Programming in Suny (version 1.0)

1. Introduction

Suny is a lightweight scripting language designed to be **small but powerful**, combining the minimalism and efficiency of **Lua** with the clarity and readability of **Python**.

Suny is intended to be a language that is:

- **Small** in implementation, so it is easy to understand and port.
- **Powerful** in expressiveness, so developers can solve real problems without feeling restricted.
- **Beginner-friendly**, encouraging experimentation and rapid learning.

1.1 Origins and Motivation

Suny was created as an experiment: *Can one person design and build a language from scratch that is both minimal and useful?*

Many modern languages are large, with complex standard libraries, heavy runtime systems, and thousands of pages of documentation. While this makes them powerful, it also makes them intimidating to beginners and difficult to fully understand.

Lua inspired Suny by showing that a small, elegant core can be extremely flexible. Python inspired Suny with its philosophy of readability and simplicity. Suny combines both ideas: a minimal implementation with a syntax that feels natural and beginner-friendly.

1.2 Philosophy of Suny

The design of Suny is guided by three principles:

1. **Simplicity**
    - The syntax is minimal, with few keywords.
    - Code should be as close to “pseudocode” as possible.
2. **Clarity**
    - Programs should be easy to read and write.
    - Indentation and structure should make logic clear at first glance.
3. **Power from smallness**
    - Instead of a large standard library, Suny focuses on a small but flexible core.
    - Advanced features can be built from simple building blocks.
    - The VM and bytecode are simple, so the language can be embedded or extended easily.

1.3 Typical Uses

Suny is not meant to replace large general-purpose languages like C++ or Java. Instead, it is designed for:

- **Learning programming concepts**: because the syntax is clean and the language is small, learners can quickly see how programming works.
- **Rapid experimentation**: with its interactive REPL, Suny encourages trial and error.
- **Scripting and automation**: Suny scripts can be written quickly to automate repetitive tasks.
- **Language research**: Suny itself is a good case study for building interpreters, compilers, and VMs.

1.4 A First Taste of Suny

Here is a simple Suny program:

```
print("Hello, Suny!")

i = 1
while i <= 5 do
    print("Count:", i)
    i = i + 1
end
```

This small example demonstrates Suny’s philosophy:

- Clean syntax (`while ... do ... end` is intuitive).
- No unnecessary boilerplate (no `main()` required).
- Immediate feedback in the REPL or script mode.

1.5 Implementation and Portability

Suny is written in **C**, which makes it:

- **Portable**: it can run on Windows, Linux, macOS, or even embedded devices.
- **Efficient**: compiled C code executes quickly with minimal overhead.
- **Compact**: the entire language implementation is small compared to larger interpreters.

The virtual machine (VM) of Suny executes bytecode instructions, similar to Lua or Python, but with a simplified design so it is easy to understand.

1.6 Vision for Suny

Suny will continue to evolve, but its vision will remain the same:

- Stay **small**: the core should remain lightweight and easy to understand.
- Stay **powerful**: expressive enough to build real applications.
- Stay **friendly**: a tool for both learners and curious developers who want to explore language design.

Like Lua, Suny will always value **elegance over complexity**. Like Python, it will always value **readability over clever tricks**.

In this way, Suny aims to be a language that is both a learning tool and a practical scripting solution: **a small language with big possibilities.**

2. Getting Started

Every programming language revolves around two fundamental concepts:

- **Input** – data provided to the program.
- **Output** – results produced by the program.

Suny follows the philosophy of being *small but powerful*: you can start writing programs immediately with very little setup, yet the language remains expressive enough for more complex projects.

2.1. Your First Program

The very first program most people write is a greeting message:

```
print("Hello, Suny!")
```

When executed, it produces:

```
Hello, Suny!
```

Explanation

- `print` is a **built-in function** that sends output to the screen.
- `"Hello, Suny!"` is a **string literal**, which means a fixed sequence of characters enclosed in double quotes.
- In Suny, strings can contain letters, numbers, punctuation, or even Unicode characters.

This simple example already shows Suny's core design principle: **clarity and simplicity**. With a single line of code, you can produce visible output.

2.2. More Printing Examples

The `print` function is versatile. Here are a few variations:

```
print(123)                 -- prints a number
print("Suny", "Language")   -- prints multiple values
print(10 + 20)             -- prints the result of an expression
print(nil)                 -- prints "nil"
```

Output:

```
123
Suny Language
30
nil
```

Notes:

- Unlike some languages, `print` in Suny automatically converts values to a string representation when displaying them.
- Multiple arguments are separated by a space.
- The special value `nil` represents “nothing” or “no value”.

2.3. Running Suny Programs

There are two main ways to run Suny code:

(a) Interactive Prompt (REPL)

The **REPL** (Read–Eval–Print Loop) lets you type commands one at a time and immediately see results.

To start the prompt, open a terminal and run:

```
suny -p
```

Example session:

```
PS C:\Suny> suny -p
Suny 1.0  Copyright (C) 2025-present, by dinhsonhai132
Type "exit()" or press Ctrl+C to quit.
>>> print("Suny is fun!")
Suny is fun!
>>> 5 * (2 + 3)
25
>>> exit()
PS C:\Suny>
```

The REPL is ideal for **learning**, quick experiments, or testing small pieces of code.

(b) Running from a File

For larger programs, it is more convenient to save your code into a file with the extension `.suny`.

Example: create a file called `main.suny` with the contents:

```
print("Welcome to Suny!")
```

Run it with:

```
PS C:\Suny> suny main.suny
Welcome to Suny!
PS C:\Suny>
```

This way, you can build reusable scripts and share them with others.

2.4. Command-Line Options

The `suny` command supports several options. To see them, type:

```
PS C:\Suny> suny -h
```

You will see:

```
Suny 1.0 Copyright (C) 2025-present, by dinhsonhai132
Usage: suny [options] [file]
Options:
  -c [file]   Compile the file
  -p          Run the prompt
  -h          Show this help
```

Explanation:

- `suny file.suny` → Runs the given program file.
- `suny -p` → Starts the interactive prompt (REPL).
- `suny -c file.suny` → Compiles the file (future versions may produce bytecode or executables).
- `suny -h` → Displays the help message.

2.5. Summary

At this point, you know how to:

- Write your first Suny program.
- Display text, numbers, and expressions using `print`.
- Run code interactively in the REPL.
- Execute code stored in `.suny` files.
- Use the command-line options for Suny.

Suny programs begin simple, but the language is designed to scale as you grow. In the following sections, we will cover **basic syntax, variables, data types, and control structures**, which together form the foundation of programming in Suny.

3. Simple Math and Operators

Suny supports a rich set of operators for performing **arithmetic calculations**, **comparisons**, and other common tasks. These operators form the foundation for writing expressions, which are evaluated to produce values.

3.1. Arithmetic Operators

Arithmetic operators allow you to perform basic mathematical calculations.

```
print(2 + 3)       # Addition: 5
print(5 - 2)       # Subtraction: 3
print(4 * 2)       # Multiplication: 8
print(10 / 2)      # Division: 5.0
print((1 + 2) * 3) # Parentheses control order: 9
```

Explanation:

- `+` adds numbers.
- `-` subtracts numbers.
- `*` multiplies numbers.
- `/` divides numbers and always produces a floating-point result (e.g., `5 / 2` → `2.5`).
- Parentheses `()` can be used to control precedence, just like in mathematics.

Integer Division and Remainder

In addition to normal division, Suny provides:

```
print(7 // 2)  # Integer division: 3
print(7 % 2)   # Modulo (remainder): 1
```

- `//` performs **floor division**, discarding the decimal part.
- `%` returns the **remainder** after division.

Exponentiation

You can raise numbers to a power with `**`:

```
print(2 ** 3)  # 2^3 = 8
print(9 ** 0.5) # Square root of 9 = 3.0
```

Notes on Arithmetic

- Division `/` always returns a floating-point number, even if the result is mathematically an integer.
- Using `//` guarantees an integer result (rounded down).
- Exponentiation works with both integers and floats.

3.2. Comparison Operators

Comparison operators compare two values and return a **Boolean result** (`true` or `false`).

```
print(3 < 5)    # true
print(5 > 3)    # true
print(2 == 2)   # true
print(2 <= 3)   # true
print(5 >= 5)   # true
print(10 != 5)  # true
```

Explanation:

- `<` → less than
- `>` → greater than
- `==` → equal to
- `<=` → less than or equal to
- `>=` → greater than or equal to
- `!=` → not equal to

Comparing Different Types

In Suny, comparisons generally make sense when values are of the same type. For example:

```
print("apple" == "apple")   # true
print("apple" == "banana")  # false
```

Comparing numbers and strings directly is not allowed and will raise an error:

```
print(5 == "5")  # Error: cannot compare number and string
```

This design keeps Suny simple and predictable.

3.3. Boolean Results

The results of comparisons are Boolean values: `true` or `false`.

```
a = (5 > 3)
print(a)    # true

b = (10 == 2)
print(b)    # false
```

These Boolean results are often used in **if statements** and **loops** (explained in later chapters).

3.4. Operator Precedence

When multiple operators appear in the same expression, Suny follows a **precedence order**:

1. Parentheses `()`
2. Exponentiation `**`
3. Multiplication, Division, Floor Division, Modulo `* / // %`
4. Addition and Subtraction `+ -`
5. Comparisons `< > <= >= == !=`

Example:

```
print(2 + 3 * 4)    # 14, because * has higher precedence
print((2 + 3) * 4)  # 20, because () overrides precedence
```

3.5. Summary

In this section, you learned:

- Suny supports **basic arithmetic operators**: `+ - * / // % **` .
- Division `/` always returns a float; use `//` for integer division.
- The modulo operator `%` returns the remainder.
- **Comparison operators** return Booleans (`true` / `false`) and include `<, >, ==, !=, <=, >=` .
- Operator precedence follows standard mathematical rules.

These operators are the building blocks for expressions. Combined with variables and control structures, they allow you to perform calculations, make decisions, and write meaningful programs.

---

4. Variables

In programming, **variables** are like containers that hold values. Instead of writing the same number or string over and over, you can store it in a variable and use the variable's name to refer to it. This makes your code shorter, clearer, and easier to change later.

In **Suny**, variables are simple and flexible. You don't have to declare their type (like `int` or `float` in C). Instead, Suny figures it out automatically when you assign a value.

4.1 Global Variables

A **global variable** is a variable created outside of any function. It can be accessed from anywhere in the program: inside functions, loops, or just at the top level.

This makes globals powerful, but also dangerous—if too many parts of your code can change the same variable, it's easy to introduce bugs.

Example in Suny:

```
a = 1
b = 2

print(a)  # prints 1
print(b)  # prints 2

b = b + 5
print(b)  # prints 7
```

Key Points:

- A global is "visible" everywhere in the program.
- Changing it inside a function changes it for everyone else too.
- This makes programs easier to write at first, but harder to maintain in the long run.

Comparison:

- **C/C++:** Globals must be declared outside of `main()` or any function, often with a type like `int x = 5;` .
- **Python:** Any variable defined at the top level of a file is global, but inside functions you must use `global x` if you want to modify it.
- **Lua:** All variables are global by default unless marked `local` .

In Suny, like Lua, variables are global unless you put them inside a function, where they become local.

4.2 Local Variables

A **local variable** exists only inside the function where you create it. Once the function finishes running, the variable disappears, and you cannot access it anymore.

This is useful because it prevents different parts of the program from interfering with each other.

Example in Suny:

```
function test() do
    x = 10
    print(x)  # prints 10
end

test()
print(x)  # error: x is not defined
```

Here, `x` is local to `test()`. Trying to use it outside gives an error.

Why Locals Are Better:

- **Safety:** No other part of your program can accidentally change them.
- **Clarity:** When reading the function, you know that the variable belongs only there.
- **Memory:** Locals only live while the function is running, so they free up memory afterward.

Good Practice:

Always prefer **local variables** unless you have a strong reason to use globals. Globals are best for values that truly belong to the whole program, such as configuration settings or constants.

4.3 Assignment

An **assignment** means giving a variable a new value.

In Suny, this is done with the `=` operator:

```
a = 5
b = "hello"
c = true
```

Once assigned, you can use the variable in calculations or other expressions.

Updating Variables

You can change a variable by reassigning it:

```
a = 0
a = a + 1   # now a is 1
a = a * 2   # now a is 2
```

Compound Assignment

Suny also supports shorthand operators:

```
a = 0
a += 1  # increase by 1
a -= 1  # decrease by 1
a *= 2  # multiply by 2
a /= 2  # divide by 2
```

These are equivalent to the longer forms:

- `a += 1` → `a = a + 1`
- `a -= 1` → `a = a - 1`
- `a *= 2` → `a = a * 2`
- `a /= 2` → `a = a / 2`

4.4 Example Program

Here's a full program that mixes globals, locals, and assignments:

```
score = 0   # global variable

function add_points(points) do
    local_bonus = 2      # local variable
    score += points + local_bonus
    print("Added", points, "points. Current score:", score)
end

add_points(5)   # Added 5 points. Current score: 7
add_points(3)   # Added 3 points. Current score: 12

print(score)    # 12
print(local_bonus)  # error: local_bonus is not defined
```

4.5 Summary

- **Variables** hold values like numbers, strings, or booleans.
- **Globals** can be used anywhere but may cause conflicts if overused.
- **Locals** are safer and should be preferred.
- **Assignments** let you change a variable's value, and compound assignments make it shorter.

Think of globals as "public" values and locals as "private" values. If you want your code to be clean, predictable, and bug-free—use locals as much as possible.

5. Data Types

Suny is **dynamically typed**, meaning you don't need to declare the type of a variable explicitly. The type of a variable is determined automatically at runtime based on the value you assign to it.

This makes Suny flexible and expressive, while still providing a consistent set of **core data types**.

The main built-in types in Suny are:

- **Boolean** (`true`, `false`)
- **Numbers** (integers and floats)
- **Strings** (text enclosed in quotes)
- **Lists** (ordered collections of items)
- **Functions** (first-class values, both named and anonymous)

5.1 Boolean

Booleans represent **truth values**: `true` or `false`.

```
is_sunny = true
is_raining = false

print(is_sunny)   # true
print(is_raining) # false
```

Boolean in Conditions

```
weather = "sunny"

if weather == "sunny" do
    print("Go outside!")
else
    print("Stay inside!")
end
```

Booleans are especially important in **control flow** (if/else, loops, etc.). They can also result from **comparison operators**:

```
print(5 > 3)    # true
print(5 < 3)    # false
print(5 == 5)   # true
print(5 != 5)   # false
```

Boolean Operators

Operator	Meaning	Example	Result
`and`	Logical AND	`true and false`	false
`or`	Logical OR	`true or false`	true
`not`	Logical NOT	`not true`	false

5.2 Numbers

Numbers are used for mathematics and calculations. Suny supports **integers** (whole numbers) and **floats** (decimal numbers).

```
a = 10       # integer
b = -5       # integer
c = 3.14     # float
d = -0.5     # float
```

Arithmetic Operators

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`5 - 3`	`2`
`*`	Multiplication	`5 * 3`	`15`
`/`	Division (float)	`5 / 2`	`2.5`
`//`	Floor division	`5 // 2`	`2`
`%`	Modulus (remainder)	`5 % 2`	`1`
`**`	Power (exponent)	`2 ** 3`	`8`

Examples

```
x = 10
y = 3

print(x + y)   # 13
print(x - y)   # 7
print(x * y)   # 30
print(x / y)   # 3.3333...
print(x // y)  # 3
print(x % y)   # 1
print(x ** y)  # 1000
```

Precedence and Parentheses

Parentheses can be used to control the order of evaluation:

```
print(2 + 3 * 4)     # 14
print((2 + 3) * 4)   # 20
```

5.3 Strings

Strings represent **text**. They are sequences of characters enclosed in **double quotes** `" "` or **single quotes** `' '`.

```
name = "Dinh Son Hai"
greeting = 'Hello, world!'

print(name)     # Dinh Son Hai
print(greeting) # Hello, world!
```

String Operations

```
first = "Hello"
second = "World"
combined = first + " " + second

print(combined)      # Hello World
print(size(combined)) # 11
```

Strings can be compared:

```
print("abc" == "abc")  # true
print("abc" != "def")  # true
```

Strings in Conditions

```
password = "1234"

if password == "1234" do
    print("Access granted")
else
    print("Access denied")
end
```

Escape Characters

Escape	Meaning	Example	Output
`\n`	Newline	`"Hello\nWorld"`	Hello World

Escape	Meaning	Example	Output
`\t`	Tab	`"Col1\tCol2"`	Col1 Col2
`\\`	Backslash	`"C:\\Path\\File"`	C:\Path\File
`\"`	Double quote	`"He said: \"Hi\""`	He said: "Hi"
`\'`	Single quote	`'It\'s sunny'`	It's sunny

5.4 Lists

Lists are **ordered collections** that can hold multiple items, even of different types.

```
numbers = [1, 2, 3, 4, 5]
names = ["Alice", "Bob", "Charlie"]
mixed = [1, "Two", true, 4.5]
```

Accessing and Modifying Items

```
print(numbers[0])  # 1
numbers[0] = 10
print(numbers[0])  # 10
```

Adding and Removing Items

```
push(numbers, 6)   # append
pop(numbers)       # remove last item
```

Length of a List

```
print(size(numbers))  # 5
```

Looping Over Lists

```
fruits = ["apple", "banana", "cherry"]

# Using index
for i in range(size(fruits)) do
    print(fruits[i])
end

# Using element directly
for fruit in fruits do
    print(fruit)
end
```

5.5 Functions

Functions in Suny are **first-class values**: They can be assigned to variables, passed as arguments, returned, and even created anonymously.

