning of the output file. The header is updated with the correct values after the processing is complete. *- the structure of the header is discussed in the Bytecode Generator section (pg 20)*
3.) The compiler then reads the *.bytecode* input line by line using a custom function for OS portability.

```
ssize_t portable_getline(char **lineptr, size_t *n, FILE *stream);
```

4.) For each line read, the compiler tokenises the input using strtok() and processes based on the token type -
    - Primitive values (INT, FLOAT, BOOL, STR)
    - Identifiers (ID)
    - Control flow instructions (OP_JMP, OP_JMPIF)
    - Opcodes (OP_ADD, OP_SUB, etc.)
5.) With the help of some helper functions, the compiler ensures proper binary encoding for writing different data types -

```
void write_uint8(FILE *f, uint8_t val) { fwrite(&val, 1, 1, f); }
void write_uint16(FILE *f, uint16_t val) { fwrite(&val, 2, 1, f); }
void write_int32(FILE *f, int32_t val) { fwrite(&val, 4, 1, f); }
void write_int64(FILE *f, int64_t val) { fwrite(&val, 8, 1, f); }
void write_double(FILE *f, double val) { fwrite(&val, 8, 1, f); }
```

6.) To convert the textual opcodes in the *.bytecode* file, the compiler uses a map_opcode() function. This function performs a series of string comparisons to map opcode names to the corresponding enum values defined in vm/vm.h -

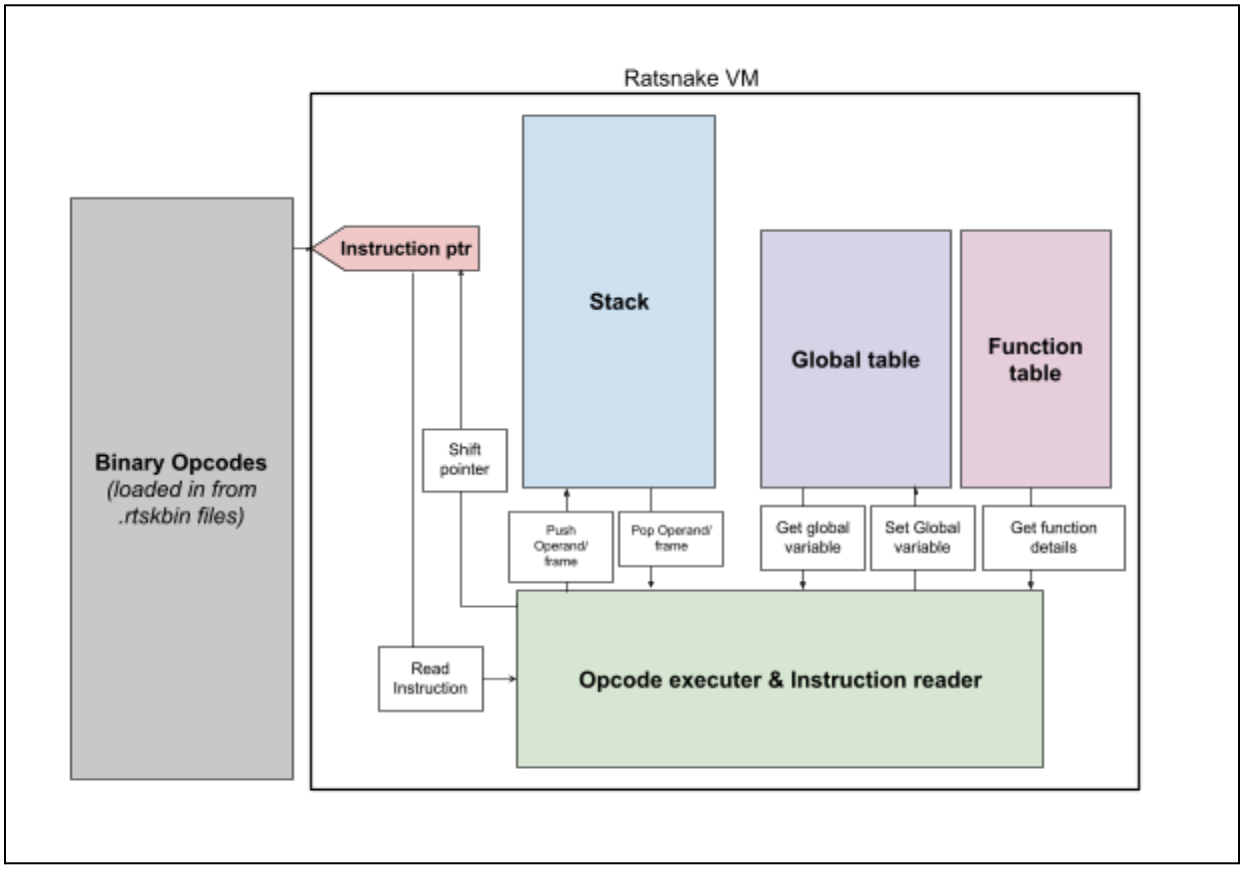
```
int map_opcode(const char *token);
```

7.) After tracking the beginning and ending of the function definition section, the compiler finally updates the header values after seeking back to the beginning of the output file.

Reading the bytecode input file line by line ensures a small memory footprint. At this stage, this binary file is ready to be executed by the VM, as seen in *ratsnake.c*.

Virtual Machine (VM)

Figure1: Below is a diagram visualising the high-level overview of the ratsnake vm.



The Ratsnake Virtual Machine (VM) is a stack-based, binary-bytecode-driven interpreter designed to execute *.rtskbin* files generated from source code. It simulates a runtime environment that supports variable storage, function execution and frames. The overall design is heavily inspired by the Beta-CPU design proposed to us in 50.002 Computational Structures.

