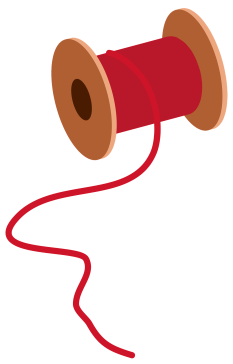
The sPool Programming Language

A General-Purpose Programming Language with Concurrency Support



Language Reference Manual

Team Nautilus

Ankur Dahal
ankur.dahal@tufts.edu

Max Mitchell
maxwell.mitchell@tufts.edu

Yuma Takahashi
yuma.takahashi@tufts.edu

Etha Hua
tianze.hua@tufts.edu

Table of Contents

1 Introduction

sPool is a general-purpose, statically-typed programming language. It supports and incorporates multiple language features, including concurrent programming via multithreading, automatic memoization for dynamic programming, and functions as first-class citizens. This document is the primary reference for the sPool Programming Language and acts as the official documentation for the language, describing its features and functionalities in detail.

This manual begins with a note on the conventions used throughout the document that help the reader understand and distinguish between code, prose, and formal grammar (section **2.1**). It then goes on to explain the lexical conventions of the language (section **2.2**), followed by the essential language features like the types used in sPool (section **4**), permitted arithmetic and logical operations (section **6**), and expressions and statements (sections **7** and **8**) that act as the primary building blocks for a complete sPool program. Finally, this manual introduces the built-in functions that are provided to users when they write a sPool program (section **10**), followed by the functions included in the sPool standard library (section **11**).

2 Conventions

2.1 Manual Conventions

Throughout this manual, text written in black, Arial font should be taken as prose. Headings for prose sections will be numbered, and appear in **bold black**. Other text written in `Roboto Mono font` takes on special meanings, depending on the style and color. Text written in *`italics and orange`* indicate rules that are defined elsewhere in the grammar. They link to that section of the grammar and are clickable. Text written in **`bold and blue`** indicate keywords with explicit meaning set out in section **2.2.2**. Text written in `plain black` indicates literal text to be written in source code. Text written in **`bold and red on off-white background`** indicate regular expressions used to describe rules. Some rules are described partially by regular expressions, and partially not. Note that within regular expressions, text coloring for *`rules`* and **`keywords`** takes precedence over **`red`** coloring, but all regular expressions have the `off-white background`. Code examples, which appear both inline with text and in chunks below prose, are indicated by `white text on a black background`, and may include some syntax highlighting for readability – the common theme across code examples is that they are always written on a black background. For instance, comments included within these code examples are highlighted `green on a black background`, and so on for readability.

2.2 Lexical Conventions

sPool is not a free-form language, meaning that certain delimiters have special semantic meanings in sPool. Specifically, these special delimiters are discussed in section **2.2.1** below with their semantic significance.

2.2.1 Comments and Other Special Characters

Comments in code are indicated using the hashtag character, `#`, and create a single line comment. Anything after this character is ignored by the compiler until a newline character is seen. Multi-line comments are not supported in sPool. An example of a single line comment in sPool is:

```
# Hello world! This is a single line comment in sPool...
```

Other special characters like spaces and tabs can be used to add horizontal spacing and indentation in the code and they have no semantic significance.

As mentioned above, the delimiters in sPool have semantic significance. Users can write sequences of *`statement`*s (in a form called *`statement_list`*) separated by delimiters. These delimiters are therefore semantically significant in only this context (i.e. separating multiple *`statement`*s in *`statement_list`*) and no more. Delimiters are simply one or more newline characters:

```
delimiter ::= \n+
```

To read more about delimiters and their roles in separating statements, please see section **8**.