5.5.1 Basic Function

```
function add(a, b) do
    return a + b
end

print(add(1, 2))  # 3
```

5.5.2 Higher-Order Functions

Functions can take other functions as arguments:

```
function apply(func, x, y) do
    return func(x, y)
end

print(apply(add, 5, 7))  # 12
```

5.5.3 Inner Functions

Functions can be nested:

```
function foo() do
    function bar() do
        print("This is bar")
    end
    return bar()
end

foo()  # Output: This is bar
```

5.5.4 Anonymous Functions

```
a = 10

getA = function() do
    return a
end

print(getA())  # 10
```

You can also call them immediately:

```
print(function() do
    return 42
end())   # 42
```

5.5.5 Lambda Functions

A **lambda** is a short form of anonymous function:

```
let f(x) = x + 1
print(f(2))  # 3
```

5.5.6 Special Local Variable: `self`

Inside a function, the special variable `self` always refers to that **function itself**.

```
function foo() do
    return self
end

anon = function() do
    return self
end
```

This makes recursion and self-reference easy, even without naming the function.

5.5.7 Example: Closure

```
function foo() do
    count = 0
    return function() do
        count = count + 1
        return count
    end
end

a = foo()
for i in range(10) do
    print(a())
end
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

5.5 Summary

- Booleans → true/false
- Numbers → integers & floats with `+ - * / // % **`
- Strings → text with escape sequences
- Lists → ordered collections with indexing and iteration
- Functions → first-class values, support closures, anonymous functions, and lambdas

7. Control Structures

Control structures determine the **flow of execution** in a program. They allow you to make decisions, repeat actions, and handle complex logic.

Suny provides these main control structures:

- **Conditional Statements** (`if`, `elif`, `else`)
- **Loops** (`while`, `for`)
- **Special Controls** (`break`, `continue`)

7.1 Conditional Statements

Conditional statements let your program choose between different paths.

7.1.1 Basic `if`

```
score = 75

if score >= 50 do
    print("You passed!")
end
```

➡ If the condition is `true`, the code inside the block runs. ➡ If it is `false`, the block is skipped.

7.1.2 `if ... else`

```
score = 40

if score >= 50 do
    print("You passed!")
else
    print("Try again")
end
```

If the first condition fails, the `else` block executes.

7.1.3 `if ... elif ... else`

You can chain multiple conditions with `elif` (short for *else if*).

```
score = 85

if score >= 90 do
    print("Excellent")
elif score >= 75 do
    print("Good job")
elif score >= 50 do
    print("You passed")
else
    print("Failed")
end
```

 Conditions are checked from top to bottom. Only the **first matching block** is executed.

7.1.4 Nested Conditions

Conditions can be placed inside each other:

```
age = 20
has_id = true

if age >= 18 do
    if has_id do
        print("Access granted")
    else
        print("ID required")
    end
else
    print("Too young")
end
```

7.1.5 Comparison Operators Recap

Operator	Meaning	Example	Result
`==`	Equal	`5 == 5`	true
`!=`	Not equal	`5 != 3`	true
`>`	Greater than	`5 > 3`	true
`<`	Less than	`3 < 5`	true
`>=`	Greater than or equal to	`5 >= 5`	true
`<=`	Less than or equal to	`4 <= 5`	true

7.1.6 Boolean Logic in Conditions

```
temperature = 30
sunny = true

if temperature > 25 and sunny do
    print("Perfect beach day")
end

if temperature < 10 or not sunny do
    print("Maybe stay home")
end
```

7.2 Loops

Loops let you **repeat code** multiple times.

7.2.1 `while` Loop

Repeats while the condition is `true`.

```
count = 0

while count < 5 do
    print(count)
    count = count + 1
end
```

Output:

```
0
1
2
3
4
```

7.2.2 Infinite `while`

```
while true do
    print("Running forever...")
end
```

➡ You usually combine this with `break` to stop.

7.2.3 `for ... in range()`

Suny provides `range(n)` to generate numbers from `0` to `n-1`.

```
for i in range(5) do
    print(i)
end
```

Output:

```
0
1
2
3
4
```

You can also use `range(start, end)`:

```
for i in range(3, 7) do
    print(i)
end
# 3, 4, 5, 6
```

7.2.4 `for ... in list`

Iterating over items directly:

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits do
    print(fruit)
end
```

Output:

```
apple
banana
cherry
```

7.2.5 Nested Loops

```
for i in range(3) do
    for j in range(2) do
        print("i =", i, ", j =", j)
    end
end
```

7.3 Loop Control: `break` and `continue`

7.3.1 `break`

Stops the loop immediately.

```
for i in range(10) do
    if i == 5 do
        break
    end
    print(i)
end
```

Output: `0 1 2 3 4`

7.3.2 `continue`

Skips to the next iteration.

```
for i in range(5) do
    if i == 2 do
        continue
    end
    print(i)
end
```

Output: `0 1 3 4`

7.4 Combining Control Structures

Complex programs often use **if statements inside loops**:

```
for i in range(1, 11) do
    if i % 2 == 0 do
        print(i, "is even")
    else
        print(i, "is odd")
    end
end
```

7.5 Summary

- `if`, `elif`, `else` → decision making
- `while` → repeat while condition is true
- `for` → iterate over ranges or collections
- `break` → exit loop early
- `continue` → skip current iteration

Control structures are the **backbone of logic** in Suny. They allow you to model real-world decisions, repeat tasks, and build dynamic programs.

---

8. Include

The **`include`** statement in Suny allows you to organize your program across multiple files or folders. Instead of writing everything in a single file, you can split your code into smaller parts (modules, configs, helpers) and bring them together when needed.

When Suny sees an `include`, it **inserts the code from that file or folder directly into the current file** before running the program. This makes it behave almost the same as if you had copy-pasted the contents manually, but in a more organized way.

8.1 Including a File

If the target is a **file**, Suny copies all its contents.

```
# config.suny
pi = 3.14
```

```
# main.suny
include "config"

print(pi)   # 3.14
```

Here, the variable `pi` becomes part of `main.suny`'s scope, as if it was defined inside it.

8.2 Including a Folder

If the target is a **folder**, Suny automatically looks for a file named `main.suny` inside that folder.

```
math/
 ├── main.suny
 └── extra.suny
```

```
# math/main.suny
square(x) = x * x
```

```
# main.suny
include "math"

print(square(5))   # 25
```

Suny only loads `math/main.suny` by default. If you need extra files (`extra.suny` in this example), you must include them explicitly:

```
include "math/extra"
```

8.3 Search Paths

Suny resolves includes in this order:

1. **Current directory** – looks for a file or folder in the same place as the current `.suny` file.
2. **Library directory** – if not found locally, Suny searches in the global library folder:

   ```
   C:/Suny/libs
   ```

   Example:

   ```
   C:/Suny/libs/
     └── std/
         └── main.suny
   ```

   ```
   include "std"
   ```

   This would load the standard library `std/main.suny`.
3. **Error** – if nothing is found, Suny throws a compilation error.

8.4 Error Cases

- **Missing file or folder**

  ```
  include "not_found"
  ```

```
Error: include target 'not_found' not found
```

- **Folder without main.suny**

```
mylib/
  └── helper.suny
```

```
include "mylib"
```

```
Error: no main.suny in folder 'mylib'
```

- **Name conflicts**

```
# a.suny
x = 10

# b.suny
x = 20
```

```
include "a"
include "b"

print(x)   # which one? result depends on last include
```

Best practice: avoid re-using the same global variable names across includes.

8.5 Summary

- Use `include` for **constants, small configs, or utility functions**.
- Keep each folder/module self-contained with its own `main.suny`.
- Avoid circular includes (`A` includes `B`, and `B` includes `A`).
- Use unique variable/function names to prevent conflicts.

---

9. C-API And Inside Suny

In **Suny**, you can directly call C functions. However, if you follow the **Suny rules**, you can integrate C into Suny in a much more powerful way. Suny provides a special macro called `SUNY_API`. This macro allows you to "embed" C code directly into the Suny environment, making C functions callable as if they were native Suny functions.

Using `SUNY_API` has several advantages:

1. **Enhanced extensibility**: You can leverage the full power of C to perform system-level operations or complex algorithms that Suny does not natively support.
2. **Seamless integration with Suny**: C functions wrapped with `SUNY_API` behave like normal Suny functions, keeping your code clean and maintainable.
3. **Reach Lua- or Python-level capabilities**: When used correctly, you can build complex modules, manipulate memory directly, or call external libraries while still enjoying Suny's simplicity.

In short, the **Suny C-API** acts as a powerful bridge between the Suny language and the real power of C, making your language flexible and much more capable.

Here's an extended, detailed version of **9.1 – The Stack Frame**, written in English in a "Lua-style" reference manual format, with clear explanations and examples:

9.1 The Stack Frame

The **stack frame** in Suny is the core runtime structure that manages function calls, local and global variables, constants, and the execution stack. Every function call in Suny, including calls to C-API functions, creates a new stack frame, allowing Suny to maintain execution context, variable scopes, and intermediate computation results.

A stack frame is represented by the `Sframe` structure, which contains the following key components:

9.1.1 Components of a Stack Frame

1. **Execution Context**
    - `f_code`: Points to the bytecode (`Scode`) associated with the current function or script.
    - `f_code_index`: Tracks the current instruction within the bytecode sequence.
    - `f_back`: Points to the previous stack frame, enabling call stack traversal during function returns or exceptions.

2. **Local and Global Variables**
    - `f_locals` and `f_globals` store pointers to `Sobj` objects representing local and global variables.
    - `f_locals_index` and `f_globals_index` track the number of currently active variables.
    - `f_locals_size` and `f_globals_size` track the allocated capacity of the respective arrays, allowing dynamic resizing when new variables are added.
3. **Execution Stack**
    - `f_stack` holds the intermediate values during computation, such as temporary results or function arguments.
    - `f_stack_index` indicates the top of the stack.
    - `f_stack_size` tracks the allocated capacity for dynamic resizing.
    - Operations like `Sframe_push` and `Sframe_pop` manipulate this stack to manage runtime values.
4. **Constants Pool**
    - `f_const` is an array holding constants defined within the current code context (e.g., numbers, strings, booleans).
    - `f_const_index` tracks the next available slot in the constants pool.
5. **C-API Integration**
    - The stack frame allows embedding C functions as Suny objects via `Sframe_load_c_api_func`.
    - Functions registered through the C-API can be called like any Suny function, while arguments and return values are managed on the stack.
6. **Garbage Collection**
    - `gc_pool` points to a `Garbage_pool` structure, enabling reference counting and memory management for dynamically allocated objects.
    - Each object pushed onto the stack or stored in a frame is reference-counted to prevent memory leaks.
7. **Label Map**
    - `f_label_map` stores mappings from label names to bytecode offsets, which is essential for control flow operations like `goto` or loops.

9.1.2 Creating a Stack Frame

A stack frame is typically created using `Sframe_new()`:

```
struct Sframe* frame = Sframe_new();
```

This function performs the following:

- Allocates memory for the frame itself and initializes all internal arrays.
- Sets all indices to zero.
- Initializes the stack, local, global, and constant arrays with default capacity (`MAX_FRAME_SIZE`).
- Prepares the frame to be initialized with a Suny code object using `Sframe_init()`.

9.1.3 Initializing a Stack Frame

Once created, a stack frame must be initialized with the code to execute:

```
Sframe_init(frame, code);
```

Here, `code` is a pointer to an `Scode` object representing the compiled Suny bytecode. Initialization includes:

- Setting `f_code` to the provided code object.
- Resetting the instruction pointer (`f_code_index`) to zero.
- Creating a label map (`f_label_map`) for fast lookup of control flow labels.

9.1.4 Stack Operations

The execution stack supports standard push and pop operations:

```
Sframe_push(frame, obj); // Push an object onto the stack
struct Sobj* top = Sframe_pop(frame); // Pop the top object
struct Sobj* back = Sframe_back(frame); // Peek the top object without popping
```

- **Push**: Adds an object to the top of the stack. If the stack exceeds its capacity, it is automatically resized.
- **Pop**: Removes and returns the object from the top of the stack. Proper reference counting ensures memory safety.
- **Back**: Returns the top object without removing it, useful for peek operations in expressions and evaluations.

9.1.5 Local and Global Variable Management

Stack frames manage variable scopes efficiently:

- **Local Variables**:

  ```
  Sframe_store_local(frame, address, obj, LOCAL_OBJ);
  struct Sobj* value = Sframe_load_local(frame, address);
  ```

  Local variables are stored in `f_locals` and are scoped to the current frame.

- **Global Variables**:

  ```
  Sframe_store_global(frame, address, obj, GLOBAL_OBJ);
  struct Sobj* value = Sframe_load_global(frame, address);
  ```

  Global variables persist across frames and scripts, stored in `f_globals`.

- **Constants**:

  ```
  Sframe_store_const(frame, obj);
  struct Sobj* c = Sframe_load_const(frame, address);
  ```

  Constants are immutable and optimized for repeated use during execution.

9.1.6 Integration with C-API Functions

Suny allows embedding C functions directly as Suny objects:

```
struct Sobj* func_obj = Sframe_load_c_api_func(frame, my_c_function, address, "my_func", args_count);
Sframe_call_c_api_func(frame, func_obj->c_api_func->func);
```

- Functions are registered globally and accessible as Suny objects.
- Arguments are passed via the stack, maintaining uniformity with Suny function calls.
- Return values are pushed onto the stack automatically.

9.1.7 Garbage Collection and Reference Counting

Every object stored in a frame—stack, local, global, or constant—is reference-counted:

- `Sgc_inc_ref(obj)` increases the reference count.
- `Sgc_dec_ref(obj, frame->gc_pool)` decreases it and triggers garbage collection when the count reaches zero.
- This ensures deterministic memory management while supporting dynamic object lifetimes.

This structured and modular design of the **Suny stack frame** enables:

- Nested function calls with independent execution contexts.
- Efficient variable and constant management.
- Seamless integration with C functions.
- Safe memory management via reference counting and garbage collection.

9.2 Basic Suny C Functions

Suny's **basic API functions** serve as the core building blocks for C integration. They provide direct access to the **execution stack**, **frame-local variables**, **global variables**, and **constants**, allowing C functions to interact naturally with the Suny environment.

By using these functions, C code can:

1. **Manipulate the stack safely** – push and pop objects while automatically managing reference counts.
2. **Access variables consistently** – read and write local and global variables without bypassing Suny's memory management rules.
3. **Create objects conveniently** – construct numbers, strings, and booleans that are fully compatible with Suny's runtime.
4. **Call and register C functions** – make C code callable as if it were native Suny code, enabling modular and extensible scripting.
5. **Ensure memory safety** – reference counting and garbage collection prevent memory leaks and dangling pointers.

Together, these functions form a **robust bridge** between Suny and C, enabling the creation of complex modules, libraries, and system-level integrations while preserving the simplicity and elegance of Suny's scripting model.

In Suny there are ... groups, each group support its own C function to work with it, for example: `Slist` group support **push**, **pop**, **free** function to work with Suny list datatype like `Slist_add`, `Slist_pop`, `Slist_free`, `Slist_change_item`, `Slist_get`, ... Or like `Sfunc` group we have `Sfunc_ready`, `Sfunc_set`, `Sfunc_set_func`, `Sfunc_obj_new`, `Sfunc_free`

9.2.1 `Sobj` functions

Here there are ... groups and how to use its function, start with `Sobj` group

`Sobj` group return `struct Sobj*` is the most important group in Suny, with `Sobj` we can creat Suny object like list object and function object...

Sobj_new

The function:

```
struct Sobj* Sobj_new(void);
```

creat a new `Sobj` object, it can be free, `Sobj_new` set object as `NULL_OBJ` type

---

Sobj_set_int

The function:

```
struct Sobj* Sobj_set_int(float value);
```

creates a **number object** with the specified float value. In Suny, all numeric objects are stored as floats.

You can perform arithmetic operations on number objects using the **`Seval`** group functions, such as `Seval_add()` :

```
struct Sobj* o1 = Sobj_set_int(1.0);
struct Sobj* o2 = Sobj_set_int(2.0);
struct Sobj* oadd = Seval_add(o1, o2);

Sio_write(oadd);  // Output: 3

Sobj_free(o1);
Sobj_free(o2);
Sobj_free(oadd);
```

- **`Sobj_free(obj)`** : Frees the memory associated with an object when it is no longer needed.
- **`Sio_write(obj)`** : Prints the object to the screen in a human-readable format. Example:

```
prompt>
3
prompt>
```

Sobj_get_obj

The function:

```
void* Sobj_get_obj(struct Sobj* obj, enum Sobj_t type);
```

returns the **main value** of an object. This is useful for extracting the underlying data of complex objects, such as functions or lists.

Example:

```
struct Sfunc* func = (struct Sfunc*) Sobj_get_obj(obj, FUNC_OBJ);
```

- The `type` parameter indicates the expected type of the object (e.g., `FUNC_OBJ`).
- This function safely retrieves the underlying value while preserving type information.

Suny Main Object Types

Suny defines the following primary object types:

Type Flag	Description

Type Flag	Description
`NUMBER_OBJ`	Numeric object (float)
`STRING_OBJ`	String object
`LIST_OBJ`	List object
`FUNC_OBJ`	Function object
`LOCAL_OBJ`	Local variable
`GLOBAL_OBJ`	Global variable
`BUILTIN_OBJ`	Built-in C function
`USER_DATA_OBJ`	User-defined data object
`CLASS_OBJ`	Class object
`INSTANCE_OBJ`	Instance of a class
`TRUE_OBJ`	Boolean true
`FALSE_OBJ`	Boolean false
`NULL_OBJ`	Null object

Sobj_make_bool

The function:

```
struct Sobj* Sobj_make_bool(int value);
```

This function creates a new boolean object.

- If the input value is `1`, the object's type is set to `TRUE_OBJ`.
- If the input value is `0`, the object's type is set to `FALSE_OBJ`.

This ensures that boolean values are consistently represented as distinct object types within the system.

Sobj_make_class

The function:

```
struct Sobj* Sobj_make_class(struct Sclass* sclass);
```

This will creat a class object which type is `CLASS_OBJ` the number value of this object is `0`

Sobj_set_func

The function:

```
struct Sobj* Sobj_set_func(struct Sfunc *func);
```

This function creates a new function object. The object's type is set to `FUNC_OBJ`, and its numeric field is initialized to `0`.