The execution pipeline begins with binary opcodes loaded from a precompiled *.rtskbin* file. These opcodes are processed sequentially by the instruction reader, which decodes each byte and then executes. The opcode executor acts as the central control unit, interpreting each opcode and performing the respective actions based on its type:

- **Stack operations** (Push Operand, Pop Operand) manipulate the VM's internal evaluation stack, used for temporary value storage and expression evaluation.
- **Global table operations** (Get global variable, Set global variable) access or mutate global variables stored in a hashmap-like structure, maintaining persistent state across function calls.
- **Function table lookups** (Get function details) retrieve metadata (e.g., argument count, bytecode address) for function calls, enabling dynamic control flow and recursion.

**Note: We will not include or cover the use of a constant table here as it is not relevant to VM execution*

Binary Opcodes

Precompiled binary bytecode loaded in from *.rtskbin* files. Serves as the raw instruction stream for the VM to execute. Some of the opcodes listed take additional parameters and require more than 1 byte in length (OP_JMP, OP_JMPIF, INT, STR, FLOAT, BOOL, ID, LOCAL).

Table 1: Lists of opcodes and their corresponding 1 byte binary value written into .rtskbin files

OPCODE	Binary value	OPCODE	Binary value
OP_ADD	0b00000000	OP_BLSHIFT	0b00010101
OP_MUL	0b00000001	OP_BRSHIFT	0b00010110
OP_SUB	0b00000010	OP_BXOR	0b00010111
OP_DIV	0b00000011	OP_BOR	0b00011000
OP_GET_GLOBAL	0b00000100	OP_BAND	0b00011001
OP_SET_GLOBAL	0b00000101	OP_LOGICAL_AND	0b00011010
OP_CALL	0b00000110	OP_LOGICAL_OR	0b00011011
OP_RETURN	0b00000111	OP_LOGICAL_NOT	0b00011100
OP_HALT	0b00001000	OP_GET_LOCAL	0b00011101
OP_JMP	0b00001001	OP_SET_LOCAL	0b00011110
OP_JMPIF	0b00001010	LOCAL	0b00011111
INT	0b00001011	OP_PRINT	0b00100000
FLOAT	0b00001100	OP_INPUT	0b00100001
BOOL	0b00001101	OP_POP	0b00100010
STR	0b00001110	OP_MOD	0b00100011
NULL	0b00001111	OP_NEQ	0b00100100
ID	0b00010000	OP_EQ	0b00100101
OP_FUNCDEF	0b00010001	OP_GEQ	0b00100110
OP_ENDFUNC	0b00010010	OP_GT	0b00100111
OP_CLASSDEF	0b00010011	OP_LEQ	0b00101000
OP_ENDCLASS	0b00010100	OP_LT	0b00101001

Special Opcodes (OP_JMP, OP_JMPIF, INT, STR, FLOAT, BOOL, ID, LOCAL**)**

Table 2: List of special opcodes which take additional bytes for a single instruction

OPCODE	Byte layout	OPCODE	Byte layout
OP_JMP	[OP_JMP][4 bytes] (5 bytes)	STR	[STR][4 bytes len][len bytes] ((5 + len) bytes)
OP_JMPIF	[OP_JMPIF][4 bytes] (5 bytes)	FLOAT	[FLOAT][8 bytes] (9 bytes)
INT	[INT][8 bytes] (9 bytes)	BOOL	[BOOL][1 byte] (2 bytes)
ID	[ID][2 bytes len][len bytes] ((3 + len) bytes)	LOCAL	[ID][2 bytes len] (3 bytes)

Stack

Implemented as a fixed-size array with a pointer to the stack top to indicate the top of a stack and a base pointer to store the current scope of the execution. Functions with Last-In-First-Out (LIFO) behaviour where operands and frames are pushed and popped when needed. Stores primitives, identifiers and function frames in the StackEntry struct wrapper. OP_POP pops a single entry from the stack.

Implementation of the stack data structure

```
typedef struct Stack{
    size_t base_pointer; // base pointer of the stack
    size_t stack_top;    // stack top always points to free space on stack
    StackEntry stack[STACK_MAX];
} Stack;
```

Implementation of StackEntry

```
typedef struct StackEntry{
    void * value;
    StackEntryType entry_type;
} StackEntry;
```

The StackEntry struct takes a void pointer as its value, as it must be able to hold PrimitiveObjects, AdvancedObjects, FunctionFrames and Identifiers. An entry type is also stored to help decode the original pointer type when it is popped from the stack.

Global Table

A custom hashmap implemented in C that maps a char* string to values wrapped in a GlobalEntry struct wrapper. Used as persistent global-scope storage for data. OP_GET_GLOBAL retrieves a value from the stack, and OP_SET_GLOBAL modifies or creates an entry into the global table hashmap.

Implementation of the GlobalEntry struct

```
typedef struct GlobalEntry{
  void * value;
  StackEntryType entry_type;
} GlobalEntry;
```

As can be seen, the table entry structures are very similar to each other. However, we have chosen to separate them like this to ensure modularity. This prevents mutating specific structs should we need to add in additional attributes and features.

Function Table

A hashmap of function names (char*) to FunctionEntry structs, which stores:

- Function name
- Function definition's address
- Number of arguments
- Number of locals declared in the struct

Loaded at the beginning before the main execution loop occurs. Accessed during OP_CALL.

Implementation of the FunctionEntry struct

```
typedef struct {
    char *name;
    size_t func_body_address;
    int num_args;
    int local_count;
} FunctionEntry;
```

Opcode executor & instruction reader

Responsible for stepping through the bytecode stream. It interprets and increments through bytes based on predetermined opcode byte sizes. The opcode executor is the main switch case statement located in the run() function of vm.c. Dispatches behaviour based on opcode.

