Programming in Suny (version 1.1)

1 Suny Runtime

The Suny Runtime, known as CSuny, is the core of the Suny system.
It executes Suny programs after the source code is translated into bytecode.
CSuny iterates through the bytecode instruction stream and executes each opcode sequentially.
The runtime focuses exclusively on opcode execution and does not handle source-level concerns.
CSuny is written in C to achieve high performance and precise control over memory, execution flow, and system resources.
This design enables Suny to communicate efficiently with low-level system layers.
A CSuny runtime instance executes one program at a time, sequentially.
Runtime instances in Suny do not share state.
Each runtime instance maintains its own execution context, including global variables and runtime-managed objects.

1.1 Responsibilities

The CSuny runtime is responsible for managing and controlling program execution at runtime.
Its primary responsibilities include:

- Executing bytecode instructions sequentially
- Managing the runtime execution state
- Resolving and accessing variables during execution
- Creating and managing runtime values and objects
- Handling function calls and returns
- Invoking native functions exposed by the host system
- Detecting and reporting runtime errors
  The runtime operates strictly at the execution level and does not perform source-level analysis or validation.

1.2 Runtime Instance

A CSuny runtime instance represents an isolated execution environment for a Suny program.

Each runtime instance maintains its own execution context, including global variables, runtime-managed objects, and internal state required for program execution. Runtime instances do not share state with one another.

A runtime instance executes one program at a time, sequentially. While multiple runtime instances may exist within the same host process, each instance operates independently and does not execute programs concurrently.

Runtime instances are created explicitly by the host environment and remain active for the duration of program execution. Once execution completes, the runtime instance can be safely destroyed and all associated runtime state is released.

1.3 High-level Execution Flow

At a high level, the execution of a Suny program within CSuny follows a linear and deterministic flow.

```
Suny Source Code
        ↓
Bytecode Translation
        ↓
Runtime Initialization
        ↓
Opcode Execution
        ↓
Program Termination / Runtime Error
```

After the source code is translated into bytecode, a CSuny runtime instance is initialized.
The runtime then iterates through the bytecode instruction stream and executes each opcode sequentially.

Execution continues until the program reaches its end or a runtime error occurs, at which point the runtime terminates execution and releases all associated runtime state.

---

2 Runtime Lifecycle

2.1 Runtime Creation

A CSuny runtime is created explicitly by the host environment before program execution begins.

During runtime creation, a new runtime instance is allocated and prepared to serve as an isolated execution environment. At this stage, no program code is executed, and no bytecode is evaluated.

Each runtime creation results in a fresh and independent runtime instance, ensuring that execution state, global variables, and runtime-managed objects are not shared with any other runtime instance.

Runtime creation is a prerequisite for program execution and must occur before any bytecode can be executed.

2.2 Initialization

During initialization, the CSuny runtime prepares all internal structures required for program execution.

This phase establishes the execution environment but does not yet begin opcode execution.

Initialization typically includes:

- Setting up the runtime execution stack used for function calls and control flow
- Initializing the global execution context
- Preparing structures for local execution scopes
- Allocating and initializing the runtime-managed heap
- Loading the program bytecode into the runtime
- Initializing garbage collection or memory tracking mechanisms
- Preparing internal lookup structures such as label or instruction maps

Once initialization is complete, the runtime is ready to begin executing bytecode instructions.

2.3 Program Execution

After initialization is complete, the CSuny runtime begins program execution.

During this phase, the runtime processes the bytecode instruction stream one opcode at a time, executing instructions sequentially in a deterministic order defined by the program's control flow.

Execution is driven by a central execution loop. In each iteration, the runtime fetches the current opcode, dispatches it to the appropriate handler, and updates the internal execution state, including the instruction pointer, runtime stack, and active execution contexts. Function calls, control flow transitions, and value manipulations are handled entirely within the runtime.

Opcode dispatch follows a structured control flow, typically implemented using a loop-based dispatch mechanism such as a switch-based opcode handler. Execution continues until the runtime reaches the end of the instruction stream or encounters a runtime error that causes execution to terminate.

2.4 Runtime Termination

Program execution in CSuny terminates when the runtime reaches the end of the bytecode instruction stream or encounters a condition that halts execution.
Termination can occur through several mechanisms:

- Normal termination: The runtime completes all bytecode instructions successfully and reaches the end of the program
- Explicit termination: An explicit exit or halt opcode instructs the runtime to stop execution
- Runtime error: A runtime error is detected during opcode execution, causing immediate termination
- Host interruption: The host environment forcibly terminates the runtime instance

Upon termination, the runtime enters a cleanup phase where all execution state is finalized. This includes:

- Finalizing the current execution context
- Releasing all runtime-managed objects and heap-allocated memory
- Clearing the execution stack
- Releasing bytecode instruction streams and associated metadata
- Invoking garbage collection or memory cleanup mechanisms to reclaim all runtime resources

Once termination and cleanup are complete, the runtime instance can be safely destroyed. The runtime guarantees that all resources associated with the terminated program are released, ensuring no memory leaks or dangling references remain.
After termination, the same runtime instance may be reinitialized to execute a new program, or it may be permanently destroyed by the host environment.

---

3 Garbage Collector

The CSuny runtime uses a **reference-counted memory management system** to manage the lifetime of runtime-managed objects.

Object memory is reclaimed **deterministically** based on reference counts rather than through tracing or periodic garbage collection cycles. Each object maintains an internal reference count that reflects the number of active owning references held by the runtime.

Memory reclamation occurs automatically when an object's reference count reaches zero. At that point, the runtime immediately releases the object's memory and associated resources. No reachability analysis, marking phase, or sweeping phase is performed.

The memory management system operates as an integral part of the runtime execution model and is tightly coupled with runtime operations such as stack manipulation, variable assignment, and object storage. This approach provides predictable object lifetimes, eliminates garbage collection pauses, and ensures precise control over memory usage within a single-threaded execution environment.

3.1 Collection Algorithm

Unlike many modern programming languages that employ tracing-based garbage collection algorithms such as mark-and-sweep or generational collection, the Suny runtime uses a **deterministic reference counting algorithm**.

The runtime does not perform reachability analysis, background tracing, or periodic collection cycles. Instead, each runtime-managed object maintains an internal reference count that is automatically updated by the runtime during operations such as stack manipulation, variable assignment, and object storage.

Object memory is reclaimed immediately when an object’s reference count reaches zero. This process is fully handled by the runtime and does not require explicit deallocation or garbage collection triggers from the program.

This design prioritizes predictability and precise control over object lifetimes. By eliminating tracing phases and collection pauses, the runtime guarantees immediate memory reclamation and consistent performance characteristics within a single-threaded execution model.

Reference Counting Model

Every object allocated in the Suny runtime maintains an internal reference count. This reference count is typically stored in the object's garbage collector metadata structure and accessed via `obj->gc->ref_count`. The reference count represents the number of active references to the object at any given point in time.

When an object is first created, its reference count is initialized to an appropriate value, typically 1, indicating that the creator holds a reference to the object. As references to the object are created, stored, or passed to other parts of the program, the reference count must be incremented. Conversely, when a reference is no longer needed, goes out of scope, or is overwritten, the reference count must be decremented.

When the reference count of an object reaches zero, the object is considered unreachable and eligible for collection. At this point, the object's memory can be safely reclaimed by the garbage collector, and all resources associated with the object are released.

Memory Management Functions

Suny provides three core functions for explicit memory management and reference counting. These functions form the foundation of the manual memory management model and must be used correctly to prevent memory leaks, dangling pointers, or premature deallocation.

