0 – Introduction

This reference manual describes and defines the Sophia programming language. Sophia is a high-level procedural programming language oriented around a robust, user-extensible type system.

High-level

The interpreter handles technical, implementation-specific processes such as memory management and presents an abstracted interface with which the user can interact. The user should rarely, if ever, have to concern themself with the physical hardware on which the program runs. A modern interpreter should be sufficiently powerful as to efficiently automate these processes.

Procedural

Programs are composed of procedures, sequences of commands that change the state of the program. Procedures can be stored as data and operated on by other procedures. The fundamental structures of the language control the order of execution of procedures.

Robust

Sophia is strongly, dynamically typed. Data values are tested at runtime against type constraints to ensure that operations are only ever performed on data of a valid type. The interpreter never performs type coercion – the user must explicitly cast data.

User-extensible

A type is defined simply as a set of constraints that a value must satisfy and a set of operations that can be performed upon a value of its type. With this in mind, it is easy for the user to define their own types.

0.1 Implementation

This reference is written using natural language for the benefit and understanding of the user. Should this create ambiguities or difficulties in implementation, more exact definitions will be provided. This reference defines Sophia; therefore, any implementation that exactly reproduces the behaviours defined by this reference is a valid implementation of Sophia.

0.2 “Hello, world!”

The example program below demonstrates a classic “Hello, world!” program:

```
print(‘Hello, world!’)
```

That’s literally it. That’s literally all you have to do. You get the point. High-level programming, baby!

1 – Program structure

The Sophia interpreter takes as its initial input a plain text file with the file name *main.sophia*. In the parsing stage of execution, the input undergoes two passes: lexical analysis, which generates a sequence of tokens, and parsing, which generates a tree of tokens.

1.1 Line structure

1.1.1 Logical lines

A logical line is a non-empty sequence of characters terminated by a newline character. A logical line contains a single statement – statements cannot cross logical line boundaries, nor can multiple statements be written on the same line.

1.1.2 Comments

A comment starts with two forward slashes (//) occurring outside of a string. A comment signifies the end of a logical line, and is ignored by the parser.

1.1.3 Blank links

Blank lines – lines containing only whitespace characters or a comment – are ignored by the parser.

1.1.4 Indentation

Indentation is a syntactic feature that determines the grouping of statements. Indentation is represented by tabs; the number of tabs before the beginning of the statement indicates the level of indentation. Spaces cannot be used for indentation; the parser rejects files written in this way.

Some statements have a statement "body" associated with them; this body must be indented exactly one level more than the head statement. Below is an example of a correctly indented program.

```
for i in range(0, 11):
	if i % 2 = 0:
		print(fib(i))
	else:
		continue
	print('!')
```

Here is the same program with indentation errors:

```
for i in range(0, 11):
		if i % 2 = 0: // Cannot increase indent by more than 1
		print(fib(i)) // Expected indent
	else:
	continue // Expected indent
		print('!') // Unexpected indent
```

1.1.5 Whitespace

Spaces are used as whitespace that separates tokens. Tabs cannot be used for this purpose, since they are parsed as indicating indentation. Whitespace is not necessary before or after operators, parentheses, and some special characters; it is necessary in all other instances.

2 – Data model

2.1 Values

Values are stored in name bindings. A name binding contains a name, a reference to a value, and a reference to a type. The value itself has no intrinsic type at runtime – the type is associated with the name – but practical limitations require that the value conform to the constraints of at least one built-in type.

All values are immutable. New values cannot be created by modifying existing values; instead, they can be constructed using operations that operate on existing values and return new values. Once a value is bound to a name, it remains accessible until a different value is bound to the same name, and it remains in memory until the name and all aliases have fallen out of scope.

2.2 Types

A type is defined as a set of constraints that a value of its type must conform to and a set of operations that can be performed upon a value of its type. The hierarchy of built-in types follows below; the type operations for built-in types are described in the reference for the standard library. A name associated with a given type permits the application of the type operations of that type and of all supertypes to the bound value.

2.2.1 untyped

This type is the base type and the default type that is assumed when a binding is created without a specified type. It has no constraints and no type operations.

2.2.2 number

This type represents all numbers. Values of this type are valid operands for arithmetic operators.