HashMap

As mentioned earlier, the Function and Global Tables are created using a custom dynamic hashmap data structure. It uses a chaining approach to handle collisions.

```
typedef struct Hashmap {
   HashmapEntry **table;
   size_t capacity;
   size_t length;
} Hashmap;
```

A new hashmap can be created using the init_hashmap() function. This function allocates memory for the hashmap, creates a table of pointers using calloc() to enforce NULL initialisation, initialises the length to 0 and sets the capacity -

```
Hashmap * init_hashmap(size_t capacity);
```

The hashmap uses the following hash function to convert string keys into table indices -

```
size_t hash(const char* str, size_t capacity) {
   size_t hash = 4123;
   while (*str) {
       hash = ((hash << 5) + hash) + (*str++);
   }
   return hash % capacity;
}
```

This function -

1. Starts with a prime number seed (4123)
2. For each character in the key string, update the hash using a variant of the djb2 algorithm
3. Takes the modulo of the hash with the table capacity to get an index within range

To set a value in the hashmap, the following function is used -

```
void hashmap_set(Hashmap * hashmap, const char * key, void * value, void
(*free_value)(void*));
```

This function -

1. Resizes the hashmap when the load factor exceeds 85% by calling -

```
void hashmap_resize(Hashmap * hashmap, void (*free_value)(void*));
```

2. Updates the existing value if the key is already present
3. Creates a new entry at the head of the collision chain if the key is new
4. Makes a copy of the key string using strdup() to ensure the hashmap owns the keys

To get a value from the hashmap -

```
void * hashmap_get(Hashmap * hashmap, const char * key);
```

The hashmap_get key computes the index by hashing the input key string and traverses the linked list, returning the value or NULL if the value is not found in the hashmap.

The dynamic resizing achieved by the hashmap_resize() function allows us to scale the table reasonably efficiently by doubling the hashmap capacity if more than 85% capacity is hit. This is done by transferring the rehashed entries into a bigger new table and then freeing the old table. Also, observe that the hashmap_resize() also takes in a function pointer to allow custom cleanup.

Thus, this hashmap implementation serves as an efficient data structure when storing and retrieving global variables and functions from their respective tables.

Primitive Objects

All values in Ratsnake, regardless of type, support the same interface, PrimitiveObject -

```
struct PrimitiveObject {
   PrimitiveType type;
   // void (*free)(PrimitiveObject* self); // all primitives except for
null must be freed
   VM* vm;
   BinaryOp add;
   BinaryOp mul;
   BinaryOp div;
   BinaryOp mod;
   CompareOp eq;
   CompareOp neq;
   CompareOp geq;
   CompareOp gt;
   CompareOp leq;
   CompareOp lt;
   DunderString __str__;
};
```

Every entity in ratsnake is formed using the above struct with the metadata, PrimitiveType to distinguish between the types. Along with this, a reference to the VM instance as well as function pointers for operators are also packed into the primitive objects.

We have implemented the following primitive types, which inherit from the base PrimitiveObject -

```
/* Integer object */
typedef struct int_Object {
   PrimitiveObject base;
   int64_t value;
   BinaryOp bwXOR;
   BinaryOp bwAND;
   BinaryOp bwOR;
   BinaryOp bwRSHIFT;
   BinaryOp bwLSHIFT;
} int_Object;
```

By placing the PrimitiveObject base as the first member, any pointer to a derived object can be safely cast to a PrimitiveObject* without any offset adjustments, regardless of the difference in data storage fields between the primitive types. This design allows for polymorphic handling of objects through the base type.

The function pointers (virtual methods) in the PrimitiveObject structure enable operator overloading. When constructors initialise a primitive object, they assign appropriate function implementations. During VM execution, operations like addition are performed by invoking these virtual methods -

// From vm.c - OP_ADD handling

```
PrimitiveObject *result =

     ((PrimitiveObject *)a.value)->add(((PrimitiveObject *)a.value),

                                       ((PrimitiveObject *)b.value));
```

This design allows for:

1. Type-specific operator implementations
2. Mixed-type operations (like adding an int to a float)
3. Runtime dispatch based on operand types

Type Conversion and Promotion

The virtual method system handles type conversions and promotions internally. For example, when adding an integer to a float:

1. The integer's **add**() method is called
2. It detects the float operand
3. It converts the integer to a float
4. It performs the floating-point addition
5. It returns a new float object

This mimics dynamic languages' type promotion rules without requiring explicit type checking in the VM's execution loop.

String Representation

Every primitive type implements a __str__ method that returns a dynamically allocated string representation -

```
char* int_to_string(PrimitiveObject* self);
```

```
char* float_to_string(PrimitiveObject* self);

char* bool_to_string(PrimitiveObject* self);

char* null_to_string(PrimitiveObject* self);

char* str_to_string(PrimitiveObject* self);
```

This enables consistent string conversion for printing and concatenation operations, similar to Python's **str()** function.

Ratsnake behaviour

Execution

The Ratsnake virtual machine operates as an inline interpreter, meaning that code is executed as it is encountered during the evaluation loop, rather than being fully validated beforehand. This execution model allows Ratsnake to attempt running any instruction that is syntactically and lexically valid, even if it is erroneous at runtime. In essence, Ratsnake behaves similarly to Python: runtime errors (such as invalid operations or type mismatches) are not caught during parsing but are instead raised during execution.

A core design philosophy of Ratsnake is its use of virtual methods for all operations. Instead of hardcoding specific behaviour for operators like (+, *, /, or ==, etc.), these operations are treated as method calls on the operand objects. This is implemented via a virtual method embedded within each PrimitiveObject or AdvancedObject. For example, when the OP_ADD instruction is encountered, the VM invokes the .add() “method” defined by the left-hand operand/ object.

This afford us a high degree of flexibility, which will be able to support AdvancedObjects to override or extend operator behavior. However, it also introduces a significant runtime overhead. Even basic operations like integer addition are no longer atomic; they require dynamic dispatch, type checking, and error handling by the method at runtime.

By deferring type enforcement and operation validity to runtime, Ratsnake supports more dynamic behaviour, duck typing, supports user-defined operator interactions and a much simpler parser. But this also makes the interpreter slower and more complex compared to statically-typed compiled languages.