`_SUNYINREF`

The `_SUNYINREF` function increments the reference count of an object by one. This function must be called whenever a new reference to the object is created, stored, or retained. Common scenarios where `_SUNYINREF` should be used include:

- Storing an object reference in a data structure such as an array, list, or map
- Assigning an object reference to a variable that will outlive the current scope
- Passing an object reference to a function that retains ownership beyond the call
- Capturing an object reference in a closure or callback

Failing to call `_SUNYINREF` when creating a new reference can result in premature deallocation if the original reference is released while the new reference is still in use.

`_SUNYDEREF`

The `_SUNYDEREF` function decrements the reference count of an object by one. This function must be called whenever a reference to the object is no longer needed, goes out of scope, or is being replaced. Common scenarios where `_SUNYDEREF` should be used include:

- A local variable holding an object reference goes out of scope
- An object reference stored in a data structure is being removed or replaced
- A function that borrowed an object reference is returning and no longer needs the reference
- An object reference is being overwritten with a new value

Failing to call `_SUNYDEREF` when a reference is no longer needed can result in memory leaks, as the object's reference count will never reach zero and its memory will never be reclaimed.

`MOVETOGC`

The `MOVETOGC` function transfers ownership of an object to the garbage collector and attempts to reclaim the object's memory if its reference count has reached zero. This function is typically called after decrementing an object's reference count to finalize its lifecycle and trigger cleanup if appropriate.

If the object's reference count is greater than zero when `MOVETOGC` is called, the object remains alive and no memory is reclaimed. However, if the reference count has reached zero, the garbage collector reclaims the object's memory, invokes any necessary destructors or finalizers, and releases all associated resources.

`MOVETOGC` serves as the bridge between manual reference counting and the garbage collector's memory reclamation process. It is the primary mechanism by which unreachable objects are identified and deallocated.

Automatic Reference Management in API Functions

While explicit reference counting is required when directly manipulating objects, **many** API functions provided by the Suny runtime manage reference counts automatically on behalf of the caller. This design reduces the burden on developers when using standard library functions and common runtime APIs.

For example, when passing an object to a Suny API function that explicitly stores or retains the object, the function will typically call `_SUNYINREF` internally to increment the reference count. Similarly, when a function releases an object or replaces a stored reference, it will call `_SUNYDEREF` to decrement the reference count.

However, **not all API functions perform automatic reference management**. Certain low-level or performance-critical APIs may transfer ownership or require the caller to explicitly manage reference counts. Such APIs are expected to document their ownership and lifetime semantics clearly.

As a result, developers must understand the ownership contract of each API they use. While many APIs handle reference management transparently, **explicit reference counting remains necessary** when creating objects, storing them in custom data structures, or interacting with APIs that do not manage reference counts automatically.

Responsibilities and Tradeoffs

The manual reference counting model places the responsibility for memory correctness on the programmer. Developers must carefully track object lifetimes, ensuring that reference counts are incremented and decremented correctly at all times. Incorrect reference counting can lead to several classes of errors:

- **Memory leaks**: Failing to decrement reference counts results in objects that are never deallocated, consuming memory indefinitely.
- **Use-after-free**: Decrementing reference counts prematurely can result in objects being deallocated while references to them are still in use, leading to undefined behavior.
- **Double-free**: Decrementing reference counts multiple times for the same reference can cause an object to be freed multiple times, corrupting memory.

Despite these challenges, the manual reference counting model offers significant advantages:

- **Deterministic deallocation**: Objects are reclaimed immediately when their reference count reaches zero, providing predictable memory usage patterns.

- **No garbage collection pauses**: The runtime does not perform periodic tracing or sweeping, eliminating unpredictable latency spikes.
- **Fine-grained control**: Developers have precise control over when objects are retained and released, enabling optimization of memory usage for performance-critical applications.

This tradeoff between control and safety is a deliberate design decision in Suny, reflecting the runtime's focus on predictability, performance, and low-level system integration.

4 Value and Type

4.1 Overview

The Suny runtime defines a set of fundamental data types that programs can use to represent and manipulate values. These types form the foundation of Suny's type system and provide the building blocks for all data structures and computations within the runtime.

Suny supports the following primitive and composite data types:

- **Boolean**: Represents logical true and false values
- **Number**: Represents numeric values, including both integers and floating-point numbers
- **String**: Represents sequences of characters and text data
- **List**: Represents ordered collections of values
- **Function**: Represents callable functions and closures
- **Class**: Represents user-defined types and class definitions
- **Userdata**: Represents opaque data managed by the host environment or native code

Each type has its own internal representation, memory layout, and set of operations. The runtime enforces type safety and provides mechanisms for type checking, type conversion, and type validation during program execution.

4.2 Type System Architecture

The Suny type system is designed around explicit type representation and runtime type checking. Every value in the runtime carries type information that identifies its type and enables the runtime to perform type-specific operations safely.

Values in Suny are typically represented as tagged unions or structures that contain:

- A type tag identifying the value's type
- The actual data payload appropriate for that type
- Metadata for garbage collection (reference count, GC flags)

This representation enables efficient type dispatch and ensures that operations are performed only on compatible types.

4.3 Boolean Type

4.3.1 Description

The Boolean type represents logical truth values. A Boolean value can be either `true` or `false`, corresponding to the logical states of truth and falsehood.

4.3.2 Representation

Booleans are typically represented as integer values, with `0` representing `false` and `1` representing `true`. The runtime may optimize Boolean storage by using a single bit or byte depending on platform and alignment requirements.

4.3.3 Operations

Common operations on Boolean values include:

- Logical AND, OR, NOT
- Equality and inequality comparison
- Type conversion to and from other types

4.3.4 Type Conversion

When converted to other types:

- To Number: `false` becomes `0`, `true` becomes `1`
- To String: `false` becomes `"false"`, `true` becomes `"true"`

4.4 Number Type

4.4.1 Description

The Number type represents numeric values. Suny supports two categories of numbers:

- **Integer**: Whole numbers without fractional components
- **Float**: Floating-point numbers with fractional components

4.4.2 Representation

Numbers are represented using standard C numeric types:

- Integers are typically represented as `int64_t` or `int32_t` depending on platform
- Floats are typically represented as `double` (64-bit IEEE 754 floating-point)

The runtime may distinguish between integer and float representations internally for performance optimization, but both are considered part of the unified Number type from the program's perspective.

4.4.3 Operations

Common operations on Number values include:

- Arithmetic operations: addition, subtraction, multiplication, division, modulo
- Comparison operations: less than, greater than, equality
- Bitwise operations (for integers): AND, OR, XOR, shift
- Mathematical functions: absolute value, power, square root

4.4.4 Integer and Float Distinction

While both integers and floats are considered Numbers, the runtime maintains internal distinctions:

- Integer operations produce integer results when possible
- Operations involving at least one float produce float results
- Division of two integers may produce a float if the result is not exact
- Explicit conversion functions allow programs to control integer/float conversion

4.4.5 Type Conversion

When converted to other types:

- To Boolean: `0` becomes `false`, all other values become `true`
- To String: Numbers are converted to their decimal string representation

4.5 String Type

4.5.1 Description

The String type represents sequences of characters and text data. Strings are immutable in Suny—once created, their contents cannot be modified. Operations that appear to modify strings actually create new string objects.

4.5.2 Representation

Strings are represented as sequences of bytes, typically encoded in UTF-8. The runtime stores strings with the following structure:

- Length field indicating the number of bytes
- Pointer to character data buffer
- Null terminator for C compatibility (if applicable)
- Hash value for efficient string comparison (optional)

4.5.3 String Interning

The runtime may employ string interning to optimize memory usage and comparison performance. Interned strings with identical contents share the same underlying memory, enabling constant-time equality comparison through pointer comparison.