2.2.2.1 integer

This type represents the set of integers from negative to positive infinity. Integers are a valid type for arithmetic operators.

2.2.2.2 float

This type represents the set of floating-point numbers as defined by IEEE-754. Floats are a valid type for arithmetic operators.

2.2.2.3 bool

This type represents the Boolean values *true* and *false*. Booleans are a valid type for logical operators but *not* arithmetic operators.

2.2.3 null

This type represents *null*, the empty value. The null type has no type operations, but untyped assertions will fail for a name that evaluates to *null*.

2.2.4 sequence

This type represents sequence data structures. Sequences contain a non-negative number of values, each of which can be accessed using a unique index. For all sequences except records, this index is an integer in the range `0 <= i < n`, where *n* is the length of the sequence. For records, this index is a string matching one of the record's keys. For a sequence *a* and index *i*, the index syntax is `a[i]`. All sequences are valid types for sequence operators and valid targets for iteration.

Sequences with integer indices support slicing. For a sequence *a* and indices *i* and *j*, the syntax `a[i:j]` creates a sequence of the same type containing all items with index *n* such that `i <= n <= j`. Note that both *i* and *j* must be valid indices; i.e., they must be within the bounds of the sequence. In addition, a "step" can be defined: with a step *k*, the syntax `a[i:j:k]` creates a sequence of the same type containing all items with index *x* such that `x = i + n*k` and `i <= n <= j`.

Sequences with integer indices also support negative indices: the index -1 refers to the last value in the sequence, the index -2 refers to the second-last value in the sequence, and so on.

2.2.4.1 string

This type represents strings as a sequence of values of Unicode code points. The values themselves are not directly accessible; accessing a single character in a string using a sequence index instead yields another string of length 1.

2.2.4.2 list

This type represents the list data structure. Because of the immutability of values in Sophia, and since sequences are considered to be values themselves, lists are functionally arrays.

2.2.4.3 record

This type represents the record data structure. Each item in a record is a key of string type that references a value. Iterating over a record yields its keys but not its values.

2.2.5 function

This type represents procedures, which in Sophia are functionally identical to functions. Note that when a function is defined, its name binding stores the type of its return value, and not the type of the function itself. Therefore, a name binding with the integer type may reference an integer *or* a function that returns an integer, and a name binding with the function type may reference a function *or* a function that returns a function. In order to use a function's type operations, it may be necessary to alias it as a function or assert the function type for it.

2.2.6 type

This type represents types.

2.2.7 module

This type represents imported modules. Modules contain any functions that they define in their own namespaces.

3 – Expressions

3.1 Values

Below follow the various ways of representing values within expressions.

3.1.1 Literals

Numbers and strings are interpreted as literals, and evaluate to the values they represent. Strings are written enclosed in single or double quotation marks.

```
1 // Initialises an integer
1.1 // Initialises a float
“1.1” // Initialises a string
```

3.1.2 Names

Names are evaluated as the values to which they are bound. If the interpreter encounters an unbound name in an expression, it throws an error.

```
a // Name
```

3.1.3 Parentheses

3.1.3.1 Arithmetic brackets

Arithmetic brackets are written with parentheses and indicate precedence of operations in an expression. The contents of arithmetic brackets are evaluated as their own sub-expression.

```
(1 + 2) // Arithmetic brackets
```

3.1.3.2 Sequence definition

A sequence definition is written inside square brackets. Values in a sequence definition are separated by commas. If the interpreter detects the use of the colon operator to define key-value pairs, it creates a record; otherwise, it creates a list.

```
[1, 2, 3] // List definition
[‘a’: 1, ‘b’: 2, ‘c’: 3] // Record definition
```

3.1.3.3 Meta-statement

The meta-statement is Sophia’s primary tool for meta-programming. It is written inside braces; the expression inside the braces must evaluate to a string. When the interpreter encounters a meta-statement, it attempts to parse the string as a valid expression or statement and evaluate or execute it accordingly.

```
{‘for i in range(10):’ | ‘\n\t’ | ‘print(i)’} // Meta-statement
```

3.2 Operators

3.2.1 Arithmetic operators

Arithmetic operators are valid for *number*. For mixed *integer* and *float*, operators return *float*. Note that unary operators have higher precedence than exponentiation.