On this note, the increase in overhead from our choice of implementation is precisely why we do not execute directly from the AST, *as mentioned in the introduction*. While AST interpretation is possible and would be a more direct choice of interpretation, it introduces even more runtime overhead due to the need for recursive tree traversal and higher memory usage. To avoid this, Ratsnake compiles source code into a compact binary bytecode format, which enables faster linear execution through a bytecode instruction stream.

Scoping (LEGB)

Ratsnake uses a simplified variation of the Local-Enclosing-Global-Builtin (LEGB) scoping model. However, as classes and advanced objects have not yet been implemented, our language currently only follows a BLG convention, omitting the enclosing scope. It should also be noted that we have specifically reordered built-in to be at the beginning, as we have decided, for simplicity and to enforce certain programming standards, built-in functions will be treated with the same precedence as keywords, and as such will always be parsed first. This prevents the user from naming local or global variables with the same name as built-ins and obscuring them.

Miscellaneous:
One distinction to make is that unlike languages that perform dynamic scoping using spaghetti stacks or fully enclosed frames. Ratsnake performs scope resolution during the parsing phase, based on context (e.g., inside or outside a function), the compiler then emits the appropriate opcode: OP_SET_LOCAL, OP_SET_GLOBAL, etc.

Hence, Ratsnake currently does not support nested functions with access to enclosing function variables, as it is unable to perform dynamic scoping. Inner functions are treated independently and cannot capture non-global variables from outer functions. Currently, our parser does not allow for the declaration of inner functions.

Built-In Scope (B):
While not explicitly structured as a separate namespace, Ratsnake includes built-in functionality via dedicated opcodes like OP_PRINT and OP_INPUT mapped to print() and input(), respectively. These instructions are always available to the VM and behave like built-in functions in other languages.

Local Scope (L):
During function execution, local variables are stored in a dedicated LocalEntry struct within the FunctionFrame struct, isolated from other functions. These variables include function arguments and any declared locals and are accessed using the OP_GET_LOCAL and OP_SET_LOCAL opcodes.

Global Scope (G):
Variables declared outside functions or explicitly set via OP_SET_GLOBAL go to the global table that persists across all execution contexts. This allows shared access and acts as the fallback for variable lookups not found in the local frame.

Functions in Ratsnake

As stated earlier, the functions are loaded into the function table before the VM runs the execution section of the bytecode. In our current compiler implementation, we don't perform any optimisations or garbage collection to ignore/clean any uncalled functions declared in the function section of the bytecode file. Hence, all functions declared are loaded into the function table, which slows down the compilation time and expends more memory due to redundant entries in the function table, which is a dynamic hashmap.

During the execution, when a function call is made via OP_CALL, a new entry of StackEntryType, *FUNCTION_FRAME* (a.k.a stackframe), is pushed onto the stack. This stackframe is created using our init_stackframe function in stackframe.c -

```
// Initialize a new stack frame
StackFrame *init_stack_frame(VM *vm, uint64_t *return_address, size_t
local_count);
```

Here is the structure of a stackframe -

```
typedef struct StackFrame {
 uint64_t *return_address; // stores the position of ip after op_call
 size_t parent_base_pointer;  //stores location of parent stackframe in
stack

 localEntry *locals[MAX_LOCALS]; // Local variable array

 size_t local_count; // Number of local variables
} StackFrame;
```

As can be seen above, our stackframe keeps track of all its local variables via an array of localEntry elements called **locals**, which are essentially an abstraction of StackEntry -

```
typedef struct { // Exactly the same as stackentry. Doing this to abstract
from stackentry.
 void *value;
 StackEntryType entry_type;
} localEntry;
```

How is the scoping enforced in our stackframe implementation?

At every function call in the execution section, a new stackframe is created, and the base pointer points to the most recently created (active) stackframe. While each stackframe contains a pointer to the parent stackframe to create a chain that preserves the call hierarchy, it is not able to reference the local variables from the parent stackframe. This helps create **Isolated Function Contexts**, *as mentioned on page 11, Scoping*.

Through these isolated contexts, we are able to localise the scoping of variables during recursion. However, this comes at the cost of other language features such as nested functions due to the **Stackframe Lifetime**. Stack frames are tied to function execution lifetime - they're created when a function is called and destroyed when it returns. Closures need access to variables even after the outer function has returned.

Note: Please refer to the Fibonacci Program Execution in the Appendix to help visualise the behaviour of functions during program execution in Ratsnake.

Ratsnake virtual machine architecture (Frontend)

Frontend Overview

The Ratsnake frontend is responsible for analysing, validating and processing the source code before passing it to the backend virtual machine (VM). It transforms raw text input from the source code file into a structured bytecode form that the VM can understand and execute. This involves recognising the program's syntax, validating its semantics, and converting it into compact bytecode instructions. The frontend plays an important role in connecting human-readable code with the low-level execution.

The compilation pipeline in the Ratsnake frontend consists of five components:

- **Lexer** - takes in the source code, tokenises it and outputs a list of the generated tokens
- **Parser** - takes in the list of tokens, parses it and outputs an Abstract Syntax Tree (AST)
- **AST Nodes** - provides the basic structure of all types of nodes in the AST
- **Semantic Checker** - takes in the AST structure and verifies its correctness
- **Bytecode Generator** - takes in the AST, traverses it and outputs a .bytecode file

Overall, the frontend supports all basic language features defined in Ratsnake: dynamic variables, expressions, conditionals, loops, functions, I/O, and a wide range of operators. Its modular design ensures that new features or syntactic rules can be added with minimal impact on existing components.

Lexer

The **custom_lexer.py** file defines a lexer, a component responsible for scanning and tokenising source code into discrete elements called tokens. Each token contains metadata: its type (integer, string, keyword, etc), value (the matched text), line number, and character position within the source code.