This ensures that newly created function objects have a well-defined default state before being assigned additional properties such as parameters, bytecode, or closure information.

Sobj_make_str

The function:

```
struct Sobj *Sobj_make_str(char* str, int size)
```

This function creates a new string object. The string object contains two fields:

- `char* string` — a pointer to the raw character array (null-terminated C string).
- `int size` — the length of the string, excluding the null terminator.

This design allows efficient access to both the raw data and the precomputed size of the string.

Sobj_make_str_obj

The function:

```
struct Sobj* Sobj_make_str_obj(struct Sstr *str)
```

This will creat a string object if you already created `struct Sstr*` object first

---

Sobj_make_char

The function:

```
struct Sobj* Sobj_make_char(char chr);
```

This function creates a new character object. The object's type is set to `CHAR_OBJ`, and its internal value is initialized with the given character `chr`.

This allows characters to be represented as distinct objects within the system, making them consistent with other primitive types such as integers, floats, and booleans.

Sobj_deep_copy

The function:

```
struct Sobj* Sobj_deep_copy(struct Sobj* obj);
```

This function creates a new object that is a deep copy of the input. Unlike a shallow copy, which only duplicates the outer structure while still sharing references to nested objects, a deep copy recursively duplicates **all levels** of the object's data.

This means that:

- For primitive types (e.g., integers, floats, booleans, and strings), the value is copied directly into the new object.
- For composite types (e.g., lists, dictionaries, or other containers), each nested element is itself deep-copied, producing a fully independent structure.

As a result, **any modification to the deep copy—whether to its top-level fields or to its nested elements—will not affect the original object**, and vice versa. This makes `Sobj_deep_copy` particularly useful when you need to work with independent object states without the risk of unintended side effects from shared references.

Sobj_shallow_copy

The function:

```
struct Sobj* Sobj_shallow_copy(struct Sobj* obj);
```

Creates a **shallow copy** of the given object. Primitive values (int, float, bool, string) are duplicated directly. For composite types (list, dict, etc.), only the outer container is copied, while nested elements still reference the same objects as the original.

Note

- Changing the container in the copy does not affect the original.
- Changing a shared nested element will affect both.

Sobj_make_list

The function:

```
struct Sobj* Sobj_make_list(struct Slist* list);
```

This function creates a new list object. The object's type is set to `LIST_OBJ`, and its numeric field is initialized to `0`. The provided `struct Slist*` is attached to the object, allowing it to manage a sequence of elements.

This design makes it possible to wrap an existing list structure inside an `Sobj`, ensuring consistent handling of list values throughout the system.

9.2.2 `Seval` Functions

The **`Seval`** group provides functions for evaluating basic expressions and operators. These functions implement fundamental operations across different object types, such as:

- Arithmetic: `+`, `-`, `*`, `/`
- Comparison: `<`, `>`, `<=`, `>=`, `==`, `!=`
- Logical: `&&`, `||`, `!`

By centralizing the evaluation logic, `Seval` ensures that operators behave consistently across the system, regardless of whether the operands are integers, floats, booleans, or other supported types.

Seval_add

```
struct Sobj* Seval_add(struct Sobj *obj1, struct Sobj *obj2);
```

Description Implements the **addition operator (`+`)**.

Rules

1. **Numbers**
    - Performs floating-point addition.
    - Both integers and floats are promoted to float internally.
    - Example: `5 + 3.2 → 8.2`.
2. **Strings**
    - Performs concatenation.
    - Example: `"foo" + "bar" → "foobar"`.
3. **Lists**
    - Concatenates two lists into a new list.
    - Elements are shallow-copied references.
    - Example: `[1,2] + [3] → [1,2,3]`.
4. **User-defined objects**
    - If the left-hand object defines a metamethod `mm_add`, it is invoked.
    - Otherwise, a runtime error is raised.

Errors

- If operand types do not match any rule, raises `TypeError: unsupported operand types for +`.

Seval_sub

```
struct Sobj* Seval_sub(struct Sobj *obj1, struct Sobj *obj2);
```

Description Implements the **subtraction operator (`-`)**.

Rules

1. **Numbers**
    - Performs floating-point subtraction.
    - Example: `10 - 4 → 6.0`.
2. **User-defined objects**
    - Invokes `mm_sub` if defined on the left-hand operand.

Errors

- Other operand types are invalid.
- Example: `"foo" - "bar"` → `TypeError`.

Seval_mul

```
struct Sobj* Seval_mul(struct Sobj *obj1, struct Sobj *obj2);
```

Description Implements the **multiplication operator (`*`)**.

Rules

1. **Numbers**
    - Performs floating-point multiplication.
    - Example: `6 * 7 → 42.0`.
2. **String × Integer**
    - Repeats the string `n` times.
    - Example: `"ha" * 3 → "hahaha"`.
    - If `n ≤ 0`, returns an empty string.
3. **List × Integer**
    - Repeats the list `n` times.
    - Example: `[1,2] * 2 → [1,2,1,2]`.
    - Elements are shallow-copied references.
4. **User-defined objects**
    - Invokes `mm_mul` if defined.

Errors

- Any other type combination results in `TypeError`.

Seval_div

```
struct Sobj* Seval_div(struct Sobj *obj1, struct Sobj *obj2);
```

Description Implements the **division operator (`/`)**.

Rules

1. **Numbers**
    - Performs floating-point division.
    - Example: `7 / 2 → 3.5`.
2. **Division by zero**
    - Triggers a runtime error: `ZeroDivisionError: division by zero`.
3. **User-defined objects**
    - Invokes `mm_div` if defined.

Errors

- Invalid operand types raise `TypeError`.
- Example: `"foo" / 2` → `TypeError`.

Seval_bigger

```
struct Sobj* Seval_bigger(struct Sobj *obj1, struct Sobj *obj2);
```

Description Implements the **greater-than operator (`>`)**.

Rules

1. **Numbers**
    - Numerical comparison.
    - Example: `5 > 3 → TRUE_OBJ`.
2. **Strings**
    - Lexicographical (dictionary order) comparison.
    - Example: `"zoo" > "apple" → TRUE_OBJ`.
3. **Lists**
    - Lexicographical element-wise comparison.
    - Example: `[2,1] > [2,0] → TRUE_OBJ`.
4. **User-defined objects**
    - Invokes `mm_gt` if defined.

Returns

- Always returns a boolean object (`TRUE_OBJ` or `FALSE_OBJ`).

Seval_smaller

```
struct Sobj* Seval_smaller(struct Sobj *obj1, struct Sobj *obj2);
```

Description Implements the **less-than operator (`<`)**.

Rules

- Same as `Seval_bigger` , but reversed.
- Example: `2 < 5 → TRUE_OBJ` .
- Example: `"ant" < "dog" → TRUE_OBJ` .

Returns

- A boolean object.

Seval_equal

```
struct Sobj* Seval_equal(struct Sobj *obj1, struct Sobj *obj2);
```

Description Implements the **equality operator (`==`)**.

Rules

1. **Primitive types (numbers, booleans, chars)**
    - Compared directly by value.
2. **Strings**
    - Deep comparison of all characters.
3. **Lists**
    - Deep equality: length must match, and all elements must be equal.
4. **User-defined objects**
    - Invokes `mm_eq` if defined.

Returns

- Boolean (`TRUE_OBJ` or `FALSE_OBJ`).

Seval_not_equal

```
struct Sobj* Seval_not_equal(struct Sobj *obj1, struct Sobj *obj2);
```

Description Implements the **inequality operator (`!=`)**.

Rules

- Defined as the logical negation of `Seval_equal` .
- Example: `5 != 3 → TRUE_OBJ` .
- Example: `"foo" != "foo" → FALSE_OBJ` .

Seval_bigger_and_equal

```
struct Sobj* Seval_bigger_and_equal(struct Sobj *obj1, struct Sobj *obj2);
```

Description Implements the **greater-than-or-equal operator (`>=`)**.

Rules

- Returns `TRUE_OBJ` if either:
    - `obj1 > obj2` , or

    - `obj1 == obj2`.
- Uses the same semantics as `Seval_bigger` and `Seval_equal`.

Seval_smaller_and_equal

```
struct Sobj* Seval_smaller_and_equal(struct Sobj *obj1, struct Sobj *obj2);
```

Description Implements the **less-than-or-equal operator (`<=`)**.

Rules

- Returns `TRUE_OBJ` if either:
    - `obj1 < obj2`, or
    - `obj1 == obj2`.
- Uses the same semantics as `Seval_smaller` and `Seval_equal`.

9.2.2 `Serror` Functions

Error handling is a **critical part of any interpreter or compiler**, since it directly affects how developers **understand and debug their programs**. The `Serror` module provides a **centralized mechanism** for creating, reporting, and managing errors across different stages of program execution:

- **Lexical errors**: when invalid characters or tokens are encountered during scanning.
- **Syntax errors**: when the parser detects invalid grammar or missing constructs.
- **Compile-time errors**: when semantic issues occur before generating bytecode or machine code.
- **Runtime errors**: when execution encounters invalid operations (e.g., division by zero).
- **Unknown/internal errors**: for unexpected conditions or unhandled cases.

By consolidating all error handling into the `Serror` API, Suny ensures that **all errors are reported in a consistent, structured format**.

The `Serror` Structure

```
struct Serror {
    char* type;           // Error type string (e.g., "SyntaxError")
    char* message;        // Human-readable explanation of the error
    int line;             // Line number where the error occurred
    int column;           // Column number where the error occurred
    struct Slexer *lexer; // Optional reference to the lexer (for file context)
};
```

- `type` distinguishes between error categories (`SyntaxError`, `RuntimeError`, etc.).
- `message` provides a clear description for the user.
- `line` and `column` give precise location info.
- `lexer` provides context (such as file name) when available.

Errors are **always created dynamically** and may be passed around between compiler phases.

Function Reference

Serror_new

The function:

```
struct Serror *Serror_new(void);
```

Allocates and initializes a fresh `Serror` object with default values:

- `type = NULL`
- `message = NULL`
- `line = 0`, `column = 0`
- A fresh `Slexer` instance is created for context

This function is typically the **first step** when building a new error. Callers should later release the error with `Serror_free`.

Serror_set

The function:

```
struct Serror *Serror_set(char *type, char *message, struct Slexer *lexer);
```

Creates and initializes a new error using:

- **type** → e.g. `"SyntaxError"`
- **message** → short explanation of what went wrong
- **lexer** → provides the current file, line, and column

Example:

```
struct Serror *err = Serror_set("SyntaxError", "Unexpected token", lexer);
Serror_syntax_error(err);
```

Serror_set_line

The function:

```
struct Serror *Serror_set_line(char* type, char* message, int line, int column);
```

Creates an error when no lexer is available. Useful in low-level contexts, or when error location is determined manually (e.g., static analysis).

Serror_syntax_error

The function:

```
int Serror_syntax_error(struct Serror *error);
```

Handles **syntax errors**.

- Prints the error in the format:

```
SyntaxError: Unexpected token
At file 'example.suny', line 12
```

- Execution halts immediately (via `break_loop()`).

Serror_runtime_error

The function:

```
int Serror_runtime_error(char *message, struct Slexer *lexer);
```

Handles **runtime errors**.

- Reports a problem that occurred **during execution** (e.g., division by zero).
- Includes the current line and column.

Example output:

```
RuntimeError: Division by zero
At line 5, column 14
```

Serror_compile_error

The function:

```
int Serror_compile_error(char *message, struct Slexer *lexer);
```

Handles **compile-time errors**, usually during semantic analysis or code generation. These errors are tied to a file context rather than execution state.

Example output:

```
CompileError: Undefined variable 'x'
At file 'main.suny', line 8
```

Serror_unknown_error

The function:

```
int Serror_unknown_error(char *message, struct Slexer *lexer);
```

Handles **unexpected internal errors** (e.g., bugs in the compiler). Used as a **catch-all fallback** to avoid silent failures.

Serror_parser

The function:

```
int Serror_parser(char *message, struct Slexer *lexer);
```

Shortcut for reporting parser-related syntax errors. Equivalent to calling `Serror_set` followed by `Serror_syntax_error`.

Serror_print

The function:

```
int Serror_print(struct Serror *error);
```

Prints a generic error message without a strict category. Mainly useful for **debugging** or **non-critical warnings**.

Serror_free

The function:

```
int Serror_free(struct Serror *error);
```

Releases memory associated with an error object:

- Clears type, message, line, and column.
- Frees the attached lexer (if any).
- Finally frees the error itself.

This function is essential to prevent memory leaks in long-running compilers or REPL sessions.

`Sast_expected_expression`

```
int Sast_expected_expression(struct Sast *sast);
```

Parser helper:

- Ensures the current AST node is a valid expression.
- If not, raises a syntax error `"Expected expression"` at the current lexer location.

Error Handling Philosophy

The `Serror` system follows these **design goals**:

1. **Consistency** → All errors use a common structure and output format.
2. **Clarity** → Messages are human-readable and point to exact locations.
3. **Extensibility** → New error categories can be added by extending `type`.

4. **Fail-fast behavior** → Most errors immediately halt execution (`break_loop`) to prevent cascading failures.
5. **Integration with AST/Lexer** → Errors carry contextual information, making debugging easier.

Example Workflow

Consider the following invalid Suny code:

```
x = (1 + )
```

Parsing this would trigger:

```
Sast_expected_expression(sast);
```

Which internally calls:

```
SyntaxError: Expected expression
At file 'test.suny', line 3
```

Execution then halts, preventing further misleading errors.

Summary

The `Serror` API provides a **robust, centralized mechanism** for error handling in Suny. From lexing to runtime, all stages of the language pipeline use the same tools, ensuring developers always see **clear, consistent, and actionable error messages**.

9.2.3 `Sfunc` Functions

The **`Sfunc` module** is the core abstraction for **function objects** in the Suny runtime system. A function in Suny is not merely a piece of code: it is a structured object that encapsulates:

1. **Code**: The bytecode (`Scode`) generated by the compiler for the function body.
2. **Arguments**: The number of parameters that the function expects.
3. **Frame (optional)**: An execution environment that contains the local variables, evaluation stack, and scope information used during execution.
4. **Metadata**: Additional bookkeeping information such as inner functions, capacity for nested closures, and references to higher-level execution structures.

By treating functions as **first-class objects**, Suny enables them to be:

- Assigned to variables.
- Passed as arguments to other functions.
- Returned from functions.
- Stored inside data structures (lists, maps, etc.).

This flexibility mirrors the design philosophy of modern dynamic languages like Python, Lua, or JavaScript, where functions are part of the object system.

The `Sfunc` API provides facilities for creating, configuring, extending, and freeing function objects. Below is a detailed breakdown.

`Sfunc` struct usually defined in `Stype.h` :

```
struct Sfunc {
    struct Sfunc *inner;              // for the inner function

    struct Sobj** inner_funcs;        // store multi inner functions
    int inner_funcs_size;             // size of inner_funcs,
    int inner_funcs_capacity;         // capacity of inner_funcs,

    struct Sframe *frame;             // stack frame of the function
    struct Scode *code;               // function store code

    int args_index;                   // arguments index
    int code_index;                   // code index

    int args_size;                    // arguments size
    int code_size;                    // code size

    struct Sobj *obj;
    struct Sobj **params;             // store the arguments

    struct Scall_context *call_context;
};
```

Sfunc_obj_new

The function:

```
struct Sfunc *Sfunc_obj_new(void);
```

Creates a new, empty function object. This function allocates memory for a `Sfunc`, initializes its fields to safe defaults, and prepares it for later configuration.

- **Behavior**:
    - `args_index` and `code_index` set to `0`.
    - `inner_funcs` initialized as `NULL`.
    - `params` allocated with a maximum argument size (`MAX_ARGS_SIZE`).
    - No `code` or `frame` is attached yet.
- **Returns**: A pointer to a new `Sfunc` object.

Usage Example:

```
struct Sfunc *func = Sfunc_obj_new();
// func is empty, must be configured before execution
```

Sfunc_free

The function:

```
int Sfunc_free(struct Sfunc *func);
```

Releases memory associated with a function object. This includes its code, argument array, and the function object itself.

- **Parameters**:
    - `func`: the function object to release.
- **Returns**: `0` on success.
- **Notes**:
    - Ensures that `Scode_free()` is called if function bytecode exists.
    - Should always be called when a function is no longer referenced, to avoid memory leaks.

Sfunc_ready

The function:

```
struct Sfunc *Sfunc_ready(struct Sfunc *func, int args_size);
```

Prepares a function to be executed by associating it with the number of arguments it expects. At present, this function is a minimal placeholder that simply returns the function unchanged, but it establishes a consistent API entry point for future extensions (such as argument binding or optimization passes).

- **Parameters**:
    - `func` : the function object.
    - `args_size` : number of arguments expected.
- **Returns**: The prepared function object.

Sfunc_set

The function:

```
struct Sfunc *Sfunc_set(struct Scode *code, int args_size, int code_size);
```

Creates a fully configured function object from bytecode and metadata. This is one of the main constructors used after compilation.

- **Parameters**:
    - `code` : compiled function bytecode.
    - `args_size` : number of arguments.
    - `code_size` : number of instructions in the bytecode.
- **Returns**: A pointer to a fully configured function object.

Example:

```
struct Scode *code = compile_function_ast(ast_node);
struct Sfunc *func = Sfunc_set(code, 2, code->size);
```

Sfunc_set_func

The function:

```
struct Sfunc *Sfunc_set_func(
    struct Sfunc *func,
    struct Sframe *frame,
    struct Scode *code,
    int args_size
);
```

Configures an existing function object by attaching a frame, code, and argument metadata. This method is used when functions are created dynamically at runtime or need to inherit execution environments.

- **Parameters**:
    - `func` : function object to configure.
    - `frame` : execution frame.
    - `code` : function bytecode.
    - `args_size` : expected argument count.
- **Returns**: The updated function object.