2.2.2 Reserved Words and Identifiers

The following keywords, literal values, and type names are reserved, and may not be used as variable names or function names. The first row indicates the section of this document with more information about that column of reserved words.

statement		*expr*	*type*		*literal*
`def`	`while`	`lambda`	`int`	`quack`	`true`
`return`	`if`		`float`	`thread`	`false`
`store`	`else`		`string`	`mutex`	
			`bool`	`list`	

There are also certain names used in our built-in functions, listed in section **10**, which may not be used either as identifiers. With these exceptions noted, the user may define variable and function names starting with uppercase or lowercase alphabet and followed by any number of alphanumeric characters and/or the underscore character.

Formally, this rule can be defined by the following grammar using regular expressions:

`name ::= [a-zA-Z][0-9a-zA-Z_]*`

3 Scope

Scope is the area of the program where a named item is recognized. In sPool, the scope of a `name` is based on two factors.

The first is the position of the name relative to other code. Names are in scope for code that comes later in the program.

The second has to do with the delimiters colon, `:`, and semicolon, `;`. These delimiters introduce a new scope in certain contexts. See sections **7** and **8** on `expr` and `statement` to learn more about these contexts. Every colon is matched by a corresponding semicolon. All names introduced after a colon will go out of scope after the corresponding semicolon.

This second rule supersedes the first rule, meaning that if a semicolon has caused a `name` to go out of scope, then that `name` is out of scope for all code following the semicolon, regardless of the first rule.

For names that are already defined, defining a new `name` in a sub-scope will shadow the original `name`. As long as the new definition is in scope, references to that `name` will refer to the new definition. Once the new definition is out of scope, references to that `name` will refer to the original definition (provided the original definition is still in scope).

For the sake of example, we will use if statements, variable definition, and variable evaluation. Please see section **8** to learn more about `statement`s, and section **7** to learn more about `expr`s. See below:

```
# since x and y are defined first, they are
# in the “global” scope for all code that follows
int x = 12
int y = 7

x # evaluates to 12
y # evaluates to 7
if (true): # entering new scope
    x = 15 # not a redefinition, just a reassignment, not a shadow
    int y = 10 # new definition shadows old definition
    int z = 13 # new name is defined
    x          # evaluates to 15
    y          # evaluates to 10
    z          # evaluates to 13
; # exit the scope
x # evaluates to 15
y # evaluates to 7
z # causes error, name is not in scope and undefined
```

4 Types

sPool supports several primitive and non-primitive types. Formally, all the types supported by sPool can be defined by the following rule:

```
type ::= int
      |  float
      |  bool
      |  string
      |  quack
      |  thread
      |  mutex
      |  (type(, type)* -> type)
      |  list<type>
```

4.1 Integers, Floats, and Booleans

`int`, `bool`, and `float` are primitive types in sPool. A value having the `int` type is an integer value in sPool. No guarantees are made regarding integer limits in sPool; they are platform-dependent. Similarly, a value having the `float` type is a floating point number in sPool. Like integer values, the precision and range of such floating point numbers are completely platform-dependent. Values having the type `bool` represent Boolean values in sPool.

4.2 Strings

sPool also supports strings as primitive types. As the name suggests, a `string` value in sPool represents a sequence of characters. To learn more about the built-in functions in sPool associated with strings, see section **10.3**. To learn more about the standard library functions in sPool associated with strings, see section **11.2**.

4.3 Quack

`quack` is a primitive type in sPool which is used to indicate the absence of a value like null or void in other languages. Due to this special nature of `quack`, a value having the `quack` type does not exist in sPool. It is mostly used in arrow types for indicating the absence of the argument or the absence of the return value.

4.4 Threads and Mutex

sPool provides the `thread` and `mutex` primitive types to support concurrency. In particular, a value having the `thread` type represents an instance of a thread, which can be considered to be an independent unit of program execution. Multiple thread instances can be executed concurrently with respect to each other. Each thread value has its own stack but can access and modify global variables at any given time.

As a synchronization primitive, the `mutex` type is introduced. A value of the `mutex` type represents a lock that can enforce limits on access to certain code regions when there are multiple threads of execution.