The lexer's main function, **tokenize()**, compiles all regex patterns into a single combined regular expression (regex), scanning the input source code and sequentially matching tokens.

For example:

- **123** becomes an INTEGER token
- **"Hello"** becomes a STRING token
- **loop** is recognised as a KEYWORD for "for loops" in our language

As tokens are identified, they are collected in a list, except for whitespaces and comments, which are explicitly ignored. The resulting list of tokens serves as input to subsequent compiler stages, such as the parser, to construct an abstract syntax tree (AST).

We designed the lexer using Python's built-in **re** module to define a set of regex for token matching. The list, **token_specification**, consisting of ordered tuples that explicitly define token types and their regex patterns, is used to produce an overarching regex. The order of these regex patterns is deliberately structured, placing multi-character tokens before single-character ones (e.g., `==` before `=`) to ensure correct matching. Comments and whitespace tokens are explicitly recognised but immediately discarded, simplifying downstream parsing tasks.

Additionally, the lexer maintains state such as line numbers, which facilitates precise error reporting and makes it easier for us to debug. This modular approach, where each token type and its pattern are explicitly defined, ensures clarity, maintainability, and ease of extension.

AST Nodes

The **custom_ast_nodes.py** file defines a comprehensive set of classes representing various nodes in an AST. These AST nodes are the fundamental building blocks used by the parser (**custom_parser.py**) to construct a structured, hierarchical representation of source code.

Each class corresponds to distinct syntactic elements of the language, such as:

- Statements - **VarDecl, LoopStmt, IfStmt, WhileStmt, ReturnStmt,**
- Expressions - **BinaryOp, UnaryOp, Literal, Identifier, CallExpr, Assignment**
- Built-in functions - **InputStmt, PrintStmt, ParseInt, ParseFloat, ParseStr, ParseBool**

When the parser analyses token streams generated by the lexer **(custom_lexer.py)**, it instantiates the respective AST node classes. These nodes form a tree that represents the entire program logically and structurally, and each branch breaks down into more specific parts.

Each node class inherits from a base class, **ASTNode**, and explicitly captures only the necessary components relevant to its structure. For example, the **IfStmt** node class captures only the condition, then-branch code block, and an optional else-branch code block, reflecting the exact grammar structure used by the parser. This explicit matching between AST nodes and grammar rules in the parser enhances clarity and ensures the parser does not carry extra responsibilities in processing the AST.

Parser

The **custom_parser.py** file implements a parser for transforming a list of tokens into an AST. The parser operates by iteratively consuming tokens produced by the lexer, categorising them into various specific node classes. It creates structured AST nodes corresponding to language constructs such as variable declarations, loops, conditional statements (**if, else, while**), function declarations, print statements, and assignments. It also includes specialised handling for built-in functions (**input(), int(), float(), str(), and bool()**).

For instance:

- **var x = 5;** becomes a **VarDecl** node containing a left child **Identifier** node for **"x"** and a right child **Literal** node for **"5"**
- **if (x > 0) { ... }** creates an **IfStmt** node containing a **BinaryOp** node condition for **"x>0"**

The parser contains different parsing functions built to handle the different types of tokens that it encounters. We implemented the recursive-descent parsing method, whereby each parsing function handles a specific precedence level and recursively calls upon other parsing functions that handle higher-precedence operations. This allows the parser to handle expressions and associativity operations accurately.

The output from the parser is an AST, structured using distinct AST node classes, with the root node being the **Program** node. This node contains a list of elements, whereby each element is a type of node class composed of other node classes, all together forming a full sentence that corresponds with a single statement in the source code, delimited by **";"**, **"{"** or **"}"**.

Additionally, since the parser is explicitly structured into distinct methods for each category of tokens (**parse_if_stmt, parse_loop_stmt, parse_var_decl, etc**), it makes the parser much more modular and easy to maintain. It also implicitly checks for errors, such as unexpected tokens or missing delimiters, via the use of the **consume()** method.

Semantic Analysis

The **custom_semantic_checker.py** file defines the semantic checker. It walks the Abstract Syntax Tree (AST) to enforce the logical rules of the Ratsnake language and ensures that syntactically valid code also behaves correctly. The semantic checker verifies multiple things, such as ensuring variables are defined before used and that function arguments are consistent with their definition. It ensures the program adheres to the language's rules even if it looks syntactically correct.

The main class, **SemanticChecker**, uses a visitor-based traversal method. For each AST node type, a corresponding **visit_** method is defined (e.g., **visit_FunctionDecl, visit_Assignment, visit_IfStmt**). These methods perform specific checks based on the node's structure and purpose. A helper function, **check()**, dispatches the correct **visit_** method dynamically based on the node's class name.

A separate **SymbolTable** class is used to manage scoping information throughout the program. This symbol table is implemented as a stack of dictionaries, allowing for nested scopes such as blocks or functions. When entering a new block, the checker calls **enter_scope()**, and calls **exit_scope()** when leaving it. For functions, it uses **enter_function_scope()** and **exit_function_scope()** to additionally track function parameters and mark the control context. Variables are defined using the **define()** method and retrieved using the **lookup()** method.

Some of the key checks performed include:

- Control variable protection in **visit_Assignment**, which raises an error if a loop variable is reassigned during execution.

- Unreachable code detection in **visit_Block**, which throws an error if any statements appear after a return in the same block.

Additional checks, such as function parameter validation, loop scoping, condition handling, and call argument verification, are implemented throughout the codebase in their respective **visit_** methods. The checker also includes basic type inference in expression visitors (**visit_Literal**, **visit_BinaryOp**, **visit_UnaryOp**).

Although Ratsnake supports dynamic typing, the semantic checker still applies basic type reasoning where needed, such as distinguishing between strings, integers, booleans, and floats in expressions or returns.