Sobj_set_func

```
struct Sobj *Sobj_set_func(struct Sfunc *func);
```

Wraps a `Sfunc` inside a generic Suny object (`Sobj`). This step is crucial, because in Suny everything that can be passed around or stored in a variable must be represented as an `Sobj` .

- **Parameters**:
    - `func` : the function object.
- **Returns**: A new `Sobj` of type `FUNC_OBJ` containing the function.

Example:

```
struct Sfunc *f = Sfunc_set(code, 1, code->size);
struct Sobj *obj = Sobj_set_func(f);
// obj can now be stored in variables, pushed on stack, etc.
```

Sfunc_set_code

The function:

```
struct Sfunc *Sfunc_set_code(struct Sfunc *func, struct Scode *code);
```

Assigns a new bytecode object to an existing function. This updates both the `code` pointer and the `code_size` field.

- **Parameters**:
  - `func` : function object to modify.
  - `code` : new code block.
- **Returns**: The updated function object.

Sfunc_insert_code

The function:

```
struct Sfunc *Sfunc_insert_code(struct Sfunc *func, struct Scode *code);
```

Appends new bytecode to an existing function's code segment. This is used for **incremental compilation** or **dynamic code injection** where the function body is constructed in multiple passes.

- **Parameters**:
  - `func` : target function.
  - `code` : additional code to insert.
- **Returns**: The updated function object.

Design Philosophy

The `Sfunc` abstraction balances **flexibility** and **simplicity**:

- Functions are **lightweight objects** with minimal mandatory fields.
- They can be created in different ways:
  - Directly (`Sfunc_obj_new`)
  - From compiled bytecode (`Sfunc_set`)
  - By attaching runtime context (`Sfunc_set_func`)
- Functions are **integrated into the object system** through `Sobj_set_func` , ensuring consistent handling with other types.

By isolating all function-related logic into `Sfunc` , the Suny runtime maintains a clear separation of responsibilities:

- **Parsing/Compilation** produces `Scode` .
- **Execution** uses `Sframe` .
- **Function management** is handled entirely through `Sfunc` .

This modularity makes the system easier to extend (e.g., closures, generators, async functions) without complicating other runtime components.

Example of creating a Suny function in C

```
struct Scompiler *compiler = Scompiler_new(); // creat a new compiler
struct Sframe* frame = Sframe_new(); // creat a new stack frame

SunyInstallLib(compiler, frame);

struct Scode *code = Scompile_from_string("print(\"this is function\")", compiler);
struct Sfunc *func = Sfunc_set(code, 0, code->size);
struct Sobj *f_obj = Sobj_set_func(func);

// call it

Svm_run_func(frame, f_obj);
```

9.2.4 The `Slist` Functions

The **`Slist` group** provides a low-level C API for working with Suny's list type. A Suny list is a dynamically sized, ordered collection of Suny objects (`struct Sobj*`). It supports **random access**, **append**, **pop**, **concatenation**, **multiplication (repeat)**, and **range creation**.

Unlike fixed-size C arrays, an `Slist` grows automatically as needed. Internally, each `Slist` manages:

- `items` → a dynamically allocated array of `Sobj*`
- `size` → the current number of elements in the list
- `capacity` → allocated slots in `items` , resized when necessary

To make lists usable inside the Suny runtime, you wrap them with `Sobj_make_list` . All operations must maintain Suny's memory model, including reference counting and garbage collection.

Slist_new

The function:

```
struct Slist* Slist_new(void);
```

Creates a new empty list. The returned list starts with `size = 0` and an initial capacity determined by the implementation.

- **Returns**: a pointer to a newly allocated `Slist` .
- **Errors**: may return `NULL` if memory allocation fails.
- **Responsibility**: caller must eventually call `Slist_free` .

Example:

```
struct Slist* list = Slist_new();
```

Slist_free

The function:

```
int Slist_free(struct Slist* list);
```

Frees the memory used by the list and its elements (if owned). After this call, the list pointer must not be used.

- **Returns**: `0` on success, nonzero on error.
- **Note**: If objects in the list are referenced elsewhere, you must ensure reference counts are decremented correctly before freeing.

Example:

```
Slist_free(list);
```

Slist_add

The function:

```
struct Slist* Slist_add(struct Slist* list, struct Sobj* obj);
```

Appends `obj` to the end of the list. If the list is full, its capacity will grow (typically doubling).

- **Returns**: the same list pointer for chaining.
- **Side effects**: increments reference count of `obj` if GC-managed.

Example:

```
Slist_add(list, Sobj_set_int(42));
```

Slist_get

The function:

```
struct Sobj* Slist_get(struct Slist* list, int index);
```

Fetches the object stored at position `index`.

- **Indexing**: zero-based (C-style).
- **Returns**: the `Sobj*` at `index`, or `NULL` if out of range.
- **Ownership**: caller should not free the returned object directly unless it increases the refcount.

Example:

```
struct Sobj* val = Slist_get(list, 0);
Sio_write(val); // prints 42
```

Sobj_make_list

The function:

```
struct Sobj* Sobj_make_list(struct Slist* list);
```

Wraps a raw `Slist` inside an `Sobj` of type `LIST_OBJ`. This makes the list a Suny value that can be passed around in the interpreter.

- **Returns**: a new `Sobj*` referencing the list.
- **Note**: Suny takes ownership of the list through the wrapper.

Example:

```
struct Sobj* obj = Sobj_make_list(list);
Sio_write(obj); // prints [42]
```

Slist_change_item

The function:

```
struct Slist* Slist_change_item(struct Slist* list, int index, struct Sobj* obj);
```

Replaces the object at `index` with a new `obj`.

- **Returns**: the same list pointer.
- **Ownership**: caller is responsible for freeing/releasing the old element if it is no longer needed.
- **Errors**: no change if `index` is out of bounds.

Example:

```
Slist_change_item(list, 0, Sobj_set_int(99));
```

Slist_pop

The function:

```
struct Slist* Slist_pop(struct Slist* list);
```

Removes the last element from the list.

- **Returns**: the modified list.
- **Note**: use `Slist_get(list, list->size - 1)` first if you need the removed element.
- **Errors**: if the list is empty, does nothing.

Example:

```
Slist_pop(list);
```

Slist_append

The function:

```
struct Slist* Slist_append(struct Slist* list1, struct Slist* list2);
```

Concatenates all elements of `list2` to the end of `list1`.

- **Returns**: `list1` with new elements added.
- **Behavior**: copies references, does not duplicate objects.
- **Errors**: may fail if memory allocation fails.

Example:

```
Slist_append(list1, list2);
```

Slist_mul

The function:

```
struct Slist* Slist_mul(struct Slist* list1, int num);
```

Creates a new list containing `num` repetitions of the elements in `list1`.

- **Returns**: a new list (not the same as `list1`).
- **Example**: `[1, 2] * 3` → `[1, 2, 1, 2, 1, 2]`.

Example:

```
struct Slist* repeated = Slist_mul(list, 3);
Sio_write(Sobj_make_list(repeated));
```

Slist_range

The function:

```
struct Slist* Slist_range(int start, int end);
```

Creates a new list containing the sequence of integers from `start` up to but not including `end`.

- **Returns**: a new list of number objects.
- **Example**: `Slist_range(0, 5)` → `[0, 1, 2, 3, 4]`.

Example:

```
struct Slist* r = Slist_range(0, 10);
Sio_write(Sobj_make_list(r));
```

Summary of `Slist` API

Function	Purpose

Function	Purpose
`Slist_new`	Create a new empty list
`Slist_free`	Free a list and its elements
`Slist_add`	Append an object to the list
`Slist_get`	Retrieve an element by index
`Sobj_make_list`	Wrap a list inside an `Sobj`
`Slist_change_item`	Replace an element at a given index
`Slist_pop`	Remove the last element
`Slist_append`	Concatenate two lists
`Slist_mul`	Repeat a list N times
`Slist_range`	Create a numeric range list

9.2.5 The `Sstr` Functions

The **`Sstr` group** provides a low-level C API for working with Suny's string type. A Suny string (`struct Sstr`) is an immutable sequence of characters, similar to strings in high-level languages. It supports **creation**, **concatenation**, **repetition (multiplication)**, and **deallocation**.

Unlike raw C strings (`char*`), an `Sstr` :

- Stores its **length** explicitly (no need for `strlen`).
- Can hold any byte sequence (not required to be null-terminated).
- Is **heap-allocated** and must be freed explicitly with `Sstr_free` .

Internally, each `Sstr` manages:

- `data` → a dynamically allocated buffer of characters
- `size` → number of characters stored (can include `'\0'`)

Sstr_new

The function:

```
struct Sstr* Sstr_new(void);
```

Creates a new empty string (`""`).

- **Returns**: a pointer to a newly allocated `Sstr` .
- **Errors**: may return `NULL` if memory allocation fails.
- **Responsibility**: caller must eventually call `Sstr_free` .

Example:

```
struct Sstr* str = Sstr_new();
```

Sstr_new_from_char

The function:

```
struct Sstr* Sstr_new_from_char(char* chr, int size);
```

Creates a new string from a raw character buffer.

- **Parameters**:
    - `chr` : pointer to character array (not necessarily null-terminated).
    - `size` : number of characters to copy.
- **Returns**: a new `Sstr*` containing the copied data.

- **Errors**: returns `NULL` if memory allocation fails.

Example:

```
struct Sstr* hello = Sstr_new_from_char("Hello", 5);
```

Sstr_free

The function:

```
void Sstr_free(struct Sstr* str);
```

Frees the memory used by a string.

- **Parameters**:
    - `str` : string to free.
- **Effect**: releases both the buffer and struct itself.
- **Note**: After this call, `str` must not be used again.

Example:

```
Sstr_free(hello);
```

Sstr_add

The function:

```
struct Sstr* Sstr_add(struct Sstr* str1, struct Sstr* str2);
```

Concatenates two strings into a new one.

- **Parameters**:
    - `str1` : first string
    - `str2` : second string
- **Returns**: new string `str1 + str2` .
- **Errors**: returns `NULL` if allocation fails.

Example:

```
struct Sstr* a = Sstr_new_from_char("foo", 3);
struct Sstr* b = Sstr_new_from_char("bar", 3);
struct Sstr* c = Sstr_add(a, b); // "foobar"
```

Sstr_mul

The function:

```
struct Sstr* Sstr_mul(struct Sstr* str, int n);
```

Creates a new string consisting of `n` repetitions of `str` .

- **Parameters**:
    - `str` : input string
    - `n` : number of repetitions
- **Returns**: a new repeated string
- **Errors**: if `n <= 0` , may return empty string

Example:

```
struct Sstr* x = Sstr_new_from_char("ha", 2);
struct Sstr* y = Sstr_mul(x, 3); // "hahaha"
```

So these are some basics group in Suny, here are some examples of making a `Suny` program but in `C`

---

10. Garbage collector

1. Introduction

Memory is one of the most important resources in any computer system. Every program, regardless of its complexity, relies on memory to store variables, objects, and runtime data structures. However, memory is also finite, and careless use of it can lead to severe problems such as **memory leaks**, **dangling pointers**, or even program crashes.

Traditionally, low-level languages such as **C** or **C++** require programmers to manage memory manually. This involves explicitly allocating memory (e.g., with `malloc` or `new`) and freeing it (with `free`) when the memory is no longer needed. While this approach gives developers full control, it also introduces significant risks. A single missing `free()` can cause a program's memory usage to grow endlessly, while an accidental double `free()` can corrupt memory and lead to undefined behavior.

To avoid these pitfalls, modern languages often implement **automatic memory management**, commonly known as **Garbage Collection (GC)**. Garbage Collection is a runtime system that automatically determines which objects are no longer accessible by the program and reclaims the memory they occupy. With GC, programmers are free from the burden of manually managing memory, and programs become more robust and secure by default.

Garbage Collection in Suny

Suny, as a high-level and beginner-friendly programming language, adopts automatic memory management from the ground up. The design philosophy of Suny emphasizes **safety**, **simplicity**, and **ease of learning**, making garbage collection a natural choice. By integrating GC directly into the runtime, Suny ensures that developers can focus on problem-solving instead of worrying about memory allocation details.

The specific strategy used by Suny is **Reference Counting Garbage Collection**. This method assigns each object a counter that tracks how many active references point to it. When the counter reaches zero, the object is destroyed immediately, freeing the associated memory. This model provides a predictable, deterministic approach to object lifetimes, which aligns well with Suny's goal of being transparent and intuitive for programmers.

Why Suny Needs GC

There are several key reasons why Suny integrates garbage collection as a core runtime feature:

1. **Preventing memory leaks** Without GC, programmers must remember to release memory manually. Even a small oversight can cause memory to accumulate indefinitely. GC ensures that memory is reclaimed as soon as objects are no longer used.
2. **Improved safety** Manual memory management often leads to dangling pointers—references to memory that has already been freed. Accessing such memory causes crashes or unpredictable results. GC eliminates this class of error entirely.
3. **Beginner-friendly programming model** One of Suny's design goals is to be approachable for newcomers. By handling memory automatically, Suny lowers the entry barrier to programming, letting learners focus on logic, algorithms, and creativity instead of low-level details.
4. **Cleaner, more expressive code** Programs written in Suny do not require explicit `free()` or memory-release calls. This reduces boilerplate code and avoids clutter, making programs easier to read and maintain.
5. **Consistency across platforms** Since Suny abstracts away memory management, programs behave consistently regardless of the underlying operating system or architecture.

---

2. Fundamentals of Reference Counting

2.1 What is Reference Counting?

At the heart of Suny's garbage collector lies a simple yet powerful idea: **Reference Counting**. This technique assigns every object in the runtime a special integer field known as the **reference count** (often abbreviated as *refcount*). The reference count is a measure of how many "owners" or "handles" currently point to that object.

The Core Concept

Whenever a variable, data structure, or another object points to a given object, the runtime **increments** its reference count. Whenever one of those references is removed or goes out of scope, the runtime **decrements** the count. When the count reaches **zero**, it means no part of the program can access the object anymore. At that exact moment, the object is destroyed, and its memory is reclaimed.

This ensures that memory usage stays tight: as soon as an object is no longer needed, Suny frees it without requiring any action from the programmer.

Analogy: Borrowed Books in a Library

A helpful way to imagine reference counting is to think about a library that tracks how many people currently have borrowed copies of a particular book:

- When someone borrows a copy → the counter goes up.
- When someone returns it → the counter goes down.
- If the counter drops to zero → it means no one is reading that book anymore, and the library can safely put it back into storage.

In Suny, objects behave exactly like those books: if nobody “holds” an object, it is no longer needed and can be removed.

Object Lifecycle in Suny

Each object in Suny passes through a simple lifecycle managed by its reference counter:

1. **Creation** When an object is first created (e.g., a list `[1, 2, 3]`), it starts with a reference count of `1`, because the variable holding it is considered the first reference.
2. **Sharing** If the object is assigned to another variable, passed to a function, or stored in a container, the reference count increases.
3. **Releasing** When variables go out of scope, or an object is removed from a container, the reference count decreases.
4. **Destruction** When the reference count hits `0`, the object is destroyed immediately. Its memory is reclaimed, and if the object holds external resources (e.g., files or sockets), they are closed as well.

Why Reference Counting Works Well in Suny

- **Immediate cleanup**: Memory is freed exactly when objects are no longer needed, without waiting for a background GC cycle.
- **Predictable behavior**: Programmers can reason about when objects will be destroyed, which is especially useful for managing resources like files or sockets.
- **Simplicity**: The model is easy to explain and understand, making Suny approachable for both beginners and advanced developers.

Comparison with Other Techniques

While other garbage collection methods (such as **mark-and-sweep** or **generational GC**) operate by scanning the memory heap at certain intervals, reference counting works **incrementally** during program execution. Every assignment or deletion updates counters on the fly. This means Suny does not need to “pause the world” to collect garbage, keeping the runtime responsive and lightweight.

The trade-off is that reference counting cannot handle **circular references** (covered later in Section 5), but its benefits of **simplicity and determinism** make it an excellent first choice for Suny’s design philosophy.

2.3 Operations on Reference Count

Reference counting in Suny is governed by two fundamental operations: **incrementing** and **decrementing** the reference counter. Although the operations themselves are conceptually simple, their implementation details and runtime implications are crucial for both performance and correctness. This section explores how Suny performs these operations, the circumstances in which they occur, and the subtleties that arise in real-world programs.

2.3.1 Incrementing a Reference Count (INCREF)

The **increment operation** increases the reference count of an object by one whenever a new reference is created. It is the most common memory-management operation in Suny, triggered by simple actions such as assigning variables, passing arguments, or inserting objects into containers.

When Does INCREF Occur?

1. **Variable Assignment**

   ```
   a = [10, 20, 30]   # refcount = 1
   b = a              # refcount = 2
   ```

   Here, `b` now points to the same list as `a`, so the runtime increments the object’s refcount.

2. **Function Calls**

   ```
   function foo(x) do
       print(x)

   arr = [1, 2, 3]    # refcount = 1
   foo(arr)           # refcount = 2 while inside foo()
   ```

   Passing `arr` to `foo` creates another reference through the parameter `x`. When the function returns, `x` goes out of scope, and the refcount decreases again.

Implementation Notes

- INCREF is typically just a single machine instruction (`obj->refcount++`).
- Despite its simplicity, it must be **inlined and optimized**, as INCREF may be executed billions of times in long-running programs.

- In a multi-threaded runtime, INCREF must use atomic operations to avoid race conditions.

2.3.2 Decrementing a Reference Count (DECREF)

The **decrement operation** decreases the reference count of an object by one whenever a reference is discarded. This is equally common and even more important than INCREF, since it determines when an object can be destroyed.

When Does DECREF Occur?

1. **Variable Reassignment**

   ```
   a = [42]        # refcount = 1
   b = a           # refcount = 2
   a = null        # refcount = 1
   b = null        # refcount = 0 → object freed
   ```

   Once both variables are cleared, the list is immediately destroyed.