4.5.4 Operations

Common operations on String values include:

- Concatenation: combining two or more strings
- Substring extraction: obtaining a portion of a string
- Length query: determining the number of characters
- Character access: retrieving individual characters by index
- Comparison: lexicographic ordering and equality testing
- Search: finding substrings or characters within a string

4.5.5 Type Conversion

When converted to other types:

- To Boolean: Empty string `""` becomes `false`, all other strings become `true`
- To Number: The runtime attempts to parse the string as a numeric literal; invalid formats may produce `0` or an error

4.6 List Type

4.6.1 Description

The List type represents ordered, mutable collections of values. Lists can contain values of any type, including mixed types within the same list. Lists support dynamic resizing and provide efficient indexed access.

4.6.2 Representation

Lists are typically represented as dynamic arrays with the following structure:

- Capacity field indicating allocated storage size
- Count field indicating current number of elements
- Pointer to data buffer containing element values

When a list's capacity is exceeded, the runtime reallocates a larger buffer and copies existing elements to maintain contiguous storage.

4.6.3 Operations

Common operations on List values include:

- Indexed access: retrieving or modifying elements by position
- Append: adding elements to the end of the list
- Insert: adding elements at specific positions
- Remove: deleting elements from the list
- Length query: determining the number of elements
- Iteration: traversing all elements in order
- Slicing: extracting sublists

4.6.4 Memory Management

List elements may be objects that require reference counting. When elements are added to a list, their reference counts are incremented. When elements are removed or the list is destroyed, reference counts are decremented appropriately.

4.6.5 Type Conversion

When converted to other types:

- To Boolean: Empty list `[]` becomes `false`, all other lists become `true`
- To String: The runtime may produce a string representation such as `"[1, 2, 3]"`

4.7 Function Type

4.7.1 Description

The Function type represents callable functions, including both user-defined functions written in Suny and native functions provided by the host environment. Functions are first-class values in Suny and can be passed as arguments, returned from other functions, and stored in data structures.

4.7.2 Representation

Functions are represented as objects containing:

- Pointer to bytecode or native code
- Parameter count and signature information
- Closure environment (captured variables from enclosing scopes)
- Function metadata (name, source location)

4.7.3 Closures

When a function references variables from enclosing scopes, it captures those variables in a closure. The closure maintains references to the captured variables, ensuring they remain accessible even after the enclosing scope has exited.

Captured variables are reference-counted to ensure correct memory management when closures outlive their defining scopes.

4.7.4 Operations

Common operations on Function values include:

- Invocation: calling the function with arguments
- Partial application: creating new functions with some arguments pre-filled
- Comparison: testing function equality (typically pointer equality)

4.7.5 Native Functions

Native functions are implemented in C or other host languages and exposed to Suny programs through the API. Native functions follow calling conventions that enable them to receive Suny values as arguments and return Suny values as results.

The runtime manages the transition between Suny bytecode execution and native code execution, handling parameter marshalling and result conversion automatically.

4.8 Class Type

4.8.1 Description

The Class type represents user-defined types and class definitions. Classes serve as templates for creating objects with specific fields and methods. Classes enable object-oriented programming patterns such as encapsulation, inheritance, and polymorphism.

4.8.2 Representation

Classes are represented as objects containing:

- Class name and metadata
- Field definitions (names and types)
- Method definitions (functions associated with the class)
- Constructor and destructor functions
- Parent class reference (for inheritance)

4.8.3 Object Instantiation

When a class is instantiated, the runtime allocates memory for an object instance, initializes its fields according to the class definition, and invokes the constructor function if defined.

Object instances maintain a reference to their class, enabling the runtime to perform type checking and method dispatch.

4.8.4 Methods and Field Access

Methods are functions associated with a class that receive the object instance as an implicit first parameter (often named `this` or `self`). The runtime handles method lookup and dispatch based on the object's class.

Field access is performed through offset-based addressing within the object's memory layout, ensuring efficient property access without name lookup overhead at runtime.

4.8.5 Inheritance

Classes may inherit from parent classes, acquiring their fields and methods. The runtime supports single inheritance, where each class has at most one direct parent class.

Method overriding enables derived classes to provide specialized implementations of inherited methods, with the runtime performing dynamic dispatch to invoke the correct method based on the object's actual type.

4.9 Userdata Type

4.9.1 Description

The Userdata type represents opaque data managed by the host environment or native code. Userdata enables Suny programs to work with external resources, native data structures, and platform-specific objects without exposing their internal representation to the runtime.

4.9.2 Representation

Userdata objects contain:

- Pointer to native data
- Type tag or metadata identifying the userdata type
- Optional destructor function for cleanup
- Reference count for memory management

4.9.3 Operations

Operations on Userdata are typically provided by native functions that understand the userdata's internal structure. The Suny runtime does not interpret or manipulate userdata contents directly.

Common patterns include:

- Native functions that accept userdata and perform operations on the underlying native object
- Conversion functions that create userdata from Suny values or extract Suny values from userdata
- Destructor functions that release native resources when userdata is garbage collected

4.9.4 Memory Management

Userdata objects participate in reference counting like other Suny objects. When a userdata object's reference count reaches zero, the runtime invokes its destructor function (if provided) before reclaiming memory.

This ensures that native resources such as file handles, network connections, or allocated memory are properly released when no longer needed.

4.9.5 Type Safety

The runtime provides mechanisms for type-checking userdata to ensure that native functions receive userdata of the expected type. Type tags or type metadata enable safe casting and validation before native operations are performed.

4.10 Type Checking and Validation

The runtime performs type checking during operation execution to ensure type safety. When an operation is performed on a value, the runtime verifies that the value's type is compatible with the operation.

Type checking occurs at runtime rather than during bytecode translation, reflecting Suny's dynamic typing model. Invalid type operations result in runtime errors that terminate execution.

The API provides functions for querying a value's type, performing type checks, and converting between types safely.

4.11 Type Conversion

The runtime supports both implicit and explicit type conversion:

Implicit conversion occurs automatically in contexts where a specific type is expected. For example, numbers may be implicitly converted to strings in string concatenation operations.

Explicit conversion is performed through dedicated conversion functions or opcodes that transform values from one type to another. Explicit conversions may fail if the source value cannot be meaningfully represented in the target type.

Conversion behavior is well-defined for all type pairs, ensuring predictable program behavior across different execution contexts.

---

5 The Language

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 mind 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: " + string(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")         # you can sub string using '+'
print(10 + 20)                      # prints the result of an expression
print(null)                         # prints "null"
```

Output:

```
123
Suny Language
30
null
```

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 `null` 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
>> print("Suny is fun!")
Suny is fun!
```

```
>> 5 * (2 + 3)
25
>> exit(1)
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(10 % 2)       # modulo: 1
print((1 + 2) * 3)  # Parentheses control order: 9
```

Explanation:

- `+` adds numbers.
- `-` subtracts numbers.
- `*` multiplies numbers.
- `%` returns the remainder after division.
- `/` 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.

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. Multiplication, Division, Modulo `* / %`
3. Addition and Subtraction `+ -`
4. 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**: `+ - * / %`.
- 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
```

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 = score + points + local_bonus
    print("Added: " + string(points) + ", Current score: " + string(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" then
    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
```

5.3 Strings

Strings represent **text**.
They are sequences of characters enclosed in **double 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
`\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 Closure

A **closure** is a function that can *remember and access* variables from the scope in which it was created, even after that outer scope has finished executing.