The values of both `thread` and `mutex` types can be used by built-in functions that allow users to interact with them and ultimately introduce concurrent program control flow in a sPool program. These are discussed in more detail in sections **10.4** and **10.5** respectively.

4.5 The Arrow Type

```
        |   (type(, type)* -> type)
```

The arrow type is a special, non-primitive type in sPool that is used to denote types of functions. For more details about calling functions, anonymous functions, and defined functions in sPool, see sections **7.2**, **7.3**, and **8.4**, respectively. Since functions are first-class citizens in sPool, it is important to be able to have explicit types for them.

As shown in the rule, the arrow “`->`” separates the types of the arguments from the return type of the function. Neither side of this arrow can be empty; if a function does not take an argument or does not return anything, the type `quack` should be used instead.

For example, a function that takes in an integer value and does not return anything has the type `(int -> quack)`, a function that takes in an integer and a boolean value and returns a thread value has the type `(int, bool -> thread)`, and a function with no arguments that returns a function that takes in an integer and returns a boolean has the type `(quack -> (int -> bool))`.

The ordering of the arguments is also significant. A function with the type `(int, bool -> quack)` expects that the first argument passed will be an integer, and the second a boolean. On the other hand, a function with the type `(bool, int -> quack)` expects that the first argument passed will be a boolean, and the second an integer.

4.6 Lists

```
       |   list<type>
```

The list type is another non-primitive type in sPool that is used to denote types of lists. When lists are declared, they must be given some type. Lists are zero-indexed, meaning the first element of any list has the index 0. Lists are homogeneous, and can only contain values of a single type. For example, a list of boolean values in sPool has the type `list<bool>`, and a list of list of integer values has the type `list<list<int>>`. To learn more about the built-in functions in sPool associated with lists, see section **10.6**. To learn more about the standard library functions in sPool associated with lists, see section **11.1**.

5 Literals

sPool provides literal values for strings, floats, integers, booleans, and lists. Formally, the following rule shows all literals supported by sPool:

```
literal ::= [0-9]+[\.]?[0-9]*
         |  \"(\\.|[^\n\"])*\"
         | true | false
         | [(expr(, expr)*)?]
```

In sPool, integer literals represent a base 10 number and may be any combination of digits 0 through 9. Leading zeroes will automatically be ignored.

Floating point values must start with at least one digit, followed by zero or more digits, followed by a dot, followed by zero or more digits.

String literals may contain zero or more characters from the set of all Unicode characters *except* the newline character and the double-quote character. Double-quotes and newlines may be represented by escaping them with the backslash. See below example demonstrating three valid string literals in sPool:

```
“text \
“
“str \””
“\”
```

The first literal contains the string `“text”` followed by a space and a newline character. The second literal contains the string `“str”` followed by a space and a double quote character. In these two instances, the backslash character precedes an otherwise illegal string character, so it is used to escape that character. The third literal contains just the backslash character.

Boolean literals are represented with the reserved words `true` and `false` (case sensitive).

List literals are created with an opening and closing square bracket pair, containing zero or more comma-separated *expr*s, which all must be of the same type. For more information about *expr*s, see section **7**. Semantically, the empty list literal `[]` is treated as a polymorphic list when used on its own; it can and will be safely changed to the empty list literal of any other type during runtime depending on the context. Some valid sPool list literals are shown below:

```
[] # an empty list, initially of type list<quack> but is implicitly
   # cast to be list of other types depending
   # on the context at runtime
[1, 2] # a list of integers (having type list<int>)
[["hi"], [""]] # a nested list of type list<list<string>>
```

6 Basic Operations

sPool supports basic arithmetic and logical operations. The operations are classified into two broad categories depending on the number of operands involved: unary and binary operations.

6.1 Unary Operations

Unary operations operate on a single operand. Formally, the unary operators supported by sPool are shown by the following rule:

```
unop ::= !
      |  -
```

`!` stands for the boolean negation operator, which expects an operand of type `bool`, and evaluates to a value having type `bool`.