2. **Scope Exit**

   ```
   function foo() do
       x = [1, 2, 3]   # refcount = 1
       return 42
   foo()  # after function ends, x goes out of scope → refcount = 0
   ```

   When the function exits, the local variable `x` is discarded, reducing the reference count of the list. Since no references remain, the object is destroyed.

3. **Container Removal**

   ```
   arr = [1, 2, 3]
   lst = [arr]       # refcount = 2
   pop(lst)
   arr = null        # refcount = 0 → freed
   ```

   Removing `arr` from `lst` triggers a DECREF, and setting `arr` to `null` finishes the process.

Immediate Destruction

If DECREF reduces the counter to zero, the runtime immediately:

1. Calls the object's destructor, if defined.
2. Recursively decrements the counts of any objects it references.
3. Returns the object's memory back to the allocator.

This **deterministic destruction** is one of the strongest advantages of reference counting. Unlike tracing garbage collectors, Suny does not delay cleanup until a background cycle.

2.3.3 Ownership Transfer and Optimization

Although INCREF and DECREF are conceptually simple, they can introduce performance overhead if applied too often. Suny's runtime is designed with the following optimizations in mind:

- **Borrowed References** In some cases, a function can use an object without taking full ownership of it. For example, reading an element from a list might not need an INCREF if the reference is short-lived.
- **Move Semantics** If an object is created in one place and then "moved" to another owner, Suny may avoid extra increments/decrements by transferring ownership directly.
- **Deferred Decrement** For certain temporary values, the runtime may defer DECREF until a safe checkpoint, reducing the number of increments and decrements in tight loops.

These techniques reduce redundant operations while preserving correctness.

2.3.4 Subtleties and Pitfalls

Although INCREF and DECREF are conceptually clear, real implementations face subtle challenges:

- **Thread Safety** Without atomic operations, simultaneous INCREF/DECREF calls from multiple threads can corrupt reference counts.
- **Cascading Destruction** When a complex object (e.g., a list of dictionaries of lists) is freed, DECREF must recursively cascade through its children. This process must be carefully ordered to avoid freeing objects still in use.
- **Performance Bottlenecks** In performance-critical programs, the sheer volume of INCREF/DECREF calls can dominate runtime cost. Any inefficiency here multiplies rapidly.

- **Bugs from Miscounts** A missing INCREF leads to premature destruction (dangling references), while a missing DECREF causes memory leaks. Both can be subtle and difficult to debug.

2.3.5 Summary

The operations of **incrementing and decrementing reference counts** form the core mechanism of memory management in Suny. Despite being represented by simple integer adjustments, these operations have far-reaching implications:

- They must be executed with high efficiency, since they occur constantly in every program.
- They provide predictable and immediate cleanup, making resource management reliable.
- They enable Suny's philosophy of simplicity and determinism in garbage collection.

The elegance of reference counting lies in its balance: an extremely simple abstraction that, when implemented carefully, yields a powerful and robust memory management model.

---

11. Userdata

In Suny, **userdata** is a specialized type of object that allows developers to store **arbitrary external or custom data** directly within the runtime while still being fully managed by Suny's garbage collector. Unlike typical Suny objects such as strings, lists, or dictionaries, userdata is designed as a **flexible container** that can hold raw memory, C structures, file handles, sockets, or any other resource that needs to coexist with Suny's memory-managed environment.

The primary purpose of userdata is to provide a **bridge between low-level resources and high-level managed code**. This allows developers to interact with external systems, libraries, or custom data types while leveraging Suny's memory safety, reference counting, and deterministic destruction mechanisms.

Userdata in Suny has several important characteristics that make it both powerful and safe:

1. **Generic Storage Capability** A single userdata object can hold any kind of external data. This could be:
    - A pointer to a C struct or a C++ object.
    - A handle to a system resource such as a file or network socket.
    - Arbitrary binary data managed outside of Suny's normal object types.
2. **Integration with Garbage Collection** Although userdata may store raw pointers, it is fully compatible with Suny's **reference counting** garbage collector. When the last reference to a userdata object is released, Suny automatically triggers its destruction routine and frees the memory or resources it encapsulates.
3. **Optional Type Information** Each userdata object can optionally be associated with a **type descriptor**, which provides metadata about the data it holds. This descriptor can define:
    - Custom destructors or finalizers to safely release external resources.
    - Type-specific operations that allow Suny scripts to interact with the userdata safely.
4. **Deterministic Resource Management** Because Suny uses reference counting, any resources wrapped in userdata are released **immediately when the object becomes unreachable**. This ensures timely cleanup of critical resources, avoiding resource leaks and making Suny suitable for systems programming tasks.

Userdata is particularly useful in scenarios where Suny scripts need to interact with:

- **External Libraries** Wrapping C libraries or other native code so that they can be safely used from Suny scripts.
- **System Resources** Managing open files, network sockets, or other handles without leaking them, as the reference counting ensures deterministic destruction.
- **Custom Data Structures** Allowing the creation of specialized structures that are not natively supported in Suny, but can still participate in the garbage-collected environment.
- **Interfacing with Hardware or Low-Level APIs** Userdata can encapsulate memory buffers, device contexts, or other hardware resources, providing safe high-level access while maintaining proper cleanup.

11.1 Defining a Custom Userdata Struct

In Suny, a **custom userdata struct** allows developers to define their own object types in C that can be safely managed by the garbage collector. The main idea is to **wrap external data** (such as C structs, file handles, or other resources) inside a struct that Suny can track and manage automatically.

Basic structure of a userdata object in C:

```
struct Suserdata {
    void* data;        // Pointer to external or custom data
    struct Stype* type; // Type descriptor for metadata and behaviors
};
```

- `data` stores the actual content or resource that the userdata encapsulates.
- `type` points to an **Stype struct**, which defines the userdata's type information, such as its behavior, associated metamethods, and destructor.

`Suserdata` is typically defined in `Stype.h`, which is the core structure used by Suny to represent **all datatypes**, including strings, lists, numbers, and custom userdata.

We can creat own custom datatype with `Suserdata` using C-API. You can see the tutorial of how to use C-API to make Suny function and libs in `# 12. Using Suny C-API to make library`, these are step by step to make your own datatype

11.1.1 Steps to Create a Custom Data Type

1. Define your custom C struct

Lets creat a vector struct, this struct contains the data you want your Suny object to hold.

```
#include <Suny/Suny.h>

struct Svector {
    float x;
    float y;
};
```

---

2. Allocate an Suserdata object

Next we creat a Suny function that return the userdata object

```
struct Svector* Svector_new(float x, float y) {
    struct Svector* vector = (struct Svector*) malloc(sizeof(struct Svector));
    vector->x = x;
    vector->y = y;
    return vector;
}

struct Sobj* Sobj_make_vector(struct Sframe* frame) {
    struct Sobj* Sx = Sframe_pop(frame);
    struct Sobj* Sy = Sframe_pop(frame);

    float x = ValueOf(Sx);
    float y = ValueOf(Sy);

    struct Svector* vector = Svector_new(x, y);
    struct Sobj* obj = Sobj_make_userdata(vector);

    return obj;
}

SUNY_API struct Sframe* Smain(struct Sframe* frame, struct Scompiler* compiler) {
    SunyInitialize_c_api_func(frame, compiler, 20, "vector", 2, Sobj_make_vector);
    return frame;
}
```

Now we **compile it into .dll and import the library**:

```
import "vector"

let vec = vector(1, 2)
print(vec)
```

```
C:\> suny main.suny
0
C:\>
```

As you can see the print function don't work because we don't set **Metamethod** to my own datatype yet.

```
let vec1 = vector(1, 2)
let vec2 = vector(3, 2)
let vec3 = vec1 + vec2
```

```
C:\> suny main.suny
0
C:\>
```

3. Using Metamethod

In Suny, **metamethods** are special functions that allow you to **customize the behavior of your userdata objects**. They define how a userdata object responds to standard operations, making it possible to integrate low-level C structures into high-level Suny scripts naturally and safely.

Without metamethods, a custom userdata object in Suny is essentially **opaque**: it stores raw data but the runtime does not know how to interpret it for operations like addition, printing, comparisons, or function calls. Metamethods give you a way to define this behavior explicitly.

Metamethods in Suny can serve several purposes:

1. **Operator Overloading** You can define how your userdata reacts to arithmetic or comparison operators such as `+`, `-`, `*`, `/`, `==`, `<`, and `>`.
    - Example: a `vector` userdata can define `+` to perform element-wise addition instead of returning an integer.
2. **Custom String Representation** The `__tostring__` metamethod controls what happens when your userdata is printed using the `print()` function.
    - Without it, printing a userdata object will typically display a memory address or a meaningless default value.
3. **Function-like Behavior** Using the `__call__` metamethod, a userdata object can behave like a function.
    - This allows the object to be executed with arguments, enabling flexible APIs where userdata can store data and perform actions.
4. **Custom Destructors / Finalizers** Metamethods can define actions to take when a userdata object is destroyed.
    - This is especially important for releasing external resources like file handles, sockets, or memory buffers, complementing Suny's **reference-counted garbage collection**.
5. **Extendable Behavior** By combining different metamethods, you can make your userdata support multiple operations naturally, making the object feel like a first-class Suny type despite being implemented in C.

By defining metamethods, developers can **integrate low-level C data seamlessly into Suny scripts**, giving userdata meaningful behavior while maintaining full memory safety, deterministic destruction, and predictable execution.

In Suny, **metamethods** are essentially **function pointers attached to userdata or Sobj objects** that define custom behavior for operations such as arithmetic, comparison, printing, function calls, and destruction. This allows userdata objects to act like **first-class Suny types** while still being implemented in low-level C.

Suny defines a `struct Smeta` that contains **all possible metamethods** defined in `smeta.h`:

```
struct Smeta {
    struct Sobj* (*mm_add)(struct Sobj*, struct Sobj*);     // +
    struct Sobj* (*mm_sub)(struct Sobj*, struct Sobj*);     // -
    struct Sobj* (*mm_mul)(struct Sobj*, struct Sobj*);     // *
    struct Sobj* (*mm_div)(struct Sobj*, struct Sobj*);     // /

    struct Sobj* (*mm_eq)(struct Sobj*, struct Sobj*);      // ==
    struct Sobj* (*mm_ne)(struct Sobj*, struct Sobj*);      // !=
    struct Sobj* (*mm_gt)(struct Sobj*, struct Sobj*);      // >
    struct Sobj* (*mm_ge)(struct Sobj*, struct Sobj*);      // >=
    struct Sobj* (*mm_lt)(struct Sobj*, struct Sobj*);      // <
    struct Sobj* (*mm_le)(struct Sobj*, struct Sobj*);      // <=

    struct Sobj* (*mm_call)(struct Sobj*, struct Sobj*);    // ()
    struct Sobj* (*mm_index)(struct Sobj*, struct Sobj*);   // []

    struct Sobj* (*mm_tostring)(struct Sobj*);             // print / string representation
    struct Sobj* (*mm_type)(struct Sobj*);                 // type info

    int (*mm_free)(struct Sobj*);                          // free / destructor
};
```

Key Points About Suny Metamethods

1. **Customizable Behavior for Operators**
    - Arithmetic (`+`, `-`, `*`, `/`) and comparisons (`==`, `!=`, `<`, `<=`, `>`, `>=`) are mapped to function pointers in `Smeta`.
    - By assigning your own functions to these fields, you can define **how your userdata reacts to each operator**.
2. **Printing and Type Representation**
    - `mm_tostring` defines what happens when `print()` is called on the object.
    - `mm_type` allows your userdata to provide a **human-readable type name**.
3. **Function Calls and Indexing**
    - `mm_call` allows the object to be callable like a function.
    - `mm_index` provides optional indexing behavior if your userdata needs it (even though Suny doesn't have tables/dicts by default, this can support

custom behaviors).

4. **Deterministic Cleanup**
    - `mm_free` is invoked when the userdata is about to be destroyed by **reference-counted GC**.
    - This is where you release memory, file handles, or other external resources safely.
5. **Dynamic Assignment of Metamethods**
    - `Smeta_set(obj, name, fn)` allows you to assign functions dynamically using string names like `"__add__"` or `"__tostring__"`.
    - `Sobj_get_metamethod(obj, name)` lets the runtime retrieve the assigned function to perform the operation.
6. **Automatic Initialization**
    - `Smeta_new()` allocates a new `Smeta` structure and initializes all metamethod pointers to `NULL`.
    - Metamethods are assigned only when needed, ensuring minimal overhead for objects that don't use them.

After creating a custom userdata type in Suny, the next essential step is to **register metamethods**. This process links your low-level C functions to high-level operations in Suny scripts, giving your userdata meaningful behavior.

In Suny, metamethods are **function pointers** stored inside a `struct Smeta` associated with your userdata (`Sobj`). Each pointer corresponds to a specific operation, such as addition, subtraction, printing, or destruction.

Step 1. Create a Metamethod Function in C

To make your custom userdata support operations like addition, you need to **write a C function that implements the desired behavior**. This function will be called automatically by Suny whenever the corresponding operation is used on your userdata objects.

Let's continue the program we did before, now we creat a function `Svector_add`:

```
...

struct Sobj* Svector_add(struct Sobj* a, struct Sobj* b) {
    struct Svector* va = get_userdata(a)
    struct Svector* vb = get_userdata(b)

    int x1 = va->x
    int y1 = va->y
    int x2 = vb->x
    int y2 = vb->y

    struct Svector* vector = Svector_new(x1 + x2, y1 + y2);
    struct Sobj* value = Sobj_make_userdata(vector);

    value->meta = a->meta;

    return value;
}
```

Explanation:

1. **Purpose**
    - `Svector_add` defines **how two vector userdata objects behave when the `+` operator is used** in Suny.
    - Without this function, performing `vec1 + vec2` would fail or produce meaningless output because Suny doesn't know how to interpret raw userdata.
2. **Retrieve the internal C structs**

    ```
    struct Svector* va = get_userdata(a);
    struct Svector* vb = get_userdata(b);
    ```

    - Each userdata object wraps a pointer to a C struct (`Svector`).
    - `get_userdata()` extracts that pointer so the function can access `x` and `y` values.
3. **Extract vector components**

    ```
    int x1 = va->x;
    int y1 = va->y;
    int x2 = vb->x;
    int y2 = vb->y;
    ```

    - These integers represent the actual coordinates stored in each vector.
    - We read them separately so we can perform arithmetic operations.

4. **Create a new vector with the sum**

   ```
   struct Svector* vector = Svector_new(x1 + x2, y1 + y2);
   ```

   - This creates a new `Svector` struct holding the summed coordinates.
   - `Svector_new` handles memory allocation and initializes the struct properly.

5. **Wrap the new vector in Suny userdata**

   ```
   struct Sobj* value = Sobj_make_userdata(vector);
   ```

   - The new `Svector` is wrapped into a Suny `Sobj` so that it **participates in garbage collection** and can interact with Suny scripts.
   - This ensures memory safety and reference counting.

6. **Copy metamethods**

   ```
   value->meta = a->meta;
   ```

   - We assign the same `meta` from `a` to the new object.
   - This allows the new vector to inherit the **operator behaviors, printing rules, and other metamethods** from the original vector.

7. **Return the new Suny object**

   - Finally, the new userdata object is returned, ready for use in the script:

     ```
     let vec3 = vec1 + vec2
     ```

   - The runtime will call this function automatically to compute the result of the `+` operator.

Lets make more function, why not?

```
...

struct Sobj* Svector_sub(struct Sobj* a, struct Sobj* b) {
    struct Svector* va = get_userdata(a)
    struct Svector* vb = get_userdata(b)

    int x1 = va->x
    int y1 = va->y
    int x2 = vb->x
    int y2 = vb->y

    struct Svector* vector = Svector_new(x1 - x2, y1 - y2);
    struct Sobj* value = Sobj_make_userdata(vector);

    value->meta = a->meta;

    return value;
}

struct Sobj* Svector_print(struct Sobj* obj) {
    struct Svector* vector = get_userdata(obj)
    printf("vector(%d, %d)", vector->x, vector->y);
    return null_obj;
}

struct Sobj* Svector_free(struct Sobj* obj) {
    struct Svector* vector = get_userdata(obj)
    free(vector)
    return null_obj;
}
```

Step 2: Register the Metamethod

Defining the metamethod function in C is only the **first half of the process**. To make Suny actually use your `Svector_add` function when `+` is called, you need to **register it** with the vector's metadata (`Smeta`).

Every userdata type in Suny can have a `meta` field that points to an `Smeta` structure. This `Smeta` holds function pointers for all available metamethods. By assigning

your custom function (`Svector_add`) to the `mm_add` field, you tell the runtime exactly how to handle the `+` operator for this userdata type.

Example: Registering the `+` Operator

```
// Allocate and configure a new meta object for vectors
struct Smeta* vector_meta = Smeta_new();

// Register the addition metamethod
vector_meta->mm_add = Svector_add;

// Optionally register a string representation for debugging/printing
vector_meta->mm_tostring = Svector_tostring;

// Create a vector userdata object and attach the meta
struct Svector* v1 = Svector_new(1, 2);
struct Sobj* obj1 = Sobj_make_userdata(v1);
obj1->meta = vector_meta;

struct Svector* v2 = Svector_new(3, 4);
struct Sobj* obj2 = Sobj_make_userdata(v2);
obj2->meta = vector_meta;

// Now Suny can evaluate obj1 + obj2 by calling Seval_add()
```

Explanation

1. **Create a new meta structure**

   ```
   struct Smeta* vector_meta = Smeta_new();
   ```

   - `Smeta_new()` initializes a fresh meta object with all metamethod pointers set to `NULL` .
   - This ensures unused operations fall back to defaults (usually an error).

2. **Assign the addition metamethod**

   ```
   vector_meta->mm_add = Svector_add;
   ```

   - Here we explicitly register our custom `Svector_add` function.
   - Now, whenever two vector objects are added, Suny's runtime will call this function automatically.

3. **(Optional) Add more metamethods**

   ```
   vector_meta->mm_tostring = Svector_tostring;
   ```

   - For usability, it's common to also register `__tostring__` so that printing vectors is human-readable.
   - Example: `print(vec1)` → `(1, 2)` instead of showing a raw pointer.