In other words:

> **Closure = Function + Its captured environment (the variables it keeps).**

This allows functions to "carry" state and behave like objects with private data.

Detailed Example

```
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
```

Here's what happens:

1. `foo()` is called. Inside it, the variable `count` is created.
2. `foo()` returns an inner function that **uses `count`**.
3. Even though `foo()` has finished, `count` does **not** disappear.
4. The returned function `a()` keeps a reference to `count`.

   This saved environment is the **closure**.
5. Each time `a()` is called, it updates the same `count` value.

So the function "remembers" its state:

```
1
2
3
...
10
```

5.6 HashMap

HashMaps (also called dictionaries or objects) are **key-value collections** that allow you to store and retrieve data using keys instead of numeric indices.

```
person = {
    "name": "Alice",
    "age": 25,
    "city": "Hanoi"
}

print(person["name"])   # Alice
print(person["age"])    # 25
```

Creating HashMaps

HashMaps are created using curly braces `{}` with key-value pairs:

```
# Empty HashMap
empty = {}

# HashMap with initial values
user = {
    "username": "hai123",
    "email": "hai@example.com",
    "active": true
}
```

Accessing Values

Use square brackets `[]` with the key to access values:

```
config = {
    "theme": "dark",
    "language": "vi"
}

print(config["theme"])      # dark
print(config["language"])   # vi
```

Modifying and Adding Values

You can change existing values or add new key-value pairs:

```
settings = {
    "volume": 50,
    "muted": false
}

# Modify existing value
settings["volume"] = 75

# Add new key-value pair
settings["brightness"] = 80

print(settings)
# { "volume": 75, "muted": false, "brightness": 80 }
```

Nested HashMaps

HashMaps can contain other HashMaps, creating nested structures:

```
game = {
    "player": {
        "hp": 100,
        "items": ["sword", "shield"]
    },
    "fps": 60
}

# Access nested values
print(game["player"]["hp"])        # 100
print(game["player"]["items"][0])  # sword

# Modify nested values
game["player"]["hp"] = 200
game["fps"] = 60
game["name"] = "RPG"

print(game)
# {
#     "player": { "hp": 200, "items": ["sword", "shield"] },
#     "fps": 60,
#     "name": "RPG"
# }
```

Common HashMap Operations

```
data = {
    "x": 10,
    "y": 20
}

# Get number of key-value pairs
print(size(data))  # 2

# Check if key exists (using conditional)
if data["z"] == null then
    print("Key 'z' does not exist")
end
```

Use Cases

HashMaps are perfect for:

- Configuration settings
- Game state management

- User profiles and data
- Structured data representation
- Mapping relationships between values

5.7 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

---

6. Class and OOP

Classes allow you to define **custom data types** with **properties** and **methods**.
They are the foundation of **Object-Oriented Programming (OOP)**.

6.1 Defining a Class

A class is defined using the `class` keyword.

```
class Vector do
    x = null
    y = null

    function __init__(self, x, y) do
        self.x = x
        self.y = y
        return self
    end
end
```

6.2 Operator Overloading

Suny allows operator overloading using special methods:

Method	Operator

Method	Operator
`__add__`	`+`
`__sub__`	`-`
`__mul__`	`*`
`__div__`	`/`
`__tostring__`	string conversion

6.3 Full Vector Example

```
class Vector do
    x = null
    y = null

    function __init__(self, x, y) do
        self.x = x
        self.y = y
        return self
    end

    function __add__(a, b) do
        return Vector(a.x + b.x, a.y + b.y)
    end

    function __sub__(a, b) do
        return Vector(a.x - b.x, a.y - b.y)
    end

    function __mul__(a, b) do
        return Vector(a.x * b.x, a.y * b.y)
    end

    function __div__(a, b) do
        return Vector(a.x / b.x, a.y / b.y)
    end

    function __tostring__(self) do
        return "Vector(" + string(self.x) + ", " + string(self.y) + ")"
    end
end
```

6.4 Using the Class

```
a = Vector(10, 20)
b = Vector(2, 4)

print(a + b)  # Vector(12, 24)
print(a - b)  # Vector(8, 16)
print(a * b)  # Vector(20, 80)
print(a / b)  # Vector(5, 5)
```

6.5 Example Loop

```
a = Vector(0, 0)

while true do
    a = a + Vector(1, 1)
    print(a)
end
```

6.6 Notes

- `self` refers to the current object.
- Operator methods always receive **two operands**.
- `__tostring__` controls how an object is printed.
- Class names usually start with a capital letter.

6.7 Inheritance

Suny supports **single** and **multiple inheritance** using the `extends` keyword.

- A subclass automatically inherits **fields** and **methods** from its parent classes.
- If a field or method is redefined, the **child overrides** the parent version.
- With multiple inheritance, resolution follows **left-to-right order**.

6.7.1 Base Classes

```
class Animal do
    leg = 0
end

class Fly do
    wings = 0
end
```

6.7.2 Single Inheritance

```
class Dino extends Animal do
    leg = 2
end
```

`Dino` inherits from `Animal` and overrides `leg`.

```
d = Dino()
print(d.leg)  # 2
```

6.7.3 Multiple Inheritance

```
class Chicken extends Animal, Fly do
    leg = 2
    wings = 2
end
```

`Chicken` inherits from both `Animal` and `Fly`.

```
c = Chicken()
print(c.leg)    # 2
print(c.wings) # 2
```

6.8 Method Inheritance Example

```
class Animal do
    function speak(self) do
```

```
        return "..."
    end
end

class Dog extends Animal do
    function speak(self) do
        return "Woof"
    end
end

print(Dog().speak())  # Woof
```

6.9 Summary

- `extends` enables inheritance
- Child classes can override fields and methods
- Multiple inheritance is supported
- Method lookup is **child → parent (left to right)**

Inheritance allows Suny programs to model real-world relationships cleanly and reuse code effectively.

---

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` , `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 then
    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 then
    print("You passed!")
else
    print("Try again")
end
```

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

7.1.3 `if ... else`

```
score = 85

if score >= 90 then
    print("Excellent")
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 then
    if has_id then
        print("Access granted")
    else
        print("ID required")
    end
else
    print("Too young")
end
```

7.1.5 `if ... elif ... else`

`elif` (else-if) is used when multiple conditions must be checked in sequence, but **only one branch should be executed**.

```
score = 75

if score >= 90 then
    print("Excellent")
elif score >= 75 then
    print("Good")
elif score >= 50 then
    print("Pass")
else
    print("Failed")
end
```

✅ Conditions are evaluated **from top to bottom**
✅ **Only the first matching block** is executed
✅ `else` is **optional**

7.1.6 Multiple `elif` branches

Any number of `elif` branches may be used:

```
x = 0
```

```
if x > 0 then
    print("Positive")
elif x == 0 then
    print("Zero")
elif x < 0 then
    print("Negative")
end
```

7.1.7 `elif` vs Nested `if`

Nested `if` (less readable):

```
if score >= 90 then
    print("Excellent")
else
    if score >= 75 then
        print("Good")
    else
        print("Failed")
    end
end
```

Using `elif` (recommended):

```
if score >= 90 then
    print("Excellent")
elif score >= 75 then
    print("Good")
else
    print("Failed")
end
```

💡 Recommendation

- Use `elif` when conditions are **mutually exclusive**
- Use nested `if` when logic is **hierarchical or dependent**

7.1.5 Comparison Operators Recap

Operator	Meaning	Example	Result

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 Loop Structures: `loop`, `times`, and `do`

In this section, we explore how to execute blocks of code repeatedly using basic looping constructs. These tools are essential for handling tasks that require iteration, such as incrementing counters or processing data sets.

---

1. The Basic `loop`

The `loop` keyword creates an **infinite loop**. It will continue to execute the code inside the `do...end` block until it is manually interrupted (e.g., by a `break` statement or a system exit).

Syntax Example:

```
i = 0

loop do
    print(i)
    i = i + 1

    if i >= 5 then
        break
    end
end
```

---

2. The `times` Iterator

When you know exactly how many times a block of code should run, use the `times` method. This is a cleaner, more readable way to handle fixed iterations compared to manual counters.

Syntax Example:

```
i = 0

loop 5 times do
    print(i)
    i = i + 1
end
```

- **How it works:** The code inside the `do...end` block will execute exactly 5 times.
- **Alternative Syntax:** You can also use curly braces for single-line loops: `5.times { |n| print(n) }`.

---

3. Comparison Table

Feature	`loop`	`times`
Duration	Infinite (until broken)	Fixed number of iterations
Use Case	Waiting for an external signal or user input	Repeating a task a specific number of times
Control	Requires an internal `break`	Ends automatically

Key Takeaways

- **The `do...end` Block:** Both constructs use `do` and `end` to encapsulate the logic that should be repeated.
- **Counter Management:** In a `loop`, you are responsible for incrementing your variables and defining the exit condition. In `times`, the iteration limit is handled for you.

> **Warning:** Always ensure that an infinite `loop` has a way to terminate, otherwise your program will hang or crash due to memory exhaustion.

7.2.6 Nested Loops

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

```
3
4
```

7.4 Combining Control Structures

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

```
for i in range(1, 11) then
    if i % 2 == 0 then
        print(string(i) + " is even")
    else
        print(string(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.suny"

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.suny"
```

8.3 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.

---

6 The Application Program Interface

The Suny Application Program Interface (API) provides a comprehensive set of functions and utilities for interacting with the CSuny runtime from host environments and external programs. The API serves as the primary interface between host applications and the Suny execution environment, enabling programs to create runtime instances, execute bytecode, manipulate runtime objects, and integrate Suny functionality into larger systems.

The Suny API is designed with several key principles in mind:

- **Host integration**: The API enables seamless embedding of the Suny runtime into host applications written in C or other languages with C interoperability.
- **Resource control**: The API provides explicit control over runtime lifecycle, memory management, and execution state.
- **Type safety**: The API enforces type distinctions and provides functions for safe type conversion and validation.
- **Error handling**: The API includes comprehensive error reporting mechanisms to detect and communicate runtime errors to the host environment.
- **Performance**: The API is designed to minimize overhead and provide efficient access to runtime functionality.

The API is divided into several logical categories, each addressing a specific aspect of runtime interaction:

- Runtime management: Creating, initializing, executing, and destroying runtime instances
- Object creation and manipulation: Allocating and working with runtime-managed objects
- Type system integration: Creating and working with Suny types and values
- Function invocation: Calling Suny functions from host code and exposing host functions to Suny programs
- Memory management: Explicit reference counting and garbage collector interaction
- Error handling: Detecting, reporting, and recovering from runtime errors

All API functions follow consistent naming conventions, parameter ordering, and return value semantics to ensure predictability and ease of use. Functions that allocate or return objects typically return pointers that must be managed according to the reference counting rules described in section 3.2.

6.1 The Stack

The CSuny runtime maintains an execution stack to manage function calls, local variables, and control flow during program execution. The stack is a core component of the runtime's execution model and serves as the primary mechanism for managing execution context as the program transitions between functions, scopes, and control flow constructs.

The stack operates on a last-in-first-out (LIFO) principle. When a function is called, a new stack frame is pushed onto the stack, creating an isolated execution context for that function. When the function returns, its stack frame is popped from the stack, and execution resumes in the caller's context.

The stack is allocated and managed entirely by the runtime. Programs do not directly manipulate the stack structure, but the runtime's opcode execution logic interacts with the stack continuously to implement function calls, parameter passing, local variable storage, and return value handling.

6.1.2 Stack Structure

In Suny, the stack is not a standalone data structure. It is embedded within an `Sframe` , which is the fundamental execution frame used by the runtime and implemented in C.

An `Sframe` represents the complete execution state of a single function invocation. The value stack used for argument passing, temporary values, and return values is allocated as part of this frame.

In addition to the stack region, an `Sframe` also maintains:

- storage for local variables,
- references to global values,
- access to constant data associated with the function.

All runtime-visible values during function execution are stored and accessed through the `Sframe` . The lifetime of the stack is therefore bound to the lifetime of its owning `Sframe` .

6.1.3 Stack Ownership and Lifetime

- A new stack is created when a function is invoked.
- The stack exists only while the owning Sframe is active.
- When the function returns, the entire stack is discarded.
- Stack values must not be accessed after the frame is destroyed.

The runtime guarantees that stack cleanup is automatic and does not require manual memory management by user-level code.

6.2 Functions and Types

In Suny, the implementation is organized into several functional groups. Each group has a specific responsibility and provides a cohesive set of operations for a particular subsystem. Key groups include `Sframe` for stack frame management, global and local variable access; `Seval` for expression evaluation and bytecode execution; `Smem` for memory allocation and heap management; `Sobj` for object initialization and construction; and additional groups for type system operations, garbage collection, and error handling.

Below are the primary functional groups in the Suny runtime. Each group encapsulates related functionality and provides a distinct set of operations.

`Sobj`

`Sobj` is the fundamental **value representation** in SUNY and forms the basis of all runtime computations.

Every value manipulated by the SUNY runtime—such as numbers, strings, booleans, functions, and objects—is represented internally as an `Sobj`. All execution structures, including the stack, locals, globals, constants, and heap references, store **pointers to `Sobj` instances**, not raw values.

`Sobj` serves as the **unified value abstraction** of the runtime. Arithmetic operations, comparisons, function calls, and data movement within the VM operate exclusively on `Sobj` references.

As a core runtime type, `Sobj` is tightly integrated with:

- The execution stack.
- Garbage collection.
- Scope management.
- Object lifetime tracking.

Correct handling of `Sobj` is therefore critical to the correctness, performance, and memory safety of the SUNY runtime.

Sobj_make_number

```
SUNY_API struct Sobj* Sobj_make_number(double value);
```

Allocates and initializes a new numeric `Sobj` storing the given double value.
The returned object is managed by the SUNY runtime and participates in garbage collection.
Numeric `Sobj` instances are immutable.

Sobj_make_bool

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

Allocates and initializes a new boolean `Sobj`.
The provided integer value is interpreted according to SUNY boolean semantics, where `0` represents false and any non-zero value represents true.

Sobj_shallow_copy

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

Creates a shallow copy of the given `Sobj` .

This function allocates a new `Sobj` instance whose internal structure is copied from the source object. For reference-type fields, the copy shares the same underlying references as the original object.

Notes:

- Primitive values (such as numbers and booleans) are copied by value.
- Reference fields (such as heap objects, tables, or strings) are not duplicated.
- Both the original object and the copied object reference the same underlying data.
- The returned object is managed by the SUNY runtime and participates in garbage collection.

Sobj_deep_copy

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

Creates a deep copy of the given `Sobj` .

This function allocates and initializes a new `Sobj` by recursively copying the contents of the source object. All owned data and nested objects are duplicated, ensuring that the returned object is completely independent of the original.

Unlike Sobj_shallow_copy, no internal references are shared between the source and the copied object.

Notes:

- Primitive types (numbers, booleans) are copied by value.
- Composite or reference types (tables, strings, objects, closures) are fully duplicated.
- Changes made to the copied object do not affect the original.
- Deep copying is more expensive than shallow copying and should be used only when full isolation is required.
- The returned Sobj is managed by the SUNY runtime and participates in garbage collection.

Sobj_make_string