`-` stands for the arithmetic negation operator, which expects an operand of type `int` or `float`, and evaluates to a value having the same type as its operand.

Both unary operators `!` and `-` have the highest precedence among all operators in sPool. Relatively, they have the same precedence and are both right-associative.

6.2 Binary Operations

Binary operations operate only on two operands of the same type. Formally, the binary operators supported by sPool are shown by the following rule:

```
binop ::= +
       |  -
       |  *
       |  /
       |  %
       |  >
       |  >=
       |  ==
    |   !=
    |   <=
    |   <
    |   &&
    |   ||
```

Arithmetic operators including `+`, `-`, `*`, `/`, `%` must have operands of the same type (`int` and `int` or `float` and `float`) and will evaluate to a value of that type. These operators perform arithmetic addition, subtraction, multiplication, division, and modulo respectively.

Comparison operators including `>`, `>=`, `<=`, `<` must have operands of the same type (`int` and `int` or `float` and `float`) and will evaluate to a `bool`. These operators perform arithmetic comparisons, namely greater than, greater than or equal to, less than or equal to, and less than respectively.

The following comparison operators: `!=`, `==` must have operands of the same type. These operators perform logical comparisons, namely non-equality and equality respectively. Equality is evaluated in terms of structural equality, *not* strict equality. This means the values of the operands are considered when comparing, but not whether the operands are stored in the same location in memory. The permitted types for the operands are all primitive types in sPool, namely `int`, `bool`, `float`, `string`, `thread`, and `mutex`. The comparison operators always evaluate to a boolean value.

Logical operators including `&&` and `||` must have operands that have types of `bool` and `bool`, and will evaluate to a value of type `bool`. These operators perform boolean AND and boolean OR respectively.

All binary operators are left-associative, meaning that the operations performed by them are grouped from left to right. The relative order of precedence for binary operations from greatest to least is:

1. `*`, `/`, `%`
2. `+`, `-`
3. `<`, `<=`, `>`, `>=`
4. `!=`, `==`
5. `&&`
6. `||`

Operands on the same line have equivalent precedence. As mentioned in section **6.1**, all unary operations have higher precedence than binary operations in sPool.

7 Expressions

Formally, expressions in sPool are defined by the following rule:

```
expr ::=  literal
       |   expr binop expr
       |   unop expr
       |   name
       |   name ((expr(, expr)*)?)
       |   lambda type ((type name(, type name)*)?): statement_list ;
       |   (expr)
```

All expressions except for function calls always evaluate to a value in sPool, which will have some *type*. For function calls, the expression may or may not evaluate to a value; it can instead return the `quack` type, which is not a sPool value. See section **7.2** for more information about function calls.

For more information on expressions containing *literal*, *unop*, and *binop*, see sections **5, 6.1,** and **6.2** respectively. In these sections, references to "operand" are the value received after evaluating the expression *expr*.

7.1 Variable Evaluation

```
    |    name
```

Writing the name of a variable which has been defined earlier will evaluate to a value. This value will have whatever type was given to this name when the name was defined. The *name* being evaluated must be defined previously in the code, or variable evaluation will fail. See section **8.1** for more on variable definition and assignment.

7.2 Function Call

```
    |    name ((expr(, expr)*)?)
```

Functions can be called using their name, followed by a set of parentheses. Any parameters expected by the function at the time of definition should be expressed at call time by *expr*s placed between these parentheses, separated by commas. For more information on defining functions, see section **8.4**. Arguments of all but list, thread, and mutex types are passed by value in sPool; lists, threads, and mutexes are passed by reference.