4. **Attach meta to objects**

   ```
   obj1->meta = vector_meta;
   obj2->meta = vector_meta;
   ```

   - This step links the C userdata with the metamethods we defined.
   - Multiple objects can share the same `meta` , so you don't have to duplicate metamethods for every instance.

Register metamethod to my own datatype

Now we know how **Metamethods** work. We will using **Metamethods** to register to my own datatype Back to the previous code, at `struct Sobj* Sobj_make_vector(struct Sframe* frame) {...}` we add

```
struct Sobj* Sobj_make_vector(struct Sframe* frame) {
    struct Sobj* Sx = Sframe_pop(frame);
    struct Sobj* Sy = Sframe_pop(frame);

    float x = ValueOf(Sx);
    float y = ValueOf(Sy);

    struct Svector* vector = Svector_new(x, y);
    struct Sobj* obj = Sobj_make_userdata(vector);

    Smeta_set(obj, "__add__", Svector_add);
    Smeta_set(obj, "__add__", Svector_add);
    Smeta_set(obj, "__tostring__", Svector_print);
    Smeta_set(obj, "__free__", Svector_free);

    return obj;
}

SUNY_API struct Sframe* Smain(struct Sframe* frame, struct Scompiler* compiler) {
    SunyInitialize_c_api_func(frame, compiler, 20, "vector", 2, Sobj_make_vector);
    return frame;
}
```

Now we compile it and run

```
import "vector"

let vec1 = vector(2, 2)
let vec2 = vector(3, 3)
print(vec1 + vec2)
```

```
C:/> suny main.suny
vector(5, 5)
C:/>
```

Summary

In this step, we learned how to **register metamethods** for custom userdata types in Suny.

- First, we created a **metamethod function in C** (e.g., `Svector_add`) that implements the desired behavior.
- Then, we **registered** it in the `Smeta` structure using either direct assignment (`vector_meta->mm_add = Svector_add;`) or the helper function `Smeta_set(obj, "__add__", Svector_add);`.
- Finally, we **attached the meta** to our userdata objects so that Suny's runtime knows how to handle operations on them.

Once registered, our `vector` userdata behaves like a **first-class Suny type**:

- It can be added with `+` (`vec1 + vec2`)
- It can be printed with a human-readable format (`print(vec1)`)
- It can be safely destroyed with `__free__`

This demonstrates how **metamethods act as the bridge** between low-level C code and high-level Suny scripts, giving custom types natural and safe behavior.

---

12. Using Suny C-API to make library

In Suny, you can make your own libraries in two different ways:

1. Pure Suny Library

A **Pure Suny Library** is simply a collection of `.suny` files that contain functions, classes, or modules written directly in the Suny language. These libraries can be imported into other Suny programs without any additional setup.

Advantages:

- **Simplicity** – You only need to know Suny, no extra tools or languages required.
- **Portability** – Since it is written in Suny, it works on every system where the Suny runtime is available.
- **Rapid Development** – Perfect for sharing common functions, utilities, or algorithms.

Example:

```
# math.suny

function add(a, b) do
    return a + b
end
```

Now you can import it anywhere:

```
include "math.suny"

print(add(1, 2))
```

---

2. C Library via Suny C-API

For more advanced use cases, Suny provides a **C-API** that lets you extend the language with functions written in C. This is useful when you need:

- **High performance** – heavy calculations can be written in optimized C code.
- **System access** – interact with operating system APIs or hardware.
- **Integration** – connect Suny with existing C libraries, networking, graphics, or databases.

With the C-API, you can expose C functions as if they were native Suny functions. When compiled into a shared library (`.dll` on Windows, `.so` on Linux), Suny can `import` them just like normal Suny modules.

Example in C (a simple add function):

```
#include <Suny.h>

struct Sobj* Sadd(struct Sframe* frame) {
    struct Sobj* a = Sframe_pop(frame);
    struct Sobj* b = Sframe_pop(frame);

    float value = ValueOf(a) + ValueOf(b);
    return Sobj_set_int(value);
}

SUNY_API struct Sframe* Smain(struct Sframe* frame, struct Scompiler* compiler) {
    SunyInitialize_c_api_func(frame, compiler, 20, "add", 2, Sadd);
    return frame;
}
```

Compile it into `math.dll` and import it to `.suny` program

```
import "math" # Suny will automatic understand that this file is "math.dll"

print(add(1, 2)) # 3
```

Explain and how to make C Suny library

When you want to extend Suny with custom `C` code, you must follow the Suny C-API function structure. This allows your `C` functions to behave like built-in Suny functions, making them callable from Suny scripts without extra glue code.

A Suny function written in C always has this form:

```
SUNY_API struct Sobj* function_name(struct Sframe* frame) {
    // Get arguments (from right to left)
    struct Sobj* arg1 = Sframe_pop(frame); // first argument
    struct Sobj* arg2 = Sframe_pop(frame); // second argument
    struct Sobj* arg3 = Sframe_pop(frame); // third argument
    // ...

    // Do your calculation
    float result = ValueOf(arg1) + ValueOf(arg2) + ValueOf(arg3);

    // Wrap result into Suny object
    struct Sobj* value = Sobj_set_int(result);

    // Return it (Suny will push this onto the stack)
    return value;
}
```

Explanation:

- **`struct Sframe* frame`** → gives access to the Suny stack for this call.
- **`Sframe_pop(frame)`** → retrieves arguments passed from Suny.
- **`ValueOf(obj)`** → converts `Sobj` into a raw C value (like `int` or `float`).
- **`Sobj_set_int(result)`** → wraps a C integer back into a Suny object.
- **`return value`** → the value is returned to Suny and pushed onto the stack.

Every library must have an **entry point** called `Smain` where you register your functions:

```
SUNY_API struct Sframe* Smain(struct Sframe* frame, struct Scompiler* compiler) {
    // Register function "sum3" that takes 3 arguments
    SunyInitialize_c_api_func(frame, compiler, 20,    "sum3",   3,    function_name);
    //             ^         ^       ^      ^
    //             address    name  args count  point to C function
    return frame;
}
```

- `20` → Address of the function
- `"sum3"` → the name used inside Suny code.
- `3` → number of arguments expected.
- `function_name` → the C function you defined.
- `SunyInitialize_c_api_func` using this function to register the function

Compile Into a Library

Suny provides **`libSuny.a`**, which is the **static library** containing all the **core runtime functions** of the Suny language. When you create your own C extension or Suny library, you must **link against this file**, because it includes:

- The **Suny Virtual Machine runtime**
- Core object system (`Sobj`, `Sframe`, etc.)
- Memory management and garbage collection hooks
- Stack operations (`Sframe_pop`, `Sframe_push`, etc.)
- Utility helpers (`Sobj_set_int`, `Sobj_make_bool`, `Sobj_make_str`, etc.)
- Function registration and metadata system

Without linking `libSuny.a`, your C functions cannot communicate with the Suny runtime.

Compile it into a **shared library** (`.dll` on Windows)

```
gcc -shared my_math.c -o my_math.dll -I C:\Suny -L C:\Suny -lSuny
```

Then import it:

```
import "my_math"

print(sum3(1, 2, 3)) # 6
```

✅ Summary

Creating a **C library for Suny** allows you to extend the language with **high-performance algorithms**, **system-level functionality**, and **third-party integrations** that would otherwise be difficult or slow in pure Suny code.

The process can be broken down into these key steps:

1. **Understand the Suny C-API**
    - Every Suny C function follows the standard structure: it receives a `struct Sframe*` (argument stack) and returns a `struct Sobj*` (result).
    - Use Suny helpers like `Sframe_pop`, `Sframe_push`, `Sobj_set_int`,... etc. to work safely with Suny objects.
2. **Link with `libSuny.a`**
    - This core library provides access to the Suny runtime, garbage collection, stack operations, and object system.
    - Without linking against it, your extension cannot interact with the Suny VM.
3. **Compile to a Shared Library**
    - Use `gcc` or another C compiler to produce a `.dll` (Windows)
    - Always include the Suny headers (`-I`) and link against the core library (`-lSuny`).
4. **Register Functions with `SunyInitialize_c_api_func`**
    - Inside your entry point function (e.g., `Smain`), register each custom function with Suny.
    - This step tells the runtime the function's **name**, **argument count**, and **pointer to the C implementation**.
5. **Import and Use in Suny Code**
    - Load your compiled library in a `.suny` script with `import "your_lib"`.
    - Call the registered C functions just like built-in Suny functions.

 With this workflow, you can design powerful Suny modules that **combine the efficiency of C** with the **ease of scripting in Suny**. This makes Suny an extensible and flexible platform for **scientific computing, system programming, graphics, networking**, and more.

---

13. Virtual machine

1. Overview

The **Suny Virtual Machine (VM)** is the **core execution engine** of the language. It is responsible for taking the compiled **Suny bytecode** and running it step by step, ensuring that programs behave as intended. Suny's VM follows the **stack-based model**, which means that most operations are performed by **pushing values onto a stack** and then manipulating them with instructions.

Instead of executing high-level Suny code directly, the source code is first parsed and compiled into **bytecode instructions**. These instructions are compact, low-level operations such as:

- **PUSH_FLOAT** (push a number, string, or object onto the stack)
- **BINARY_ADD, BINARY_SUB, BINARY_MUL, BINARY_DIV** (perform arithmetic using the top elements of the stack)
- **FUNCTION_CALL** (invoke a function)
- **JUMP_TO, POP_JUMP_IF_FALSE** (control flow for loops and conditionals)
- **LOAD_GLOBAL / STORE_GLOBAL** (access variables and memory)

The VM then runs in a **fetch–decode–execute loop**, where it:

1. **Fetches** the next instruction from the bytecode.
2. **Decodes** the instruction to determine what operation to perform.
3. **Executes** the instruction by manipulating the stack, memory, or control flow.

Because Suny uses a stack-based VM, it is **simpler to implement** compared to register-based VMs, while still being flexible enough to support advanced features like:

- Functions and recursion
- Object management
- Garbage collection
- User-defined libraries
- Game rendering and input handling (via SGL – Suny Game Library)

This design makes Suny similar in architecture to other stack-based languages like **Python, Lua, and Java's JVM**, but tailored to remain lightweight, educational, and extensible.

In short, the Suny VM is what makes the language **portable and consistent**: the same Suny program can run on any system that provides the VM, without modification.

---

2. Execution Model

The **execution model** of the Suny Virtual Machine describes how bytecode instructions are processed at runtime. Like most stack-based interpreters, Suny relies on a **continuous loop** that drives program execution, often referred to as the **fetch–decode–execute cycle**.

1. Program Startup

When a Suny program is compiled, it produces a sequence of **bytecode instructions**. These instructions are loaded into a structure called a **frame** (representing the execution context). The VM then initializes:

- A **program counter (PC)** to track the current instruction.
- A **value stack** to hold operands and intermediate results.
- A **global scope** for variables and functions.
- A **call stack** for function invocation and return management.

---

2. The Execution Loop

At the heart of the VM is the main loop:

1. **Fetch** – The VM reads the next instruction from the bytecode using the PC.
2. **Decode** – The instruction is matched to its operation (e.g., `BINARY_ADD`, `PUSH_STRING`, `JUMP_IF_FALSE`).
3. **Execute** – The corresponding operation is performed:
    - **Stack operations** push or pop values.
    - **Arithmetic operations** consume operands and push results.
    - **Control flow operations** change the PC to implement conditionals and loops.
    - **Function operations** create new frames on the call stack.

After each execution, the PC is advanced (unless changed by a jump), and the cycle repeats until either:

- A `RETURN` instruction is reached, or
- The program ends with `PROGRAM_END`.

---

3. Stack-Oriented Evaluation

Since Suny is **stack-based**, nearly all instructions operate on the value stack:

- Example: To compute `3 + 5`, the compiler generates:
    - `PUSH_FLOAT 3`
    - `PUSH_FLOAT 5`
    - `BINARY_ADD`
- At runtime, `BINARY_ADD` pops the top two values, computes the sum, and pushes the result back.

This approach keeps the VM **compact** and reduces the need for complex register management.

---

4. Function and Scope Management

When a function is called:

- A new **frame** is created with its own local variables and stack.
- The current PC and execution state are saved on the **call stack**.
- After the function returns, control is passed back to the previous frame.

This enables **recursion**, **nested functions**, and structured scope handling.

---

5. Error and Halt Conditions

The VM halts execution when:

- It reaches a `PROGRAM_END` instruction.
- A runtime error occurs (e.g., invalid operation, undefined variable).
- A fatal error in memory allocation or garbage collection prevents further execution.

Error handling ensures the VM can report issues gracefully, without corrupting program state.

3. Instruction Set (Opcodes)

3.1 Opcode review

Below is the **official opcode set** for the Suny Virtual Machine. Each instruction is defined as a **1-byte code** (e.g., `\x01`, `\x02`), with optional arguments following it in the bytecode stream.

---

1. Stack & Data Push

Opcode	Mnemonic	Description	Stack Effect
		Push a floating-point	

Opcode	Mnemonic	Description	Stack Effect
`\x01`	`PUSH_FLOAT`	number	`- → value`
`\x22`	`PUSH_STRING`	Push a string object	`- → string`
`\x38`	`LOAD_TRUE`	Load constant `true`	`- → true`
`\x39`	`LOAD_FALSE`	Load constant `false`	`- → false`

---

2. Arithmetic Operations

Opcode	Mnemonic	Description	Stack Effect
`\x02`	`BINARY_ADD`	Add two top values	`a, b → (a+b)`
`\x03`	`BINARY_SUB`	Subtract (second – top)	`a, b → (a-b)`
`\x04`	`BINARY_MUL`	Multiply two values	`a, b → (a*b)`
`\x05`	`BINARY_DIV`	Divide (second ÷ top)	`a, b → (a/b)`

---

3. Comparison Operations

Opcode	Mnemonic	Description	Stack Effect
`\x12`	`BINARY_BIGGER`	Compare greater than (`>`)	`a, b → (a>b)`
`\x13`	`BINARY_SMALLER`	Compare less than (`<`)	`a, b → (a<b)`
`\x14`	`BINARY_EQUAL`	Equality (`==`)	`a, b → (a==b)`
`\x15`	`BINARY_BIGGER_EQUAL`	Greater or equal (`>=`)	`a, b → (a>=b)`
`\x16`	`BINARY_SMALLER_EQUAL`	Less or equal (`<=`)	`a, b → (a<=b)`
`\x17`	`BINARY_NOT_EQUAL`	Not equal (`!=`)	`a, b → (a!=b)`

---

4. Logic Operations

Opcode	Mnemonic	Description	Stack Effect		
`\x42`	`AND_LOG`	Logical AND	`a, b → a&&b`		
`\x43`	`OR_LOG`	Logical OR	`a, b → a		b`
`\x44`	`NOT_LOG`	Logical NOT	`a → !a`		

---

5. Program Control

Opcode	Mnemonic	Description
`\x10`	`PROGRAM_START`	Entry point marker
`\x11`	`PROGRAM_END`	End of program
`\x32`	`EXIT_PROGRAM`	Exit immediately

Opcode	Mnemonic	Description
`\x33`	`STOP_PROGRAM`	Stop execution (can be resumed)
`\x45`	`SKIP`	Skip this instruction

---

6. Jump & Flow Control

Opcode	Mnemonic	Description
`\x25`	`POP_JUMP_IF_TRUE`	Pop stack; jump if true
`\x26`	`POP_JUMP_IF_FALSE`	Pop stack; jump if false
`\x27`	`JUMP_IF_TOP_IS_TRUE`	Jump if top is true (without popping)
`\x28`	`JUMP_IF_TOP_IS_FALSE`	Jump if top is false (without popping)
`\x29`	`JUMP_FORWARD`	Jump forward N instructions
`\x30`	`JUMP_BACKWARD`	Jump backward N instructions
`\x31`	`SKIP_TO_INDEX`	Skip to a specific instruction index
`\x27`	`JUMP_TO`	Jump to absolute index
`\x28`	`ADD_LABEL`	Add a label for future jumps

---

7. Functions

Opcode	Mnemonic	Description
`\x18`	`MAKE_FUNCTION`	Define a function object
`\x19`	`END_FUNCTION`	End function definition
`\x20`	`FUNCTION_CALL`	Call function with N args
`\x21`	`RETURN_TOP`	Return top of stack to caller

---

8. Variables

Opcode	Mnemonic	Description
`\x08`	`LOAD_GLOBAL`	Load global variable
`\x09`	`STORE_GLOBAL`	Store into global variable

---

9. Data Structures

Opcode	Mnemonic	Description
`\x34`	`LOAD_ITEM`	Load element from list/string
`\x35`	`STORE_ITEM`	Store element in list/string

Opcode	Mnemonic	Description
`\x36`	`BUILD_LIST`	Build list from N stack values
`\x37`	`LEN_OF`	Get length of list/string

---

10. Objects & Classes

Opcode	Mnemonic	Description
`\x40`	`CLASS_BEGIN`	Start class definition
`\x41`	`CLASS_END`	End class definition
`\x46`	`LOAD_ATTR`	Load attribute from object
`\x47`	`STORE_ATTR`	Store attribute into object
`\x48`	`LOAD_METHOD`	Load method reference from object/class

---

11. I/O

Opcode	Mnemonic	Description
`\x06`	`PRINT`	Print top stack value
`\x07`	`POP_TOP`	Discard top stack value

3.2 How each opcode work?

Here is how each opcode in the Suny Virtual Machine (SVM) works. In order to keep the system well-organized, all available opcodes are divided into 11 functional groups. Each group corresponds to a different category of behavior inside the VM, such as stack manipulation, arithmetic operations, control flow, function handling, or class/object management.

By grouping the opcodes, the documentation makes it easier for developers to quickly understand not only what each instruction does, but also how they cooperate to execute a Suny program step by step. For example, stack operations provide the foundation, arithmetic and logical operators extend computational power, and control-flow instructions enable conditional execution and looping. Together, these groups form the complete instruction set of SVM and define the execution model of the language.