Bytecode Generator

The bytecode generator, **custom_bytecode_generator.py**, translates the validated AST into a list of bytecode instructions, like a low-level, compact form that the virtual machine can execute. Each AST node is visited and converted into one or more instructions that match the backend virtual machine's behaviour. For example, a **BinaryOp** becomes a series of **OP_ADD** or **OP_SUB** instructions, and a **WhileStmt** is turned into conditional jump commands with calculated offsets.

The bytecode generator is made to distinguish between global and local variables and also manage function boundaries. This results in a clean, executable stream of bytecode stored in a .bytecode file with clear separation of the main execution code from the function definitions, and also a 64-byte header that contains the positions of the various sections of the bytecode file.

Structure of BytecodeHeader when read in the backend:

```
typedef struct {
  size_t func_section_start;
  size_t func_section_end;
  size_t class_section_start;
  size_t class_section_end;
  size_t execution_section_start;
  uint8_t padding[24]
} BytecodeHeader;
```

In addition, we also decided that the traversing of nodes in the bytecode generation should be done using the Visitor Pattern. This allows it to separate the bytecode generation logic from the AST node structures, whereby each AST node type has a dedicated **visit_** method, such as **visit_Assignment** or **visit_IfStmt**, which will be called by the general **visit()** method. This approach simplifies extending the generator to support new language constructs, as adding a new node type only requires defining a corresponding **visit_** method.

Problems and limitations

Memory Leaks

To our knowledge, the current version of Ratsnake contains memory leaks, primarily because PrimitiveObject instances are not explicitly freed. This behaviour is intentional, as memory management is expected to be handled by a future garbage collector (GC). The GC will be responsible for identifying and freeing unreachable or stale objects. As of now, these leaks persist, but they are unlikely to pose a problem unless an excessive number of objects are created, such as in tight loops or recursive calls.

Some of these memory-related issues are partially mitigated through the use of the constant table. This is a pre-allocated array of commonly used values such as true, false, null, and integers in the range -512 to 512. When a new primitive value is requested, the VM first checks if the value exists in the constant table. If it does, the same reference is reused instead of allocating a new object. This design avoids creating redundant copies and helps reduce memory overhead.

Jump limits

In what can only be described as a poor design choice, the string primitive can hold up to 2^{32} bytes, which is approximately 4GB of string data. However, due to how jump instructions are encoded. OP_JMP and OP_JMPIF use a 32-bit signed offset, which limits their range to approximately ±2GB. This means that it is technically possible to define a single string so large that it **cannot be jumped over**, breaking the assumption that control flow and loop blocks can enclose any instruction.

The jump limit presents practical limits on how long these blocks can be before the VM fails. However, in practice, this should not be an issue as having a single code block longer than 2GB is exceedingly rare. A possible fix for this is to simply add in another opcode, “OP_GOTO”, which directly sets the instruction pointer to that location instead of jumping using an offset.

Is this 50.051 Compliant?

OS Compatibility - Our code compiles on both Windows and Linux. We are yet to test it on Mac.

File Processing - Our interpreter frontend parses source code from a *.rtsk* file and writes the IR/bytecode to the *.bytecode* file. Our IR_compiler reads this .bytecode file generated by the frontend and writes to a *.rtskbin* file. Lastly, this *.rtskbin* file is then run by the vm - *as seen in ratsnake.c in the root project directory.*

Parser - Our frontend contains a parser which is able to parse source code into bytecode and detect some invalid input syntax during the syntax analysis as well as meaningless input during the semantic analysis. *Refer to the frontend section for a detailed overview.*

Compiler flags - Due to our use of the C99-enabled dynamic arrays in the stackframe implementation, our code is not able to compile with the suggested flags in the project instructions. However, this allows for a cleaner and more compact codebase which is easier to maintain and extend.

State Machines - The regex used in our compiler Frontend uses state machines under the hood. Moreover, one could argue that our massive switch case in the run function in vm.c is a state machine since each opcode effectively serves as a state, which affects how we move around the instruction pointer to execute the bytecode instructions.

Documentation - We have a pretty exhaustive README, which contains our project description and usage guide for those who want to use and understand our interpreter. Moreover, our codebase is filled with comments to describe the behaviour of our functions and structs. Each .c file is accompanied with a .h header file which declares the structs and functions used.

Challenges and Future Improvements

The development of Ratsnake v1.0 has been an ambitious journey into language design and virtual machine implementation. Looking back at our implementation, we can identify several key challenges we faced and lessons learned, as well as exciting opportunities for future releases.

Implementation Challenges

Memory Management

One of the most significant challenges in Ratsnake's current implementation is memory management. *As noted on page 26*

> “To our knowledge, the current version of Ratsnake contains memory leaks, primarily because PrimitiveObject instances are not explicitly freed. This behaviour is intentional, as memory management is expected to be handled by a future garbage collector (GC).”

While our constant pool system mitigates some issues by caching commonly used values, a comprehensive solution requires a proper garbage collection system. Implementing GC involves complex algorithms for tracking object references and determining when objects are no longer reachable, which proved beyond the scope of our initial release.

Type System Limitations

Our PrimitiveObject system provides an elegant foundation for dynamic typing, but it comes with overhead:

1. Every value, even simple integers, requires full object allocation
2. Operations require virtual method dispatch through function pointers
3. Type conversions and promotions add runtime complexity

This approach prioritises consistency and simplicity over performance, which may limit Ratsnake's efficiency for computation-heavy applications.

Parser and Semantic Analysis Complexity

Creating a reliable parser and semantic analysis that can handle the full range of language constructs proved challenging. While our recursive descent parser correctly handles most cases, some edge cases in error reporting and recovery could be improved.

Lessons Learned

The Value of Intermediate Representations

The decision to use a bytecode intermediate representation between the frontend and backend proved invaluable. This clean separation allowed us to:

1. Develop the frontend and backend independently
2. Simplify the VM by focusing purely on execution rather than parsing
3. Create a more optimised binary format for execution

This architecture, inspired by established VMs like the JVM and CPython, significantly enhanced maintainability and allowed parallel development of components.