```
struct Sobj *Sobj_make_string(char* str, int size);
```

Allocates and initializes a new `Sobj` representing a string value.

This function creates a string-type `Sobj` by copying size bytes from the provided character buffer str. The resulting string object is immutable and managed by the SUNY runtime.

Sobj_make_char

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

Allocates and initializes a new `Sobj` representing a character value.

This function creates a character-type `Sobj` that stores a single char value. Character objects in SUNY are treated as immutable scalar values and are managed by the SUNY runtime.

`Seval`

`Seval` is the **evaluation layer** of the SUNY runtime, responsible for performing computations and operations on `Sobj` values.

While `Sobj` defines what a value is, `Seval` defines how values interact.

All runtime operations—such as arithmetic (`+` , `-` , `*` , `/`), comparisons, logical operations, and object-level behaviors—are executed through the `Seval` system. `Seval` consumes one or more `Sobj` references, applies the requested operation according to SUNY semantics, and produces a resulting `Sobj` .

`Seval` operates exclusively on `Sobj` pointers and never on raw C values directly. This guarantees consistent behavior across:

- Different value types (number, string, bool, object, etc.)
- Garbage-collected memory
- Dynamic typing rules

Typical responsibilities of `Seval` include:

- Arithmetic evaluation (`+` , `-` , `*` , `/` , `%`)
- Comparison and equality checks
- Logical operations
- Object and value coercion (if supported)
- Dispatching operations based on `Sobj` type
- Error signaling for invalid operations (e.g. type mismatch)

Because all computation flows through `Seval` , it acts as a **semantic gatekeeper** of the SUNY runtime, enforcing language rules and ensuring that operations remain safe, predictable, and consistent.

Ok, đây là **mục tài liệu hóa chuẩn** cho hàm này, đúng kiểu runtime/VM doc 👇

Seval_add

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

Performs the **addition (`+`) operation** between two `Sobj` values according to SUNY runtime semantics.

`Seval_add` is part of the `Seval` evaluation layer and is responsible for dispatching the addition operation based on the runtime types of its operands.

Behavior:

The function evaluates operands in the following order:

1. **Null check**
   If either operand has type `NULL_OBJ` , a runtime error is raised:

   ```
   attempt to perform arithmetic on a null value
   ```

2. **String addition**
   If both operands are of type `STRING_OBJ` , the operation is delegated to:

   - `eval_string(obj1, obj2, __add)`

   This typically represents string concatenation.

3. **List addition**
   If both operands are of type `LIST_OBJ` , the operation is delegated to:

   - `eval_list(obj1, obj2, __add)`

   This may represent list concatenation or merging, depending on SUNY semantics.

4. **User-defined data**
   If either operand is of type `USER_DATA_OBJ` , the operation is delegated to:

   - `eval_userdata(obj1, obj2, __add)`

   This allows user-defined types to override or customize the `+` operator.

5. **Numeric addition (default)**
   If none of the above cases match, the operands are treated as numeric values and their underlying numeric fields are added:

```
obj1->value->value + obj2->value->value
```

The result is wrapped into a new numeric `Sobj` .

Returns:

- A newly allocated `Sobj` containing the result of the addition.
- The returned object is managed by the SUNY runtime and participates in garbage collection.

Notes:

- `Seval_add` operates **only on `Sobj` references**, never on raw C values.
- Type dispatch is dynamic and resolved at runtime.
- This function acts as a central enforcement point for SUNY's `+` operator semantics.
- Extending the `+` operator for new types typically requires updating the corresponding `eval_*` handler.

Nice, hàm này gọn và rất "VM-core" 👍
Đây là **mục tài liệu hóa tương ứng** cho `Seval_sub` , cùng style với `Seval_add` :

Seval_sub

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

Performs the **subtraction (`-`) operation** between two `Sobj` values according to SUNY runtime semantics.

`Seval_sub` is part of the `Seval` evaluation layer and handles dynamic dispatch for the subtraction operator.

Behavior:

The evaluation proceeds as follows:

1. **Null check**
   If either operand has type `NULL_OBJ` , a runtime error is raised:

   ```
   attempt to perform arithmetic on a null value
   ```

2. **User-defined data dispatch**
   If either operand is of type `USER_DATA_OBJ` , the operation is delegated to:

    - `eval_userdata(obj1, obj2, __sub)`

   This allows user-defined types to customize the behavior of the subtraction operator.

3. **Numeric subtraction (default)**
   If no special case applies, both operands are treated as numeric values and subtraction is performed on their underlying numeric fields:

   ```
   obj1->value->value - obj2->value->value
   ```

   The result is wrapped into a new numeric `Sobj`.

Returns:

- A newly allocated `Sobj` containing the result of the subtraction.
- The returned object is managed by the SUNY runtime and participates in garbage collection.

Notes:

- `Seval_sub` operates exclusively on `Sobj` references.
- Unlike `Seval_add`, subtraction does **not** define special behavior for strings or lists.
- Operator overloading is supported only through `USER_DATA_OBJ`.
- Type validation is enforced at runtime.

Seval_mul

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

Performs the **multiplication (`*`) operation** between two `Sobj` values according to SUNY runtime semantics.

`Seval_mul` belongs to the `Seval` evaluation layer and dynamically dispatches the multiplication operator based on operand types.

Behavior:

The function evaluates operands in the following order:

1. **Null check**
   If either operand has type `NULL_OBJ`, a runtime error is raised:

   ```
   attempt to perform arithmetic on a null value
   ```

2. **String multiplication**
   If either operand is of type `STRING_OBJ`, the operation is delegated to:

   - `eval_string(obj1, obj2, __mul)`

   This may represent string repetition or another string-specific behavior defined by SUNY semantics.

3. **List multiplication**
   If either operand is of type `LIST_OBJ`, the operation is delegated to:

   - `eval_list(obj1, obj2, __mul)`

   This may represent list repetition or expansion.

4. **User-defined data dispatch**
   If either operand is of type `USER_DATA_OBJ`, the operation is delegated to:

   - `eval_userdata(obj1, obj2, __mul)`

   This enables operator overloading for custom data types.

5. **Numeric multiplication (default)**
   If none of the above cases apply, both operands are treated as numeric values and their underlying numeric fields are multiplied:

   ```
   obj1->value->value * obj2->value->value
   ```

   The result is wrapped into a new numeric `Sobj`.

Returns:

- A newly allocated `Sobj` containing the result of the multiplication.
- The returned object is managed by the SUNY runtime and participates in garbage collection.

Notes:

- `Seval_mul` supports polymorphic behavior for strings and lists.
- Operator dispatch is resolved dynamically at runtime.
- Numeric multiplication is the fallback behavior.
- Errors are raised immediately on invalid null operands.

Seval_div

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

Performs the **division (`/`) operation** between two `Sobj` values according to SUNY runtime semantics.

`Seval_div` is part of the `Seval` evaluation layer and is responsible for dispatching the division operator at runtime.

Behavior:

The evaluation proceeds as follows:

1. **Null check**
   If either operand has type `NULL_OBJ`, a runtime error is raised:

   ```
   attempt to perform arithmetic on a null value
   ```

2. **User-defined data dispatch**
   If either operand is of type `USER_DATA_OBJ`, the operation is delegated to:

   - `eval_userdata(obj1, obj2, __div)`

   This allows custom data types to define their own division behavior.

3. **Numeric division (default)**
   If no special case applies, both operands are treated as numeric values and division is performed on their underlying numeric fields:

   ```
   obj1->value->value / obj2->value->value
   ```

   The result is wrapped into a new numeric `Sobj`.

Returns:

- A newly allocated `Sobj` containing the result of the division.
- The returned object is managed by the SUNY runtime and participates in garbage collection.

Notes:

- `Seval_div` does **not** perform an explicit division-by-zero check at this level.
- Division-by-zero behavior depends on the underlying numeric representation (e.g. IEEE-754 semantics).
- String and list types do not define division behavior by default.
- Operator overloading is supported only through `USER_DATA_OBJ`.

`Sframe`

`Sframe` represents an **execution frame** in the SUNY runtime.

An `Sframe` encapsulates the complete runtime state required to execute a block of code, such as a function call, a script, or a nested execution context. Each frame is independent and forms part of a **frame chain** that models call stacks and lexical scope relationships.

At runtime, the SUNY VM always executes within a current `Sframe` .

Core Responsibilities

An `Sframe` is responsible for:

- Tracking instruction execution progress.
- Managing the operand stack.
- Holding references to local, global, heap, and constant values.
- Supporting closures and lexical environments.
- Integrating with garbage collection.
- Linking to parent frames for scope resolution.

Field Overview

Code & Control Flow

- `int f_code_index`

  Current instruction pointer (program counter) within `f_code` .

- `struct Scode *f_code`

  Pointer to the bytecode or executable code being evaluated.

- `struct Slabel_map *f_label_map`

  Mapping of labels to instruction offsets, used for jumps and control flow.

- `int code_line`

  Source-level line number corresponding to the current instruction (for debugging and error reporting).

Operand Stack

- `struct Sobj **f_stack`

  The operand stack used for evaluation.

- `int f_stack_index`

  Index of the current top of the stack.

- `int f_stack_size`

  Total allocated capacity of the stack.

The stack stores **pointers to `Sobj`**, not raw values.

Local, Global, Heap, and Constant Storage

Each of these areas is stored as an array of `Sobj*` with its own index and size:

- `f_locals`, `f_locals_index`, `f_locals_size`

  Local variables for the current execution frame.

- `f_globals`, `f_globals_index`, `f_globals_size`

  Global variables accessible from this frame.

- `f_heaps`, `f_heap_index`, `f_heap_size`

  Heap-allocated objects referenced by this frame.

- `f_consts`, `f_const_index`, `f_const_size`

  Constants (literals, compiled constants) used during execution.

Memory Management

- `struct Garbage_pool *gc_pool`

  Garbage collection pool associated with this frame.

This allows the runtime to track object lifetimes relative to frame execution.

Compilation & Symbol Resolution

- `struct ScompilerUnit *compiler`

  Reference to the compiler unit that produced this frame’s code.

- `struct Stable *table`

  Symbol table used for name resolution and scope lookup.

Closures & Environment

- `struct Sobj *f_obj`

  Object bound to this frame, typically used for closures or method contexts.

- `struct Senvi *envi`

  Lexical environment captured by the frame.

- `struct Sframe *parent`

  Pointer to the parent frame, forming the call stack and scope chain.

Summary

`Sframe` is the **central execution unit** of the SUNY runtime.
It binds together code, data, control flow, memory management, and scope into a single structure.

All SUNY execution—from top-level scripts to deeply nested function calls—ultimately occurs within an `Sframe`.

Sframe_push