1. Stack & Data Push

This group contains the most fundamental opcodes in the Suny Virtual Machine. Their main responsibility is to push values onto the stack, which acts as the core working area of the VM. Every calculation, comparison, or function call in Suny begins with having the right values placed on the stack. Without these push operations, higher-level instructions (like arithmetic or control flow) would not have the data they need to operate.

1.1. `PUSH_FLOAT`

When the Suny Virtual Machine executes the PUSH_FLOAT opcode, it must read a floating-point constant from the bytecode stream and place it on the stack. The implementation looks like this:

```
 struct Sframe *
Svm_evalutate_PUSH_FLOAT(struct Sframe *frame) {

    // Read 4 bytes (little-endian order) from bytecode
    byte_t b1 = get_next_code(frame);
    byte_t b2 = get_next_code(frame);
    byte_t b3 = get_next_code(frame);
    byte_t b4 = get_next_code(frame);

    // Combine the 4 bytes into a 32-bit unsigned integer
    uint32_t i = (uint32_t)b1 |
                 ((uint32_t)b2 << 8) |
                 ((uint32_t)b3 << 16) |
                 ((uint32_t)b4 << 24);

    // Interpret those 4 bytes as a float
    float value;
    memcpy(&value, &i, sizeof(value));

    // Push the float value onto the stack
    Sframe_push_number(frame, value);

    return frame;
}
```

Step-by-Step Explanation:

1. **Fetch the Bytes**
    - The VM calls `get_next_code(frame)` 4 times, each retrieving one byte from the bytecode stream.
    - Since Suny encodes floats as 4-byte IEEE 754 binary format, these 4 bytes represent one float value.
2. **Reconstruct 32-bit Integer**
    - The 4 bytes are combined using bit shifting and OR operations to form a `uint32_t`.
    - This handles the correct **little-endian** byte order.
3. **Convert Integer to Float**
    - The raw 32-bit integer is reinterpreted as a `float` using `memcpy`.
    - This ensures no type-punning issues and works portably across compilers.
4. **Push onto the Stack**
    - Finally, the `Sframe_push_number(frame, value)` function pushes the decoded float onto the VM's runtime stack.
    - This value now becomes available for subsequent operations (`BINARY_ADD`, `STORE_GLOBAL`, etc.).

Effect on the VM:

- **Before Execution**: Stack is `[ ... ]`
- **Bytecode**: `PUSH_FLOAT 3.14` (encoded as 4 bytes)
- **After Execution**: Stack is `[ ..., 3.14 ]`

1.2. `PUSH_STRING`

The `PUSH_STRING` instruction is used to load a string constant from the bytecode and place it onto the VM stack.

Execution Steps

1. **Read the string length**
    - The instruction first reads the string size (in bytes) from the bytecode stream.
    - Depending on the implementation, this can be a single byte (0–255) or multiple bytes (for longer strings).
2. **Allocate memory**
    - A buffer of `size + 1` is allocated to hold the string characters plus the null terminator (`'\0'`).
3. **Copy characters**
    - Each character is read from the bytecode and copied into the buffer.
4. **Null-terminate**

- The buffer is terminated with `'\0'` so it can safely function as a C-style string.

5. **Push onto the stack**
    - The resulting string object is pushed onto the current frame's stack.
    - Memory ownership is transferred to the VM, which must eventually free or garbage-collect the string.

C Implementation

```
 struct Sframe *
Svm_evaluate_PUSH_STRING(struct Sframe *frame) {
    // 1. Read size (1 byte; upgrade to 4 bytes if needed)
    int size = get_next_code(frame);

    // 2. Allocate buffer (+1 for null terminator)
    char* buff = malloc(size + 1);
    if (!buff) {
        // Handle allocation failure
        fprintf(stderr, "Memory allocation failed in PUSH_STRING\n");
        exit(1);
    }

    // 3. Copy string characters
    for (int i = 0; i < size; ++i) {
        buff[i] = get_next_code(frame);
    }

    // 4. Null-terminate
    buff[size] = '\0';

    // 5. Push to stack (VM owns 'buff')
    Sframe_push_string(frame, buff, size);

    return frame;
}
```

Example Bytecode Encoding

Suppose we want to push the string `"hi"`:

```
PUSH_STRING  2  'h'  'i'
```

- `2` = length of the string
- `'h'`, `'i'` = characters
- The VM reads this and pushes `"hi"` onto the stack.

1.3. `LOAD_TRUE` and `LOAD_FALSE`

This two opcodes push **boolen** value into the stack

```
struct Sframe *
Svm_evaluate_LOAD_TRUE
(struct Sframe *frame) {
    Sframe_push_bool(frame, 1);
    return frame;
}

struct Sframe *
Svm_evaluate_LOAD_FALSE
(struct Sframe *frame) {
    Sframe_push_bool(frame, 0);
    return frame;
}
```

---

2. Arithmetic Operations

2.1. `BINARY_ADD`, `BINARY_DIV`, `BINARY_MUL`, `BINARY_SUB`

1. Pop **two values** from the stack (let's call them `rhs` and `lhs` where `rhs` is the top of stack).
2. Depending on the opcode, evaluate:
    - `BINARY_ADD` → call `Seval_add(lhs, rhs)` (`lhs + rhs`)
    - `BINARY_DIV` → call `Seval_div(lhs, rhs)` (`lhs / rhs`)
    - `BINARY_MUL` → call `Seval_mul(lhs, rhs)` (`lhs * rhs`)
    - `BINARY_SUB` → call `Seval_sub(lhs, rhs)` (`lhs - rhs`)
3. Push the result back onto the stack.

---

3. Comparison Operations

3.1. `BINARY_BIGGER`, `BINARY_SMALLER`, `BINARY_EQUAL`, `BINARY_BIGGER_EQUAL`, `BINARY_SMALLER_EQUAL`, `BINARY_NOT_EQUAL`

1. Pop **two values** from the stack (`rhs`, `lhs`).
2. Depending on the opcode, evaluate:
    - `BINARY_BIGGER` → call `Seval_bigger(lhs, rhs)` (`lhs > rhs`)
    - `BINARY_SMALLER` → call `Seval_smaller(lhs, rhs)` (`lhs < rhs`)
    - `BINARY_EQUAL` → call `Seval_equal(lhs, rhs)` (`lhs == rhs`)
    - `BINARY_BIGGER_EQUAL` → call `Seval_bigger_and_equal(lhs, rhs)` (`lhs >= rhs`)
    - `BINARY_SMALLER_EQUAL` → call `Seval_smaller_and_equal(lhs, rhs)` (`lhs <= rhs`)
    - `BINARY_NOT_EQUAL` → call `Seval_not_equal(lhs, rhs)` (`lhs != rhs`)
3. Push the result (boolean) back onto the stack.

This function is a part of SVM evaluate opcodes `BINARY_SMALLER`, `BINARY_BIGGER`, ... `BINARY_NOT_EQUAL` and `BINARY_ADD`, ... `BINARY_DIV`

```
struct Sframe *
Svm_evalutate_BINARY_OPER
(struct Sframe *frame, byte_t op) {
    struct Sobj *obj2 = Sframe_pop(frame);
    struct Sobj *obj1 = Sframe_pop(frame);

    float value1 = obj1->value->value;
    float value2 = obj2->value->value;

    struct Sobj *obj = NULL;

    switch (op) {
        case BINARY_ADD: {
            obj = Seval_add(obj1, obj2);
            break;
        } case BINARY_SUB: {
            obj = Seval_sub(obj1, obj2);
            break;
        } case BINARY_MUL: {
            obj = Seval_mul(obj1, obj2);
            break;
        } case BINARY_DIV: {
            obj = Seval_div(obj1, obj2);
            break;
        } case BINARY_BIGGER : {
            obj = Seval_bigger(obj1, obj2);
            break;
        } case BINARY_SMALLER : {
            obj = Seval_smaller(obj1, obj2);
            break;
        } case BINARY_EQUAL : {
            obj = Seval_equal(obj1, obj2);
            break;
        } case BINARY_NOT_EQUAL : {
            obj = Seval_not_equal(obj1, obj2);
            break;
        } case BINARY_BIGGER_EQUAL : {
            obj = Seval_bigger_and_equal(obj1, obj2);
            break;
        } case BINARY_SMALLER_EQUAL : {
            obj = Seval_smaller_and_equal(obj1, obj2);
            break;
        } default: {
            break;
        }
    }

    Sgc_dec_ref(obj1, frame->gc_pool);
    Sgc_dec_ref(obj2, frame->gc_pool);

    Sframe_push(frame, obj);
    return frame;
}
```

---

4. Logic Operations

4.1. `AND_LOG`, `OR_LOG`, `NOT_LOG`

- `AND_LOG`
    1. Pop **two values** from the stack (`rhs` , `lhs`).
    2. If both `lhs` and `rhs` evaluate to **true**, push `true` . Otherwise, push `false` .
- `OR_LOG`

1. Pop **two values** from the stack (`rhs` , `lhs`).
2. If **at least one** of `lhs` or `rhs` is true, push `true` . Otherwise, push `false` .

- `NOT_LOG`

1. Pop **one value** from the stack (`val`).
2. If `val` is true, push `false` . If `val` is false, push `true` .

---

5. Program Control

5.1. `PROGRAM_START`

- Marks the **entry point** of the program.
- Tells the VM where the **main program execution** begins.
- At this point the VM:

1. Resets and initializes the **stack**, **globals**, and runtime state.
2. Sets the **instruction pointer (IP)** to the start of the program code.

5.2. `PROGRAM_END`

- Marks the **end of execution** for the program.
- When the VM encounters this opcode:

1. Stop executing bytecode.
2. Free or clean up resources (stack frames, heap allocations if needed).
3. Return control back to the host environment (e.g., exit status).

5.3. `EXIT_PROGRAM`

- Forces the VM to **immediately terminate execution**, no matter where it is in the program.
- Unlike `PROGRAM_END` (which is the natural end of a program), `EXIT_PROGRAM` can appear **anywhere** in the bytecode.
- When executed:

1. Clear the stack and free resources.
2. Stop execution instantly.
3. Optionally return an **exit status** (e.g., success/failure code) to the host environment.

5.4. `STOP_PROGRAM`

- Suspends the program's execution at the current point.
- Unlike `EXIT_PROGRAM` (which terminates completely), `STOP_PROGRAM` **halts the VM loop** but keeps the runtime state (stack, globals, memory) intact.
- This allows:

1. Debuggers or external tools to inspect the VM state.
2. The program to be **resumed later** from the same point if supported.

- When executed:
  - The instruction pointer (IP) stops advancing.
  - Control is handed back to the host without destroying runtime data.

---

6. Jump & Flow Control

6.1 `ADD_LABEL`

The `ADD_LABEL` opcode is a **special instruction**. Before the Virtual Machine begins executing the main program, it performs a preprocessing pass over all opcodes.

Whenever the VM encounters an `ADD_LABEL 0x..` , it associates the given label address `0x..` with the **current instruction index** and stores this mapping inside the `Slabel_map` .

The `Slabel_map` is a member of the `Sframe` structure. It must be initialized before execution starts using:

```
frame->f_label_map = Slabel_map_set_program(code);
```

This ensures that all labels are registered and accessible during execution.

6.2 `POP_JUMP_IF_TRUE`, `POP_JUMP_IF_FALSE`

- **`POP_JUMP_IF_TRUE 0x..`** Pops the top value from the stack. If the value is `true` (`1`), the instruction pointer jumps to the given address `0x..`.
- **`POP_JUMP_IF_FALSE 0x..`** Pops the top value from the stack. If the value is `false` (`0`), the instruction pointer jumps to the given address `0x..`.

These opcodes are commonly used for conditional branching in the program flow.

6.3 `JUMP_IF_TOP_IS_TRUE`, `JUMP_IF_TOP_IS_FALSE`

- **`JUMP_IF_TOP_IS_TRUE 0x..`** Checks the value at the top of the stack **without popping it**. If the value is `true` (`1`), the instruction pointer jumps to the given address `0x..`.
- **`JUMP_IF_TOP_IS_FALSE 0x..`** Checks the value at the top of the stack **without popping it**. If the value is `false` (`0`), the instruction pointer jumps to the given address `0x..`.

Unlike `POP_JUMP_IF_TRUE` / `POP_JUMP_IF_FALSE`, these instructions **do not remove** the top stack value, making them useful when the condition needs to be reused later.

6.4 `JUMP_FORWARD`, `JUMP_BACKWARD`

- **`JUMP_FORWARD n`** Moves the instruction pointer forward by `n` steps (relative jump).
- **`JUMP_BACKWARD n`** Moves the instruction pointer backward by `n` steps (relative jump).

These opcodes are often used for implementing **loops** or skipping over code sections.

6.5 `SKIP_TO_INDEX`

- **`SKIP_TO_INDEX idx`** Moves the instruction pointer directly to the instruction at index `idx`. This is useful for quickly skipping over blocks of code during preprocessing or execution.

6.6 `JUMP_TO`

- **`JUMP_TO 0x..`** Performs an **absolute jump** to the address `0x..`. The address must correspond to a valid entry in the `Slabel_map` (created with `ADD_LABEL`).

This instruction is commonly used for **unconditional branching**, such as function calls, exits, or goto-style jumps.

---

7. Functions

7.1 `MAKE_FUNCTION`

`MAKE_FUNCTION <args_count>`

Creates a new **function object**. This opcode expects the function's **code block** (usually pushed earlier onto the stack) and the number of arguments it takes.

- **`<args_count>`** specifies how many arguments the function requires.
- The function object is then **pushed onto the stack**, so it can be stored in a variable or passed around like any other value.

This is the fundamental step for defining functions in the VM.

Here is source code and how its work:

```
struct Sframe *
Svm_evaluate_MAKE_FUNCTION
(struct Sframe *frame) {
    byte_t args_count = get_next_code(frame); // get the arguments count

    int code_size = 0;

    struct Scode *code = Scode_new(); // creat a new code object

    int func_level = 1; // function level for inner fucntion

    byte_t op = get_next_code(frame);

    while (1) {
        if (op == MAKE_FUNCTION) {
            func_level++;
        }

        if (op == END_FUNCTION) {
            func_level--;
        } else if (op == MAKE_FUNCTION) {
            func_level++;
        }

        if (op == END_FUNCTION && func_level == 0) break;

        PUSH(code, op);

        ++code_size;

        op = get_next_code(frame);
    }

    PUSH(code, END_FUNCTION);

    struct Sfunc *func = Sfunc_set(code, args_count, code_size); // set function with code size and argument count
    struct Sobj *f_obj = Sobj_set_func(func);

    Sframe_push(frame, f_obj);

    return frame;
}
```

7.3 `FUNCTION_CALL`

The `FUNCTION_CALL` opcode is used to invoke a function.

- It **pops** the top object from the stack.
- If the object is a **user-defined function object**, the VM creates a new frame and begins executing the function's bytecode.
- If the object is a **builtin object** (native function provided by the VM), the VM directly executes the corresponding C/C++ implementation and pushes the return value onto the stack.
- If the object is **not callable**, the VM either returns to the current frame or raises an error (depending on how strictly invalid calls are handled).

This instruction is central to enabling function execution, recursion, and integration with builtin/native functions.

When the VM encounters a `FUNCTION_CALL`, it does **not** directly read and execute the raw bytecode of the function object. Instead, it creates a **call context object** in Suny, named `Scall_context`.

The `Scall_context` acts as a container that holds all the necessary state for the function call, including:

- **Reference to the function object** – the bytecode, metadata, and argument count.
- **Caller frame** – a pointer to the current `Sframe` that initiated the call.
- **Callee frame** – a new execution frame created for the function, which has its own local variables, stack, and label map.
- **Arguments** – values popped from the caller's stack that match the function's `<args_count>`.
- **Return address** – the instruction index in the caller frame where execution will continue once the function finishes.

Once the `Scall_context` is prepared, the VM transfers control to the new frame. When the function completes (via `RETURN_VALUE`), the VM destroys the

`Scall_context` and resumes execution in the caller frame at the stored return address.

The `Scall_context` Structure

Defined in `Stype.h`, the `Scall_context` holds all the metadata and runtime state required to manage a single function call:

```
struct Scall_context {
    struct Sfunc *func;              // Pointer to the function object being executed

    struct Sframe *frame;            // The current stack frame of this call
    struct Sframe *main_frame;       // Reference to the main program's frame (caller)

    struct Sobj **local_t;           // Array of local variables or arguments

    struct Scode *code;              // Pointer to the function's compiled bytecode

    int args_index;                  // Index into the argument list
    int code_index;                  // Current instruction pointer within the function
    int stack_index;                 // Stack pointer for this call
    int local_index;                 // Local variable index

    struct Sobj *obj;                // Generic object reference used during execution
    struct Sobj *ret_obj;            // Return value, to be pushed back to the caller's stack
};
```

The `Scall_context` essentially works as a **bridge** between the **caller frame** and the **callee frame**, providing the runtime with enough information to:

- Initialize and run a function in its own isolated execution space.
- Track arguments, locals, and return values.
- Handle recursion (each call gets its own `Scall_context`).
- Switch safely back to the caller once the callee completes.