Object-Oriented Design in C

Implementing object-oriented concepts in C taught us valuable lessons about memory layout and polymorphism. Our struct-based inheritance model demonstrated that:

1. Careful attention to memory layout enables polymorphic behaviour
2. Function pointers can effectively simulate virtual methods
3. Type tags are essential for runtime type checking in a dynamic language

These techniques could be further refined in future versions to improve both performance and capability.

Future Directions for Ratsnake v2.0

Object-Oriented Programming Support

A natural evolution for Ratsnake would be to extend its primitive object system to support full object-oriented programming features:

Class System

The groundwork for classes already exists in our bytecode format, with some placeholders for class system opcodes and metadata in the bytecode header regarding the Classes section. Completing this implementation would involve:

1. Class definition syntax in the language grammar
2. Method and property declaration support
3. Instance creation and initialisation
4. Method lookup and dispatch mechanisms

Inheritance

Building on the class system, inheritance would require:

1. Syntax for specifying base classes
2. A mechanism for method overriding
3. Support for calling base class methods
4. Proper handling of the method resolution order

AdvancedObjects Integration

Our codebase already contains references to an **AdvancedObjects** system, which could be expanded to become the foundation for user-defined types. This would bridge the gap between our existing primitive types and a full object system.

Garbage Collection

Implementing a proper garbage collector would resolve the current memory management limitations. We could explore several approaches:

1. **Mark-and-Sweep GC**: A straightforward algorithm that marks all reachable objects and sweeps away unreachable ones
2. **Reference Counting**: Tracking object references and freeing objects when their count reaches zero
3. **Generational GC**: Optimising collection by focusing on recently created objects, which typically have shorter lifetimes

Performance Optimizations

Several optimisations could improve Ratsnake's performance:

1. **Just-In-Time Compilation**: Translating hot bytecode paths to native code
2. **Specialised Bytecode**: Adding specialised instructions for common operations
3. **Inline Caching**: Optimising method lookups for frequently accessed methods
4. **Loop Unrolling** and other optimisations inspired by the C/C++ compilers

Conclusion

Ratsnake v1.0 represents a solid foundation for a dynamic programming language with a clean VM architecture. The challenges we faced and lessons we learned have positioned us well for future development. With careful extension of our existing systems, Ratsnake v2.0 could evolve into a more powerful, efficient, and feature-rich language while maintaining the simplicity and elegance of the original design.

The journey from a stack-based VM with basic primitives to a full-featured language with object-oriented capabilities is challenging but achievable. By focusing on incremental improvements in key areas—memory management, the type system, and language features—we can continue to build on the strong architectural foundation established in v1.0.

Appendix

Fib.rtsk Bytecode

To view the fib.bytecode output of running ratsnake with our aforementioned fib.rtsk program, use the following command -

```
./ratsnake fib.rtsk -keep_ir
```

The keep_ir flag allows us to view the IR or the bytecode of the fib.rtsk program. Here is the fib.bytecode or the intermediate representation of fib.rtsk -

```
◆◆◆
```

```
IDFUNC 3 fib
OP_CALL
LOCAL 0
OP_GET_LOCAL
INT 1
OP_SUB
IDFUNC 3 fib
OP_CALL
OP_ADD
OP_RETURN
OP_JMP 0
__NULL__
OP_RETURN
OP_ENDFUNC // end of function section
```

Fibonacci Program Execution Stack Trace

Legend

- **functions:** function table containing all the FunctionEntries mapped by function name as key and FunctionEntry struct as value.
- **SF**: Stack Frame (The Stack sees this entry as FUNCTION_FRAME)
- **BP**: Base Pointer - points to the current active stack frame
- **SP**: Stack Pointer - points to next free location on stack (stack.top)
- **IP**: Instruction Pointer - points to current bytecode instruction
- **Locals**: Array containing the current BP's stackframe localEntries.
- **RA**: Return Address

Initial Program State (After loading the functions into the function table)
Stack: [] (empty)
SP: 0 (points to first position)
BP: 0 (no frames yet)
IP: 64 (INT 5)
Header: 0-63 (first 64 bytes in the bytecode)
functions: {"fib" : {name = "fib", func_body_address = 82, num_args = 1, local_count = 1}}

Program Execution

Preparing to call fib(5)
Stack: [5, "fib"]
SP: 2 (points after "fib")
BP: 0 (still in main program)
IP: 66 (OP_CALL instruction)

Call fib(5)
Stack: [SF:fib(5)]
SP: 1 (points after the stack frame)
BP: 0 (points to SF:fib(5))
IP: 96 (first instruction in fib function)
Locals in SF:fib(5): [{value = 5, entry_type = PRIMITIVE_OBJ}]
RA in SF:fib(5): 67 (ID 4 test instruction after OP_CALL)

Inside fib(5) - checking n <= 1 (false)
Stack: [SF:fib(5)]
SP: 1
BP: 0
IP: 105 (LOCAL 0 instruction - start of else branch)

Preparing to call fib(3) from fib(5)
Stack: [SF:fib(5), 3, "fib"]
SP: 3 (points after "fib")
BP: 0 (still points to SF:fib(5))
IP: 120 (OP_CALL instruction for first recursive call)

Call fib(3)
Stack: [SF:fib(5), SF:fib(3)]
SP: 2 (points after SF:fib(3))
BP: 1 (points to SF:fib(3))
IP: 96 (first instruction in fib function)
Locals in SF:fib(3): [{value = 3, entry_type = PRIMITIVE_OBJ}]
RA in SF:fib(3): 121 (instruction after first OP_CALL in fib(5))

Inside fib(3) - checking n <= 1 (false)
Stack: [SF:fib(5), SF:fib(3)]
SP: 2
BP: 1
IP: 105 (LOCAL 0 instruction - start of else branch)

Preparing to call fib(1) from fib(3)
Stack: [SF:fib(5), SF:fib(3), 1, "fib"]