Syntax	Operator	Operand type
`+a`	Unary positive	number
`-a`	Unary negative	
`a + b`	Addition	number, number
`a - b`	Subtraction	
`a * b`	Multiplication	
`a / b`	Division	number, number (> float)
`a ^ b`	Exponentiation	number, number (right-binding)
`a % b`	Modulo	number, number

3.2.2 Sequence operators

Sequence operators are valid for *sequence*. Both operands must be, or must be subtypes of, the same built-in type. Intersection creates a new sequence containing only the values or keys shared by *a* and *b;* items are ordered by their order in *a*. Union creates a new sequence concatenating *a* and *b;* it throws an error if the operands are records and share any keys.

Syntax	Operator	Operand type
`a & b`	Intersection	sequence, sequence
`a | b`	Union	

3.2.3 Logical operators

Logical operators (except for *in)* are valid for *bool*. Sophia does not have "truthiness" of values; these operators are only valid for Booleans.

Syntax	Operator	Operand type
`not a`	Logical not	bool
`a and b`	Logical and	bool, bool
`a or b`	Logical or	
`a xor b`	Logical xor	
`a in b`	Membership	untyped, sequence

3.2.4 Comparison operators

The equality and inequality operators are valid for *untyped*; other comparison operators are valid for *number*.

Syntax	Operator	Operand type
`a = b`	Equality	untyped, untyped
`a != b`	Inequality	
`a < b`	Less-than	number, number
`a > b`	Greater-than	
`a <= b`	Less-than-or-equal	
`a >= b`	Greater-than-or-equal	

3.2.5 Special operators

Special operators have important syntactic functions, as described below, and are all right-binding.

`a.b()`	Dot operator	Associates a name and a type operation or a module and a function.
`(a, b)` `[a, b]`	Comma operator	Groups together values as parameters, as arguments, or in sequence definition.
`[a: b]` `[a:b:c]`	Colon operator	Defines a key-value pair in a record definition, or defines a list slice.

3.2.6 Subscription

The subscription of a name of type *sequence* selects and returns an item from the sequence, as described in 2.2.4. The syntax follows below:

```
a[i] // Returns the element at index i
a[i:j] // Creates a list of the elements from i to j
a[i:j:k] // Creates a list of every kth element from from i to j
```

3.2.7 Function calls

A function call calls a function with zero or more arguments. Arguments are comma-separated. The number of arguments provided must be the same as the number of parameters associated with the function. Some built-in functions can take a variable number of arguments; user-defined functions cannot.

When a function is called, a new scope is created. The function binding is copied into the new scope, and the arguments provided in the function call are bound to their parameters. The body of the function's definition is then executed.

Sophia uses tail call optimisation. If the interpreter detects a tail call, it will overwrite the current scope instead of creating a new scope.

```
a(b, c) // Function call
```

3.3 Operator precedence

The precedence of operators is given below, from highest (strongest binding) to lowest (weakest binding). Operators in the same row have the same precedence and are parsed from left to right.

Operator	Name
`+a` `-a`	Unary operators
`.`	Dot operator
`^`	Exponentiation

`*`
`/`
`%`
	Multiplication, division, modulo

`+` `–`	Addition, subtraction
`&` `|`	Intersection, union

`<`
`>`
`<=`
`>=`
	Comparison operators

`=` `!=`	Equality operators
`and`	Logical and
`or`	Logical or
`:`	Colon operator
`,`	Comma operator

`(…)`
`[…]`
`{…}`
	Parentheses

4 – Single-line statements

4.1 Expression

Expressions can themselves be statements. This usually occurs for function calls that return *null* and have side effects, as shown below.

```
print(‘Hello, world!’) // Expression statement
```

4.2 Assignment

An assignment binds a value and a type to a name, as shown below.

```
a: 1 // Untyped assignment
integer a: 1 // Typed assignment
```

If there exists no name binding in the current namespace with the specific name, one is created. Otherwise, the existing name binding is updated. The name binding is updated with a reference to the specified value. Because of how values are stored in Sophia – values only ever have one reference – if this overwrites an existing value, that value can be destroyed immediately.