Executing a Function Call

When the VM encounters a `FUNCTION_CALL` opcode, it transfers execution to the function:

```
struct Sframe *
Svm_evaluate_FUNCTION_CALL(struct Sframe *frame) {
    // Step 1: Pop the top object from the stack
    struct Sobj *f_obj = Sframe_pop(frame);

    // Step 2: Check if the object is a builtin function
    if (f_obj->type == BUILTIN_OBJ) {
        // Call the native C API function directly
        Sframe_call_c_api_func(frame, load_c_api_func(f_obj));
        return frame;
    }

    // Step 3: Check if the object is a class
    if (f_obj->type == CLASS_OBJ) {
        // Instantiate a new object by copying the class definition
        struct Sobj *obj = Sobj_creat_a_copy_version_of_class(f_obj->f_type->f_class);
        // Push the new object instance onto the stack
        Sframe_push(frame, obj);
        return frame;
    }

    // Step 4: Otherwise, treat the object as a user-defined function
    struct Scall_context *context = Scall_context_new();

    // Initialize the call context (link caller frame and callee function)
    Scall_context_set_frame(context, frame, f_obj);

    // Execute the function in its own call context
    Svm_run_call_context(context);

    // Clean up the call context after execution
    Scall_context_free(context);

    // Decrement reference count for garbage collection
    Sgc_dec_ref(f_obj, frame->gc_pool);

    // Return control to the caller's frame
    return frame;
}
```

Execution Flow Breakdown

1. **Pop target object** – The VM pops the top of the stack to determine what is being called.
2. **Builtin function** – If it's a builtin, the VM resolves the associated C/C++ implementation and executes it directly.
3. **Class object** – If it's a class, the VM creates a new object instance and pushes it back to the stack.
4. **User-defined function** – For normal functions, the VM allocates a new `Scall_context`, sets up the call environment, and executes the bytecode of the function in a fresh frame.
5. **Return & cleanup** – Once the function finishes, the `Scall_context` is freed, the return object is pushed back to the caller's stack, and execution resumes from the saved return address.

---

8. Variables

Variables in the VM are divided into **global variables** and **local variables**. This section describes how the VM loads and stores values from these variable spaces.

Global variables live in a shared global scope accessible across functions and modules. Local variables are private to a specific function call (inside a stack frame).

8.1 `LOAD_GLOBAL`

Description: `LOAD_GLOBAL` loads a global variable from the global symbol table using its address/index. The value is then pushed onto the **operand stack**, making it available for further operations (arithmetic, function calls, comparisons, etc.).

Execution Steps:

1. Read the global variable address (`int`) from bytecode.
2. Retrieve the object stored at that address in the global scope.

3. Push the object's value onto the current stack.

C Implementation:

```
struct Sframe *
Svm_evalutate_LOAD_GLOBAL(struct Sframe *frame) {
    int address = get_next_code(frame);                // read global variable index
    struct Sobj *obj = Sframe_load_global(frame, address);

    Sframe_push(frame, obj->f_value);                  // push value to stack
    return frame;
}
```

Stack Behavior:

```
Before: [... ]
After:  [... , global[address]]
```

Example in Pseudo-Suny Code:

```
x = 10
print(x)
```

Generated Bytecode (simplified):

```
PUSH_FLOAT 10
STORE_GLOBAL 0
LOAD_GLOBAL 0
CALL_FUNCTION print
```

Execution:

- `STORE_GLOBAL 0` → saves `10` into global variable `x`.
- `LOAD_GLOBAL 0` → retrieves `10` from globals and pushes it onto the stack.
- `CALL_FUNCTION print` → consumes `10` from the stack and prints it.

8.2 `STORE_GLOBAL`

Description: `STORE_GLOBAL` pops a value from the stack and stores it into the global scope at the specified address. This opcode is responsible for assigning values to global variables.

Execution Steps:

1. Pop the top value from the operand stack.
2. Read the target global variable address from bytecode.
3. Store the value into the global variable table at that address.

C Implementation:

```
struct Sframe *
Svm_evalutate_STORE_GLOBAL(struct Sframe *frame) {
    struct Sobj *obj = Sframe_pop(frame);              // pop value from stack

    int address = get_next_code(frame);                // read global variable index
    Sframe_store_global(frame, address, obj, GLOBAL_OBJ);

    return frame;
}
```

Stack Behavior:

```
Before: [... , value]
After:  [...]
```

Example in Pseudo-Suny Code:

```
x = 42
y = x + 8
```

Generated Bytecode (simplified):

```
PUSH_FLOAT 42
STORE_GLOBAL 0         ; x = 42
LOAD_GLOBAL 0
PUSH_FLOAT 8
BINARY_ADD
STORE_GLOBAL 1         ; y = x + 8
```

Execution:

1. Push `42` and store it at global `x`.
2. Load `x`, push `8`, then add → result `50`.
3. Store `50` into global `y`.

---

9. Data Structures

The VM provides support for working with **lists**, **strings**, and **user-defined data objects**. These data structures can be manipulated using dedicated opcodes that allow **loading, storing, creating, and querying** their contents.

Opcode	Mnemonic	Description
`\x34`	`LOAD_ITEM`	Load an element from a list/string or user data object
`\x35`	`STORE_ITEM`	Store a value into a list/string
`\x36`	`BUILD_LIST`	Build a new list from N stack values
`\x37`	`LEN_OF`	Get the length of a list/string

9.1 `LOAD_ITEM`

Description: Retrieves an element at a given index from a **list**, **string**, or **user-defined object** and pushes it onto the stack.

Behavior:

- **List (`LIST_OBJ`)**: Returns the element at `index`.
- **String (`STRING_OBJ`)**: Returns the character at `index` as a new `CHAR_OBJ`.
- **User-defined object (`USER_DATA_OBJ`)**: Calls a custom indexing method if available.
- If the object type is unsupported, pushes `0` as a default value.

Stack Behavior:

```
Before: [... , target, index]
After:  [... , target[index]]
```

C Implementation:

```
struct Sframe *
Svm_evaluate_LOAD_ITEM(struct Sframe *frame) {
    struct Sobj *index = Sframe_pop(frame);
    struct Sobj *list = Sframe_pop(frame);

    if (list->type == LIST_OBJ) {
        if (index->value->value >= list->f_type->f_list->count) {
            printf("Error: index out of range\n");
            SUNY_BREAK_POINT;
            return frame;
        }
        struct Sobj *item = Slist_get(list->f_type->f_list, index->value->value);
        Sframe_push(frame, item);
    } else if (list->type == STRING_OBJ) {
        if (index->value->value >= list->f_type->f_str->size) {
            printf("Error: index out of range\n");
            SUNY_BREAK_POINT;
            return frame;
        }
        char c = list->f_type->f_str->string[index->value->value];
        struct Sobj *obj = Sobj_make_char(c);
        Sframe_push(frame, obj);
    } else if (list->type == USER_DATA_OBJ && list->meta && list->meta->mm_index) {
        struct Sobj *ret = list->meta->mm_index(list, index);
        Sframe_push(frame, ret);
    } else {
        Sframe_push(frame, Sobj_set_int(0));
    }

    Sgc_dec_ref(list, frame->gc_pool);
    Sgc_dec_ref(index, frame->gc_pool);
    return frame;
}
```

9.2 `STORE_ITEM`

Description: Stores a value into a list at a specified index.

Stack Behavior:

```
Before: [... , list, index, value]
After:  [...]
```

C Implementation:

```
struct Sframe *
Svm_evaluate_STORE_ITEM(struct Sframe *frame) {
    struct Sobj *value = Sframe_pop(frame);
    struct Sobj *index = Sframe_pop(frame);
    struct Sobj *list = Sframe_pop(frame);

    int index_value = index->value->value;
    struct Sobj *pre_item = list->f_type->f_list->array[index_value];

    list->f_type->f_list->array[index_value] = value;

    dec_ref(pre_item);
    dec_ref(index);
    inc_ref(value);

    Sgc_dec_ref(pre_item, frame->gc_pool);
    Sgc_dec_ref(index, frame->gc_pool);

    return frame;
}
```

9.3 `LEN_OF`

Description: Pushes the length of a **list** or **string** onto the stack.

Stack Behavior:

```
Before: [... , list/string]
After:  [... , length]
```

C Implementation:

```
struct Sframe *
Svm_evaluate_LEN_OF(struct Sframe *frame) {
    struct Sobj *list = Sframe_pop(frame);

    if (list->type == LIST_OBJ) {
        Sframe_push(frame, Sobj_set_int(list->f_type->f_list->count));
    } else if (list->type == STRING_OBJ) {
        Sframe_push(frame, Sobj_set_int(list->f_type->f_str->size));
    }

    Sgc_dec_ref(list, frame->gc_pool);
    return frame;
}
```

9.4 `BUILD_LIST`

Description: Creates a new list from **N values** popped from the stack and pushes the new list onto the stack.

Stack Behavior:

```
Before: [... , val1, val2, ..., valN]
After:  [... , [val1, val2, ..., valN]]
```

C Implementation:

```
struct Sframe *
Svm_evaluate_BUILD_LIST(struct Sframe *frame) {
    int size = get_next_code(frame);
    struct Slist *list = Slist_new();

    for (int i = 0; i < size; ++i) {
        struct Sobj *item = Sframe_pop(frame);
        inc_ref(item);
        Slist_add(list, item);
    }

    struct Sobj *obj = Sobj_make_list(list);
    Sframe_push(frame, obj);

    return frame;
}
```

Example in Pseudo-Suny Code

```
a = 10
b = 20
c = 30
lst = [a, b, c]    # BUILD_LIST 3

x = lst[1]          # LOAD_ITEM
lst[2] = 42         # STORE_ITEM

n = size(lst)       # LEN_OF
```

Simplified Bytecode:

```
PUSH_FLOAT 10
PUSH_FLOAT 20
PUSH_FLOAT 30
BUILD_LIST 3
STORE_GLOBAL 0       ; lst

LOAD_GLOBAL 0
PUSH_FLOAT 1
LOAD_ITEM
STORE_GLOBAL 1       ; x = lst[1]

LOAD_GLOBAL 0
PUSH_FLOAT 2
PUSH_FLOAT 42
STORE_ITEM           ; lst[2] = 42

LOAD_GLOBAL 0
LEN_OF
STORE_GLOBAL 2       ; n = size(lst)
```

---

4. Summary

The **Suny Virtual Machine (VM)** is the backbone of the language, responsible for executing bytecode instructions generated from Suny source code. It uses a **stack-based architecture**, where values are pushed and popped from a stack to perform computations, manage variables, and handle control flow.

Key points covered in this chapter:

- **Stack-Based Execution:** Most operations, including arithmetic, comparisons, and function calls, are performed using a stack.
- **Instruction Set:** The VM supports a variety of opcodes for **arithmetic, logic, jumps, function calls, and data manipulation**.
- **Variables:** The VM differentiates between **global** and **local variables**, providing efficient access and storage through dedicated opcodes.
- **Data Structures:** Lists, strings, and user-defined objects can be manipulated via `LOAD_ITEM`, `STORE_ITEM`, `BUILD_LIST`, and `LEN_OF`.

- **Function Calls:** User-defined and builtin functions are executed using a **call context (`Scall_context`)**, enabling recursion and isolated execution frames.
- **Error Handling:** The VM performs runtime checks (e.g., index out of range) to ensure safe and predictable execution.
- **Portability:** Suny programs run consistently across different systems as long as the VM is provided.

Overall, the Suny VM provides a **robust, flexible, and efficient runtime environment** , combining simplicity for educational purposes with enough power to support advanced language features, including user-defined libraries and the Suny Game Library (SGL).

---

14. Standard Libraries

Suny provides a collection of standard libraries that extend the core language with useful features. These libraries are always available and can be imported into your program to simplify common tasks such as mathematics, string processing, input/output, and more.

14.1 The `stdlib` Library

The **`stdlib`** library is one of the most powerful and essential components of Suny's standard libraries. It provides a rich collection of **mathematical functions** and **advanced datatypes** that allow developers to write complex programs with ease and efficiency.

Key Features

1. **Mathematical Functions** The `stdlib` library includes a wide range of mathematical functions commonly used in scientific and engineering applications. Examples include:
    - `sin(x)` – Compute the sine of an angle (in radians).
    - `cos(x)` – Compute the cosine of an angle.
    - `tan(x)` – Compute the tangent of an angle.
    - `cot(x)` – Compute the cotangent of an angle.

    These functions are optimized for performance and provide accurate results suitable for both everyday calculations and advanced computations.
2. **Datatypes Provided** In addition to math utilities, `stdlib` also extends the language with several important datatypes:
    - **Vector** – Represents a sequence of numbers with built-in operations like addition, subtraction, and dot product. Useful in linear algebra and graphics.
    - **Complex** – Provides native support for complex numbers, including arithmetic operations and magnitude/phase calculations.
    - **BigInt** – Arbitrary precision integer type for working with numbers larger than standard machine integers. Essential for cryptography, number theory, or very large computations.

Example Usage

```
import "stdlib"

# Using math functions
let angle = 1.5708   # approx π/2
let s = sin(angle)   # -> 1.0
let c = cos(angle)   # -> 0.0

# Working with vectors
let v1 = vector([1, 2, 3])
let v2 = vector([4, 5, 6])
let v3 = v1 + v2     # -> [5, 7, 9]

# Complex numbers
let z1 = complex(2, 3)    # 2 + 3i
let z2 = complex(1, -1)   # 1 - i
let z3 = z1 * z2                # -> 5 + i

# Big integers
let big = bigint("12345678901234567890")
let big2 = big * 100
```

Why `stdlib` Matters

The `stdlib` library is designed to be **general-purpose and foundational**, making it the go-to library for a wide variety of programming tasks. Whether you are writing simple scripts, performing scientific computations, or handling large-scale numerical problems, `stdlib` provides the building blocks you need.

14.2 The `random` Library

The `random` library provides functionality for generating random numbers and making random selections. It is useful in applications such as simulations, games, randomized testing, and procedural generation.

Unlike some libraries that are entirely written in Suny, the `random` library integrates with a **native DLL (`random.dll`)**, which provides the core random number generator. Suny code then builds convenient wrappers on top of this functionality.

Key Functions

1. `random()`
    - **Description:** Returns a floating-point number between `0.0` and `1.0`.
    - **Implementation:** Internally calls `Srandom` from the `random.dll`.
    - **Example:**

```
let x = random()
print(x)   # -> 0.731293...
```

2. `randint(a, b)`
    - **Description:** Returns a random integer between `a` and `b` (inclusive).
    - **Details:**
        - If `a > b`, the values are swapped to ensure correctness.
        - Uses `random()` internally to generate the result.
    - **Example:**

```
let n = randint(1, 6)
print(n)   # -> a number between 1 and 6, simulating a dice roll
```

3. `choice(l)`
    - **Description:** Returns a random element from the list `l`.
    - **Details:**
        - Uses `randint` to pick a valid index from the list.
        - Works with any list, regardless of element type.
    - **Example:**

```
let fruits = ["apple", "banana", "cherry"]
let pick = choice(fruits)
print(pick)   # -> randomly "apple", "banana", or "cherry"
```

Example Program

```
import random

# Generate a random float
let r = random()
print(r)

# Roll a dice
let dice = randint(1, 6)
print(dice)

# Random choice from a list
let colors = ["red", "green", "blue", "yellow"]
print(choice(colors))
```

Why `random` Matters

The `random` library makes it simple to **add unpredictability** to programs. Whether simulating dice rolls, shuffling game states, or generating random test data, it provides an easy-to-use API while still leveraging a native DLL backend for performance and reliability.

---

15. SGL (Suny Game Library)

The **Suny Game Library (SGL)** is a high-level framework built on top of Suny, designed for creating **2D games and interactive graphics applications**. It provides developers with a collection of APIs for **window management, rendering, input handling, and sound**, making it easier to build games without needing to interact directly with low-level system APIs.

SGL is modeled after popular game libraries such as **Pygame, SDL, and LOVE2D**, but is fully integrated into the Suny ecosystem. This means developers can leverage Suny syntax, Suny's object system, and standard libraries alongside SGL.

15.1 Features of SGL

- **Window Management**: Create and manage application windows.
- **Rendering API**: Draw primitives (rectangles, circles, lines), images, and text.
- **Input Handling**: Keyboard, mouse, and controller support.
- **Audio Support**: Play background music and sound effects.
- **Timing Utilities**: Control game loops, frame rates, and animations.
- **Event System**: Handle quit events, key presses, and mouse actions.

15.2 Basic Example: Moving a Cube

The following example demonstrates how to create a **game window** and control a simple **cube** using the W, A, S, D keys.

```
include "random"
include "SGL"

# Window settings
width = 700
height = 700
init("Limbo", width, height)

# Colors
screen_color = [255, 255, 255]   # white background
cube_color = [0, 0, 0]           # black cube

# Cube properties
cube_x = 0
cube_y = 0
cube_size = 20
cube_vel = 20

running = true

# Main game loop
while running do
    # Clear screen
    screen_fill(screen_color[0], screen_color[1], screen_color[2])

    # Draw cube
    draw_square(cube_x, cube_y, cube_size,
                cube_color[0], cube_color[1], cube_color[2])

    # Handle input
    key = get_key()
    if key == "a" then cube_x = cube_x - cube_vel end
    if key == "d" then cube_x = cube_x + cube_vel end
    if key == "w" then cube_y = cube_y - cube_vel end
    if key == "s" then cube_y = cube_y + cube_vel end

    # Keep cube inside window
    if cube_x < 0 then cube_x = 0 end
    if cube_x > width - cube_size then cube_x = width - cube_size end
    if cube_y < 0 then cube_y = 0 end
    if cube_y > height - cube_size then cube_y = height - cube_size end

    # Update screen
    flip()
end

close()
```

Explanation:

- `init(title, width, height)` → Opens a new game window.
- `screen_fill(r, g, b)` → Fills the window with a background color.
- `draw_square(x, y, size, r, g, b)` → Draws a square on the screen.
- `get_key()` → Returns the last key pressed.
- `flip()` → Refreshes the screen after drawing.
- `close()` → Closes the game window.

15.3 Core Modules

15.3.1 Window & Events

- `init(title, width, height)` → Open a new game window.
- `is_open()` → Returns `true` if the window is open.
- `close()` → Closes the window.

15.3.2 Rendering

- `screen_fill(r, g, b)` → Clear the screen with a color.
- `draw_square(x, y, size, r, g, b)` → Draw a square.
- `draw_circle(x, y, radius, r, g, b)` → Draw a circle.
- `draw_line(x1, y1, x2, y2, r, g, b)` → Draw a line.
- `draw_text(x, y, text, size, r, g, b)` → Draw text.
- `draw_image(x, y, file_path)` → Draw an image from a file.

---

16. End

by dinhsonhai132