SP: 4 (points after "fib")
BP: 1 (still points to SF:fib(3))
IP: 120 (OP_CALL instruction for first recursive call)

Call fib(1)
Stack: [SF:fib(5), SF:fib(3), SF:fib(1)]
SP: 3 (points after SF:fib(1))
BP: 2 (points to SF:fib(1))
IP: 96 (first instruction in fib function)
Locals in SF:fib(1): [{value = 1, entry_type = PRIMITIVE_OBJ}]
RA in SF:fib(1): 121 (instruction after first OP_CALL in fib(3))

Inside fib(1) - checking n <= 1 (true)
Stack: [SF:fib(5), SF:fib(3), SF:fib(1)]
SP: 3
BP: 2
IP: 99 (LOCAL 0 instruction - start of if branch)

fib(1) returns 1
Stack: [SF:fib(5), SF:fib(3), 1]
SP: 3 (points after return value)
BP: 1 (restored to point to SF:fib(3))
IP: 121 (instruction after first OP_CALL in fib(3))

Preparing to call fib(2) from fib(3)
Stack: [SF:fib(5), SF:fib(3), 1, 2, "fib"]
SP: 5 (points after "fib")
BP: 1 (still points to SF:fib(3))
IP: 131 (OP_CALL instruction for second recursive call)

Call fib(2)
Stack: [SF:fib(5), SF:fib(3), 1, SF:fib(2)]
SP: 4 (points after SF:fib(2))
BP: 3 (points to SF:fib(2))
IP: 96 (first instruction in fib function)
Locals in SF:fib(2): [{value = 2, entry_type = PRIMITIVE_OBJ}]
RA in SF:fib(2): 132 (instruction after second OP_CALL in fib(3))

Preparing to call fib(0) from fib(2)
Stack: [SF:fib(5), SF:fib(3), 1, SF:fib(2), 0, "fib"]
SP: 6 (points after "fib")
BP: 3 (still points to SF:fib(2))
IP: 120 (OP_CALL instruction for first recursive call)

Call fib(0)
Stack: [SF:fib(5), SF:fib(3), 1, SF:fib(2), SF:fib(0)]
SP: 5 (points after SF:fib(0))
BP: 4 (points to SF:fib(0))
IP: 96 (first instruction in fib function)
Locals in SF:fib(0): [{value = 0, entry_type = PRIMITIVE_OBJ}]
RA in SF:fib(0): 121 (instruction after first OP_CALL in fib(2))

fib(0) returns 0
Stack: [SF:fib(5), SF:fib(3), 1, SF:fib(2), 0]
SP: 5 (points after return value)
BP: 3 (restored to point to SF:fib(2))
IP: 121 (instruction after first OP_CALL in fib(2))

Preparing to call fib(1) from fib(2)
Stack: [SF:fib(5), SF:fib(3), 1, SF:fib(2), 0, 1, "fib"]
SP: 7 (points after "fib")
BP: 3 (still points to SF:fib(2))
IP: 131 (OP_CALL instruction for second recursive call)

Call fib(1) (second time)
Stack: [SF:fib(5), SF:fib(3), 1, SF:fib(2), 0, SF:fib(1)]
SP: 6 (points after SF:fib(1))
BP: 5 (points to SF:fib(1))
IP: 96 (first instruction in fib function)
Locals in SF:fib(1): [{value = 1, entry_type = PRIMITIVE_OBJ}]
RA in SF:fib(1): 132 (instruction after second OP_CALL in fib(2))

fib(1) returns 1 (second time)
Stack: [SF:fib(5), SF:fib(3), 1, SF:fib(2), 0, 1]
SP: 6 (points after return value)
BP: 3 (restored to point to SF:fib(2))
IP: 132 (instruction after second OP_CALL in fib(2))

fib(2) calculates 0 + 1 and returns 1
Stack: [SF:fib(5), SF:fib(3), 1, 1]
SP: 4 (points after return value)
BP: 1 (restored to point to SF:fib(3))
IP: 132 (instruction after second OP_CALL in fib(3))

fib(3) calculates 1 + 1 and returns 2
Stack: [SF:fib(5), 2]
SP: 2 (points after return value)
BP: 0 (restored to point to SF:fib(5))
IP: 121 (instruction after first OP_CALL in fib(5))

Preparing to call fib(4) from fib(5)
Stack: [SF:fib(5), 2, 4, "fib"]
SP: 4 (points after "fib")
BP: 0 (still points to SF:fib(5))
IP: 131 (OP_CALL instruction for second recursive call)

Call fib(4)
Stack: [SF:fib(5), 2, SF:fib(4)]
SP: 3 (points after SF:fib(4))
BP: 2 (points to SF:fib(4))
IP: 96 (first instruction in fib function)
Locals in SF:fib(4): [{value = 4, entry_type = PRIMITIVE_OBJ}]
RA in SF:fib(4): 132 (instruction after second OP_CALL in fib(5))

(After multiple recursive calls similar to above) fib(4) returns 3
Stack: [SF:fib(5), 2, 3]
SP: 3 (points after return value)
BP: 0 (restored to point to SF:fib(5))
IP: 132 (instruction after second OP_CALL in fib(5))

fib(5) calculates 2 + 3 and returns 5
Stack: [5]
SP: 1 (points after return value)
BP: 0 (back in main program context)
IP: 67 (ID 4 test instruction)

Main program continues with the returned value
Stack: [5, "test"]
SP: 2 (points after "test")
BP: 0
IP: 71 (OP_SET_GLOBAL instruction)

After OP_SET_GLOBAL (test = 5)
Stack: []
SP: 0 (empty stack)
BP: 0
IP: 72 (STR 8 "fib(5): " instruction)
Globals: {test: 5}

After pushing the string and printing
Stack: []
SP: 0
BP: 0
IP: 81 (ID 4 test instruction)
Output: "fib(5): 5"