If the assignment was typed, the name binding is also updated with a reference to the specified type. If the assignment was untyped, a newly created name binding is bound with a reference to *untyped*, and an already existing name binding is not updated; that is, it retains the already stored type. In order to remove the type from a name binding, an assignment must be explicitly labelled as *untyped*.

4.3 Aliasing

Aliasing creates an alias for a name, as shown below.

```
b is a // Untyped alias
float b is a // Typed alias
```

An alias creates a name binding whose associated value is a reference to another name binding. An alias cannot reference a name binding outside of its scope, and aliases cannot be chained – an alias cannot reference another alias. Evaluating an alias yields the same value as evaluating the aliased name, and vice versa; assigning to an alias and assigning to the aliased name have the same result.

Aliases are useful in that they can be bound with a different type to the name that they reference. Thus, it is possible to treat a value as having multiple different types, and consequently it is possible to perform type operations from different types on the same value, if it satisfies the constraints for these types.

4.4 The “pass” keyword

The *pass* keyword is a null operation; executing it does not affect the state of the program in any way.

```
pass // Does nothing
```

4.5 The “continue” keyword

The *continue* keyword, when executed in a while loop or a for loop, immediately jumps to the beginning of the next iteration of the loop.

```
while true:
      print('1') // Prints ‘1’ forever
      continue
      print('2') // Never prints ‘2’
```

4.6 The “break” keyword

The *break* keyword, when executed in a while loop or a for loop, immediately terminates the loop.

```
while true:
      print('1') // Prints ‘1’ once
      break
      print('2') // Never prints ‘2’
```

4.7 Return statement

A return statement evaluates the expression following it and immediately returns out of the enclosing function, yielding the evaluated value as a return value. If the function is typed, it also checks the type of the return value.

```
integer a ():

      return 1 // Returns 1

print(a()) // Prints ‘1’
```

The return statement can only be used inside a function definition. If a function definition has no return statement, it returns *null* upon executing the last statement in the function’s control flow.

4.8 Import statement

The import statement finds at least one named module and binds it to the current scope, as shown below.

```
import a, b, c // Imports the modules a.sophia, b.sophia, c.sophia
```

When this statement is executed, the interpreter takes these steps for each named module:

- Find a file with the specified name and the file extension *.sophia*.
- Create a new instance of the interpreter and execute the file with a restricted subset of instructions – the interpreter ignores any statement that is not a function definition or a type definition.
- Bind the resulting namespace to the current scope of the main namespace.

Every function and type defined in the imported modules then becomes available in the main namespace.

5 – Multiple-line statements

Multiple-line statements consist of two parts. The first line of the statement is the head. It contains information specific to the statement; it starts with a keyword or a name and ends with a colon. Every following line with a higher indentation level than the head comprises the body.

5.1 If statement

The if statement is a structure for conditional control flow. An if statement takes an expression that evaluates to *bool*. If the expression is false, the body is passed and the next statement is executed. If the expression is true, the body is executed and all subsequent else statements are passed.

```
if false:
      print(‘1’) // Passed
else if true:
      print(‘2’) // Prints ‘2’
else:
      print(‘3’) // Passed
```

5.2 While statement

The while statement is a structure for conditional iteration. A while statement takes an expression that evaluates to *bool*. If the expression is true, the body is executed and the condition is evaluated again. If the expression is false, the body is passed and the loop ends.

```
a: 0
while a < 10:
      print(‘!’) // Prints ‘!’ 10 times
      a: a + 1
```

5.3 For statement

The for statement iterates over a sequence. It takes a name, the loop index, and a value of type *sequence*. For every item in the sequence, the item is bound to the index and the body is executed. When every item in the sequence has been used, the else statement (if present) is executed, the index is unbound, and the loop ends.

```
a: [1, 2, 3]
for i in a:
      print(i) // Prints ‘1’, then ‘2’, then ‘3’
```

The loop index is reserved, and cannot be rebound by the user.