In the function call, the *expr*s passed must be of the same type as the parameters expected by the function's definition, or an error will occur. There must be the same number of *expr*s passed during the call as the parameters expected by the function's definition, or an error will occur. The *name* must be of a function defined and in scope at call-time, a built-in function, or a function from the standard library, otherwise an error will occur. For more information on scope, see sections **3**. For more information on built-in and library functions, see sections **10** and **11**, respectively.

```
# After defining the function name before its use, call it by passing
# in respective arguments
myFunc1()    # calls myFunc1 with no arguments
myFunc2(1)   # calls myFunc2 with an integer value passed as argument
```

7.3 Anonymous Functions

```
    |   lambda type ((type name(, type name)*)?): statement_list ;
```

`lambda` is a unique expression which evaluates to a value having the arrow type. After the keyword `lambda`, the first *type* indicates the return type of the function. If no return value is desired, the return type of `quack` should be given. Any parameters follow between parentheses. Each parameter given must include a *type* indicating the type of the expected parameter, and a *name* which will be used locally to refer to the parameter within the body of the function. Parameters are comma-separated. To indicate no parameters, use empty parentheses. The body of the function is a *statement_list*. To learn more about *statement_list* see section **8**. The body is delimited with a colon at the start and a semicolon at the end.

`lambda` expressions cannot be called at the site of definition. See the below *illegal* sPool code:

```
lambda bool (int i): return false;(5)
```

To call a lambda explicitly, assign it to a name, either with a variable assignment, or by passing it to a function. Then, call it using its name. See the below *valid* sPool code which fixes the above error:

```
(int -> bool) myFunction = lambda bool (int i): return false;
myFunction(5)
```

7.4 Parentheses

```
    |   (expr)
```

Parentheses can be used to set precedence of expression order for expressions occurring within a single larger expression. The *expr* within the parentheses will be evaluated first, just as parentheses are used in mathematical formulas. All valid expressions may have their precedence and associativity controlled with parentheses. See the below examples:

```
-(2 + 3)      # evaluates to -5
-2 + 3        # evaluates to 1
5 * (1 + 7)   # evaluates to 40
5 * 1 + 7     # evaluates to 12
```

8 Statements

Statements in sPool are *not guaranteed* to produce a value and may simply be used for side effects. It is important to note that all expressions are statements but not all statements are expressions.

```
statement ::= type name = expr
           |  name = expr
           |  if (expr): statement_list (else statement_list)? ;
           |  while (expr): statement_list ;
           |  def (store)? type name ((type name(, type name)*)?):
                    statement_list ;
           |  return (expr)?
           |  expr
```

Certain statements require an explicit type to be given. More information on types can be found in section **4**.

Statements can be sequenced to use imperative programming features. This form is called `statement_list`.

```
statement_list ::= statement delimiter statement_list
                |  statement
                |
```

Where this form appears, users may write zero or more `statement`s, each separated by a `delimiter`. Aside from this (where delimiters are mandated), users may utilize newline characters to separate and beautify their code without having an effect on computation. The same is true for whitespace characters, like tabs and spaces.

8.1 Variable Definition and Assignment

```
  |  type name = expr
```

Variables can be defined, but they must have their type given at the time of definition. Variables also must be assigned a value at the time of definition. The right-hand side `expr` of assignment must evaluate to the same type as `type` on the left-hand side. If it does not, a type error will be thrown.

```
    |   name = expr
```

The value of variables can be reassigned later in code. The right-hand side `expr` of assignment must evaluate to the same type that the `name` on the left-hand side was given during definition. If it does not, a type error will be thrown. If the `name` on the left-hand side was never defined, an unbound name error will be thrown. Both variable definitions and assignments are right-associative, meaning that they are grouped from right to left and have the lowest precedence among all operators supported by sPool.

```
# Variable Definition
int    x = 1
string a = "hi"

# x evaluates to 1, a evaluates to “hi” now

# Reassignment of the value of a variable
x = 3
a = "world"

# x evaluates to 3, a evaluates to “world” now
```