```
struct Sobj *Sframe_push(struct Sframe *frame, struct Sobj *obj);
```

Description

`Sframe_push` pushes an object onto the **operand stack** of an execution frame.

The operand stack is used by the SUNY virtual machine to store intermediate values during expression evaluation, instruction execution, and function calls.

Semantics

- The function places the given `Sobj` pointer on top of the frame's operand stack.
- If the stack has no remaining capacity, it is expanded before the push occurs.
- The stack stores **references (`Sobj*`)**, not copied values.
- The pushed object becomes the new top-of-stack element.

Error Handling

- If `frame` is `NULL`, a runtime error is raised.
- No type validation is performed on `obj`.

- Stack overflow is handled internally by growing the stack.

Return Value

- Returns the same `Sobj*` that was pushed onto the stack.
- This allows chaining and inline use during evaluation.

Runtime Notes

- `Sframe_push` does **not** allocate or clone the object.
- Object lifetime is managed by the garbage collector associated with the frame.
- The function performs no evaluation logic and does not modify execution state beyond the operand stack.

Typical Usage

- Pushing literals
- Storing intermediate results of expressions
- Passing arguments to function calls
- Receiving return values

Sframe_pop

```
struct Sobj *Sframe_pop(struct Sframe *frame);
```

Description

`Sframe_pop` removes the top object from the operand stack of an execution frame and **releases the frame's ownership** of that object.

In the SUNY runtime, the operand stack is an explicit owner of objects.
Popping an object from the stack therefore represents a **lifetime transition**, not just a value retrieval.

Semantics

- The operand stack index of the frame is decremented.
- The reference to the top `Sobj` is removed from the stack.
- The frame **decrements the reference count** of the object.
- The object **may be destroyed immediately** if its reference count reaches zero.
- No garbage collector is invoked by this operation.

Ownership Model

- The operand stack is considered an **owning reference**.
- `Sframe_pop` represents the stack **releasing ownership**.
- If the caller intends to keep using the returned object, it must **explicitly acquire ownership** (e.g. by incrementing the reference count).

Failure to do so may result in use-after-free errors.

Error Handling

- If `frame` is `NULL`, a runtime error is raised.
- If the operand stack is empty, a runtime error is raised (stack underflow).

Return Value

- Returns the `Sobj*` that was removed from the stack.
- The returned pointer is **not guaranteed to remain valid** unless the caller establishes ownership.

Notes

- `Sframe_pop` intentionally exposes object lifetime mechanics.
- This function teaches the distinction between:
    - value flow
    - reference ownership
    - object destruction
- Automatic memory safety is deliberately not provided at this level.

Typical Usage

- Consuming operands during expression evaluation
- Implementing stack-based VM instructions
- Demonstrating reference counting and ownership transfer in runtime systems

---

Sframe_load_global

```
struct Sobj *Sframe_load_global(struct Sframe *frame, int address);
```

Description

`Sframe_load_global` retrieves an object from the global storage of an execution frame using a numeric address.

This function is used by the SUNY virtual machine to access global variables during runtime execution.

Semantics

- The function resolves the global object located at `address` in the frame's global storage.
- The object is **not removed** from the global table.
- A new reference to the object is acquired.
- No garbage collection is triggered by this operation.

Ownership Model

- The global table holds an owning reference to each object.
- `Sframe_load_global` creates an additional reference for the caller.
- The caller is responsible for releasing the object when it is no longer needed.

Error Handling

- A runtime error is raised if `frame` is `NULL` .
- A runtime error is raised if `address` is out of bounds or refers to an uninitialized global.

Return Value

- Returns a pointer to the global `Sobj` .
- The returned object remains valid as long as at least one owning reference exists.

Typical Usage

- Loading global variables into the operand stack
- Resolving names at runtime
- Implementing global access bytecode instructions

Sframe_store_global