5.4 Assertion

An assertion is a structure that asserts the existence and type of a given name. An untyped assertion checks that the name is bound and that its value does not evaluate to *null*. A typed assertion checks that the name is bound and that its value is of the specified type. If these checks succeed, the body is executed. If these checks fail, the else statement (if present) is executed. Its syntax is as follows:

```
a: 1
assert a: // Untyped assertion
```

```
      print(a) // Prints ‘1’
assert integer a: // Typed assertion
      print(a + 1) // Prints ‘2’
```

In a typed assertion, it is possible to assert that a name’s value is of a subtype of the type bound to the name. In this case, the type bound to the name is changed to that subtype during execution of the body, and then reverted after the body ends.

```
a: [1, 2, 3]
assert list a:
      print(a.length()) // Prints ‘3’
print(a.length()) // Error
```

Note that an untyped assertion is not the same as a typed assertion of type *untyped*. This difference can be seen when asserting a name that evaluates to *null*. An untyped assertion fails because *a* evaluates to *null*, but a typed assertion succeeds because *null* is a subtype of *untyped*.

```
a: null
assert a:
      print(‘!’) // Passed
assert untyped a:
      print(‘!’) // Prints ‘!’
```

5.5 Function definition

A function definition defines a user-defined function.

The head of the function definition defines the function’s name, the type of its return value, the function’s parameters, and the type of each parameter. The body of the function definition is the code that is executed when the function is called.

When a function definition is executed, the name of the function is bound to the local scope with the definition as its value and the type of its return value as its type. The body of the definition is not executed during the definition.

The syntax for a function definition follows as such:

```
integer a (integer n, integer m):

      return n + m
```

5.6 Type definition

A type definition defines a user-defined type.

The head of the type definition defines the type’s name and supertype, if specified. Specifying a supertype indicates that the constraints and type operations of the supertype are valid for the defined type. The body of the type definition is the code that is executed when the type is checked.

When a type definition is executed, the name of the type is bound to the local scope with the definition as its value and *type* as its type. The body of the definition *is* executed during the definition in order to bind type operations to the type; however, only function definitions are executed.

The base syntax for a type definition follows as such:

```
type a:

      pass

type b extends a:

      pass
```

An example of a type definition follows:

```
type even extends integer:

      constraint:
            even % 2 = 0

      integer half (even):

            return even / 2

a: 2
assert even a:
      print(a.half()) // Prints 1
```

The name of the type definition acts as a reserved name that references the checked value – in this example, within the type definition, *even* is a copy of the value bound to *a*. When the type definition is executed, the interpreter finds all function definitions whose first parameter is the reserved name – in this case, *even* – and binds them to the type definition as type operations. The behaviour of the reserved name is similar to the behaviour of the *self* keyword in many object-oriented languages.

When a value is checked for type, a new scope is created, a reserved name binding is created binding the checked value to the name of the type, and the body of the type definition is executed. The execution of the body is considered to be equivalent to checking the value's type; if it succeeds, the value is of the given type, and if it fails, the value is not of the given type. If the type extends a supertype, the value is tested for all supertypes in sequence before being tested for the given type. The user can define constraints to test the value's properties, as explained in 8.6.1.

Values cannot be typed with the type currently being defined, as shown below.

```
type a extends integer:

      a b: 1 // Invalid typing; a is not yet defined
```

5.6.1 Constraints

Constraints are structures used in type definitions that test the properties of the value being checked for type. The head consists of the keyword *constraint*. The body consists of one or more expressions that evaluate to *bool*. If an expression evaluates to *false* and the type check is not being performed for a typed assertion, an error is thrown. If an expression evaluates to *false* and the type check is being performed for a typed assertion, the assertion fails. The syntax for a constraint follows below:

```
constraint:
      a % 2 = 0
      a > 0
      a in b
```

5 – Example programs

This section contains example programs that demonstrate idiomatic use of Sophia's features.

5.1 Prime number finder

This program prints a list of prime numbers from 2 to the highest prime within the specified range, using several optimisations to increase performance for higher values.

```
list found_primes: [2]

type prime extends integer:

	constraint:
		prime > 1

	for i in found_primes: // Possible because of the fundamental theorem of
arithmetic
		if i > (prime ^ 0.5) + 1: // Possible because of symmetry of factors
			break
		constraint:
			prime % i != 0

for i in range(1, 1000, 2): // Skips even numbers
	assert prime i:
		found_primes: found_primes | [i]

print(found_primes)
```