8.2 If and Else

```
    |   if (expr): statement_list (else statement_list )? ;
```

`if` conditionals are built using an expression, which must evaluate to a boolean value, and a list of statements. Users have the option to include an **`else`** clause, which will only contain a series of statements, and no conditional. Note the use of the colon and semicolon. Colon opens the **`if`** statement, and semicolon ends it, *after* any associated **`else`** clause. When nested if statements are used, this clarifies the “dangling else” problem:

```
string s = ""
if (x < 3000):
    s = "LT 3000"
    if (x > 100):
        s = "GT 100"
    else
        s = "else";
;
```

In the above example, the `else` clause comes before the second `if` statement is closed, which indicates it should associate with the second `if` statement. If the value of `x` is 50, this statement assigns `s` to `“else”`.

```
string s = ""
if (x < 3000):
    s = "LT 3000"
    if (x > 100):
        s = "GT 100";
else
    s = "else"
;
```

In the second example, the `else` clause comes after the second `if` statement is closed, and before the first `if` statement is closed, which indicates it should associate with the first `if` statement. If the value of `x` is 50, this statement assigns `s` to `“LT 3000”`. Note that the indentation and horizontal spacing here are merely for readability and stylistic convention, and do not affect the result of running the code. Newlines, on the other hand, do have a semantic meaning and separate statements, as mentioned in section **2.2.1**.

8.3 Looping

```
    |  while (expr): statement_list ;
```

In sPool, code can be run repeatedly using a `while` loop. There are no for loops or foreach loops included in the syntax of sPool; however, this functionality can be implemented by the user with the `while` loop. The *`expr`* is checked at the start of each pass through the *`statement_list`*. If it evaluates to `true`, the *`statement_list`* runs. Otherwise, the *`statement_list`* is skipped. It is required that the *`expr`* evaluates to a boolean value.

The following sPool code illustrates an example of a nested while loop:

```
while (x < 10):
    while (y > 10):
        z = x * y
        y = y - 1
    ;
    x = x + 1
;
```

8.4 Function Definition and Return Statement

```
    | def (store)? type name ((type name(, type name)*)?):
statement_list ;
```

The keyword `def` can be used to begin function definition.

In function definition, there is first an optional keyword `store` which indicates whether or not this function will utilize built-in dynamic programming to store a mapping from argument values to results. When this keyword is present, the function result will be cached and stored in memory, which can be reused in subsequent function calls. This helps reduce extra computation by minimizing redundant function calls. Cached results persist between individual calls. At most, 32 results may be stored before subsequent new results evict older results to make space. The `store` feature is only available with function definitions using the keyword `def`; anonymous `lambda` functions do not have this option.

Next, the return *`type`* of the function must be given, followed by a *`name`* for the function. If a function is intended to return nothing, a *`type`* of `quack` should be given (section **4.3**). Next, any formal parameters to the function should be given within parentheses, as a bind list of *`type`* followed by *`name`*, separated by commas. If no parameters are desired, users can leave the parentheses empty.

The ultimate type of the function defined is an arrow type (section **4.5**), which will go from the type(s) set out in the formal parameters to the type given as the return type.

```
    |  return (expr)?
```

Inside the body of a function defined with `def` or `lambda` that does not return `quack`, sPool semantically requires that there be at least one `return` statement. For functions that do return `quack`, the `return` statement is optional in their body, and may be used simply to exit the function early. Although syntactically valid, return statements used outside functions are semantically invalid and will result in an error. When they are reached in a function body, they indicate to terminate execution of the function and make the function call evaluate to either the optional *`expr`* on the right-hand side of `return`, or nothing, if no *`expr`* is given. The type of *`expr`* being returned must be the same type as the return type described at the function's definition. If no *`expr`* is included, the function evaluates to nothing, the `quack` type.

Similarly to `if` statements and `while` statements, the function definition's body begins with a colon and ends with a semicolon.