```
struct Sobj *
Sframe_store_global(struct Sframe *frame, int address, struct Sobj *obj,
```

Description

`Sframe_store_global` stores an object into the global storage of an execution frame at the specified address.

This function implements the runtime semantics of assigning a value to a global variable in the SUNY virtual machine.

Parameters

- `frame`
  The execution frame whose global storage is being accessed.
- `address`
  A numeric index identifying the global storage slot.
  The address is resolved by the compiler or bytecode generator.
- `obj`
  The object to be stored in the global slot.
- `type`
  The expected object type for the global slot.
  This is used to enforce type constraints during assignment.

Semantics

- The function writes `obj` into the global storage at `address` .
- If the target slot already contains an object, the frame releases its ownership of the previous object.
- The frame acquires ownership of the new object.
- The stored object is not pushed onto the operand stack.

Ownership Model

- Global storage holds an owning reference to stored objects.
- Storing an object transfers ownership to the global table.
- The caller remains responsible for any additional references it holds.

Type Handling

- The `type` parameter specifies the required runtime type of the object.
- If `obj` does not match the expected type, a runtime type error is raised.
- Type enforcement occurs at runtime, not during compilation.

Error Handling

- A runtime error is raised if `frame` is `NULL`.
- A runtime error is raised if `obj` is `NULL`.

Return Value

- Returns the stored `Sobj*`.
- The returned pointer refers to the object now owned by the global storage.

Typical Usage

- Assigning values to global variables
- Implementing global assignment bytecode instructions
- Updating runtime state during program execution

---

7 Extending Suny with C

7.1 Introduction

Suny is designed to be embedded and extensible. While the core language provides essential functionality, real-world applications often require capabilities beyond what the standard library offers—such as file I/O, networking, database access, or integration with existing C/C++ codebases.

The Suny C API allows you to extend the language by writing native C functions that can be called directly from Suny scripts. This approach offers several advantages:

- **Performance**: Computationally intensive operations can be implemented in C for maximum speed
- **Access to system resources**: File systems, networks, and hardware can be accessed through C libraries
- **Code reuse**: Existing C/C++ libraries can be wrapped and exposed to Suny
- **Custom functionality**: Domain-specific features can be added without modifying the interpreter

How It Works

Extending Suny involves three main steps:

1. **Write C functions** following the Suny calling convention
2. **Register functions** with the Suny interpreter using the API

3. **Call functions** from Suny scripts just like built-in functions

For example, after creating and registering a C function `sqrt()` , you can use it in Suny:

```
import "math"

result = sqrt(16);   // returns 4
print(result);
```

The Suny interpreter handles all the details of calling your C code, passing arguments, and returning results.

What You'll Learn

This chapter shows you how to:

- Write C functions callable from Suny
- Handle different data types between C and Suny
- Register your functions with the interpreter
- Build complete extension libraries
- Handle errors properly in C extensions

By the end of this chapter, you'll be able to create your own libraries to extend Suny's capabilities for any purpose.

7.2 Writing C Functions for Suny

Function Signature

All C functions callable from Suny must follow this signature:

```
SUNY_API struct Sobj* Sadd(struct Sframe *frame);
```

Where:

- `SUNY_API` - To know that this function is a Suny CAPI function
- `Sframe* frame` - Pointer to the stack frame of Suny
- Returns `Sobj` - The result to return to Suny

Basic Example

Here's a simple function that adds two numbers:

```
SUNY_API struct Sobj* Sadd(struct Sframe* frame) {
    struct Sobj* b = Sframe_pop(frame);      // Get the second argument
    struct Sobj* a = Sframe_pop(frame);      // Get the first argument

    struct Sobj* c = Seval_add(a, b);        // Add the two arguments usir

    MOVETOGC(a, frame->gc_pool);             // Move to the garbage pool,
    MOVETOGC(b, frame->gc_pool);             // Same

    return c;                                // return the result, the res
}
```

After compilation, the output can be built as a shared library (`.dll` on Windows or `.so` on Linux). The `load` function allows you to dynamically load the library and invoke an exported function:

```
load("my_lib.dll", "Sadd", [1, 2])
```

The array `[1, 2]` represents the function arguments in order.

7.3 Registering Functions and Build Your Own Library

The Registration Function

To make your C functions available in Suny, you need to create a `Smain` function that registers all your functions:

```
SUNY_API struct Sframe* Smain(struct Sframe* frame, struct Stable *table)
    Sinitialize_c_api_func(frame, table, 20, 2, "add", Sadd);
    return frame;
}
```

Parameters:

- `frame` - The current stack frame
- `table` - The symbol table where functions are registered
- Must return the `frame`

Registration Function Details

The `Sinitialize_c_api_func` function registers a C function:

```
Sinitialize_c_api_func(frame, table, hash_value, arg_count, "name", funct
```

Parameters:

- `frame` - Current stack frame
- `table` - Symbol table
- `virtual address` - virtual address for the function name (use any unique number)
- `arg_count` - Number of arguments the function expects
- `"name"` - Function name as it appears in Suny
- `function_pointer` - Pointer to your C function

Complete Example: Math Library

Here's a complete library with multiple functions:

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

// Add function
SUNY_API struct Sobj* Sadd(struct Sframe* frame) {
    struct Sobj* a = Sframe_pop(frame);
    struct Sobj* b = Sframe_pop(frame);
    struct Sobj* c = Seval_add(b, a);
    MOVETOGC(a, frame->gc_pool);
    MOVETOGC(b, frame->gc_pool);
    return c;
}

// Subtract function
SUNY_API struct Sobj* Ssub(struct Sframe* frame) {
    struct Sobj* a = Sframe_pop(frame);
    struct Sobj* b = Sframe_pop(frame);
    struct Sobj* c = Seval_sub(b, a);
    MOVETOGC(a, frame->gc_pool);
    MOVETOGC(b, frame->gc_pool);
    return c;
}
```

```
// Square root function
SUNY_API struct Sobj* Ssqrt(struct Sframe* frame) {
    struct Sobj* arg = Sframe_pop(frame);
    double value = Sobj_to_number(arg);
    struct Sobj* result = Sobj_make_number(sqrt(value));
    MOVETOGC(arg, frame->gc_pool);
    return result;
}

// Register all functions
SUNY_API struct Sframe* Smain(struct Sframe* frame, struct Stable *table)
    Sinitialize_c_api_func(frame, table, 20, 2, "add", Sadd);
    Sinitialize_c_api_func(frame, table, 21, 2, "sub", Ssub);
    Sinitialize_c_api_func(frame, table, 22, 1, "sqrt", Ssqrt);
    return frame;
}
```

Building Your Library

On Windows:

```
gcc -shared -o mathlib.dll mathlib.c -I/path/to/suny/include
```

On Linux:

```
gcc -shared -fPIC -o mathlib.so mathlib.c -I/path/to/suny/include
```

Using Your Library in Suny

After building, you can use your functions:

```
import "mathlib"    # import your .dll or .so file here

print(add(1, 2))
print(sub(1, 2))
print(sqrt(4))
```

Important Notes

1. **Argument count** must match the actual number of arguments your function expects
2. **Smain** is automatically called when the library is loaded
3. All functions must follow the `SUNY_API struct Sobj* FuncName(struct Sframe* frame)` signature

---

8 End

Suny is a small embedded interpreter language built with simplicity and clarity in mind.
This document has described its core structure, execution model, C API, and extension mechanism.

The goal of Suny is to provide a lightweight scripting engine that can be embedded into other applications. It is designed to be easy to understand, easy to integrate, and easy to extend.

Suny keeps its core minimal while allowing developers to add native functions and extend the language through its API. This makes it suitable for small tools, experiments, and learning purposes.

Future improvements may include:

- Garbage collection enhancements
- Standard library expansion
- Module system improvements
- Debugging and tooling support

Suny is not intended to be a large production language. Instead, it serves as a compact and educational interpreter project that demonstrates how an embedded scripting language can be implemented.

This concludes the documentation of the Suny embedded interpreter language.

- Document made by dinhsonhai132 (Đinh Sơn Hải)