See the below example of a function:

```
# callLambda is a function that takes in a function and an integer
# and returns a boolean
def bool callLambda((int -> bool) myFunction, int callWith):
    callWith = callWith + 1
    return myFunction(callWith)
;


# calling the function:
callLambda(lambda bool (int i): return false;, 5)


 # a function that takes in no arguments and returns nothing
def quack printHI():
    println("Hello world!")
    return; # empty return for function that returns nothing
```

9 Program

program ::= *statement_list*

At the highest level, a sPool program is a list of statements, which may or may not begin with some number of newline characters. There is no explicitly defined entry point for a program written in sPool, like a main function. The normal control flow of sPool begins with the very first statement written in the source file and continues sequentially.

10 Built-In Functions

sPool implements several built-in functions that act as simple building blocks using which a user can write clean, efficient sPool programs. The following subsections contain the names, types, and contracts for such built-in functions provided in sPool.

For any of these functions mentioned in sections **10** and **11**, violating the contract results in an error/exception being thrown. It is the responsibility of the developer to call the given functions with permissible values as arguments.

10.1 Built-In Functions for Printing

Name	Function Type	Function Contract
`print`	`(string -> quack)`	Takes in a string as its argument and prints out the contents of the string to the standard output without a newline at the end.
`println`	`(string -> quack)`	Takes in a string as its argument and prints out the content of the string to the standard output with an additional newline appended at the end.

sPool provides two built-in functions for printing string to standard output. They both take in string values as arguments and return nothing, outputting the string value to standard output. An example of these functions in action is given below:

```
print("hi") # prints “hi” without a newline
println("hello world") # prints “hello world” followed by a newline
```

10.2 Built-In Functions for Type Conversions

Name	Function Type	Function Contract
`int_to_string`	`(int -> string)`	Takes in an integer and returns its string representation.
`float_to_string`	`(float -> string)`	Takes in a floating point number and returns its string representation.
`bool_to_string`	`(bool -> string)`	Takes in a boolean value and returns its string representation.
`int_to_float`	`(int -> float)`	Converts an integer into a floating point number.
`float_to_int`	`(float -> int)`	Converts a floating point number into an integer by rounding down to the nearest integer.

sPool provides several functions for converting values of one type to another. These are particularly useful when performing arithmetic operations between values of different types like integers and floating point numbers.

Some examples showing these functions in use are:

```
int_to_string(157) # returns "157"
float_to_string(157.234) # returns "157.234"
bool_to_string(true) # returns "true"
int_to_float(157) # returns 157.0
float_to_int(1.234) # returns 1

println(int_to_string(157)) # prints "157" followed by a newline
```

10.3 String Built-In Functions

Name	Function Type	Function Contract
`String_len`	`(string -> int)`	Returns the length of a string as an integer.
`String_concat`	`(string, string -> string)`	Concatenates the two strings supplied as arguments, and returns the concatenated string.
`String_substr`	`(string, int, int -> string)`	Takes in a string value, a start index, and an end index, and returns the substring of the original string starting at the given start index and ending at the end index. The second argument is inclusive, and the third argument is exclusive. An error will be raised if any of the given indices are out of bounds. The start index must be in the range [0, n - 1], and the end index must be in the range [0, n], where n is the length of the supplied string. The start index must be less than the end index.

sPool provides built-in functions that operate on values having the `string` type. Basic functionalities of enquiring the length of a string, concatenating two strings together, and getting a substring of another string are provided by default in sPool:

```
String_len("hello") # returns 5
String_concat("hello", "world") # returns "helloworld"
String_substr("hello", 1, 3) # returns "el"
```

10.4 Mutex Built-In Functions

Name	Function Type	Function Contract
`Mutex`	`(quack -> mutex)`	Initializes and returns an instance of type `mutex`.
`Mutex_lock`	`(mutex -> quack)`	Locks a mutex at the line of code where invoked. Subsequent code may only be run by at most one thread at any given time, until `Mutex_unlock` is invoked.
`Mutex_unlock`	`(mutex -> quack)`	Unlocks a mutex at the line of code where invoked. Subsequent code may be run concurrently by any number of threads at any given time, until `Mutex_lock` is invoked.

10.5 Thread Built-In Functions

Name	Function Type	Function Contract
`Thread`	`(string, string -> thread)`	Returns a thread which will execute code concurrently. The code it will execute is determined by the first string passed, which represents the name of a function. The second string passed represents the arguments to be passed when calling that function, separated by commas. Note that any issues with the function not being defined or improper arguments being passed will *not* raise errors when `Thread` is called; these errors will arise when the function passed as the first argument is called on the new thread asynchronously.
`Thread_join`	`(thread -> quack)`	Waits for the thread supplied as the argument to complete its execution. Control flow of the thread that calls this function is suspended until the thread supplied as the argument is done executing.

Using threads is not syntactically trivial in sPool. The following example demonstrates the use of threads in sPool:

```
mutex myLock = Mutex() # use the Mutex() function to create a mutex value

def store quack myfunc(int a, int b, int c):
    Mutex_lock(myLock) # acquire the mutex
    println(String_concat("Inside thread: ", int_to_string(a + b + c / 2)))
    Mutex_unlock(myLock); # release the mutex


# For t1:
# myfunc will be called with integers 1, 3, and 5 as its arguments in
# a separate thread, asynchronously with respect to the calling
# thread
thread t1 = Thread("myfunc", "1, 3, 5")

# For t2:
# myfunc will be called with integers -1, 0, and 2 as its arguments in
# a separate thread, asynchronously with respect to the calling
# thread
thread t2 = Thread("myfunc", "-1, 0, 2")


Thread_join(t1) # wait for t1 to be done executing
Thread_join(t2) # wait for t2 to be done executing

println("Main thread: done.")
```

As mentioned in the function contract, the region of code surrounded by calls to `Mutex_lock` and `Mutex_unlock` restrict the number of concurrently executing threads in that region to at most one. In a concurrent environment where there is a possibility of modifying shared mutable variables by multiple concurrently executing threads, the region of code that may introduce such modifications should be surrounded by the calls to `Mutex_lock` and `Mutex_unlock` functions to prevent race conditions. A mutex is the simplest level of synchronization primitive provided by sPool to solve this problem, which can be built upon by users to make their functions and other regions of code thread-safe and robust against such unexpected problems which may end up introducing non-determinism in their programs.

10.6 List Built-In Functions

In the following table, we use `list<α>` to represent the type of a list of αs. For instance, `list<int>` is the type of a list of integers and `list<bool>` is the type of a list of booleans. Therefore, for the sake of brevity, the type variables such as α, β, γ are used in the table to denote any valid types in sPool (see section **4** for more details on the types in sPool). Furthermore, if any of the function types contain the same type variable, they need to refer to the same sPool type.

Name	Function Type	Function Contract
`List`	`(int, α -> list<α>)`	Takes in an integer n representing the size of the list to be initialized and a value of type α, and initializes and returns a list of size n with the supplied value filled in.
`List_at`	`(list<α>, int -> α)`	Takes in a list of α and an index i, and returns the element (of type α) present at the given index in the list. The second argument should be in the range [0, n - 1], where n is the length of the supplied list.
`List_replace`	`(list<α>, int, α -> quack)`	Takes in a list of α, an integer index i, and a value of type α and replaces the value present at index i of the supplied list with the new value given as the third argument. The second argument of this function must be in the range [0, n - 1], where n is the length of the supplied list.
`List_insert`	`(list<α>, int, α -> quack)`	Takes in a list of α, an integer index i, and a value of type α and inserts the supplied value at index i of the supplied list. All elements in the list with their original index >= i get pushed one index forward. The second argument of this function must be in the range [0, n], where n is the length of the supplied list.
`List_remove`	`(list<α>, int -> quack)`	Takes in a list of α and an integer index i, and modifies the list with the element at that index removed. All elements at index >i are now at one index lower, so that there is no "hole" in the list. The second argument of this function must be in the range [0, n - 1], where n is the length of the supplied list.
`List_len`	`(list<α> -> int)`	Takes in a list of α and returns its length as an integer.

The `List_replace`, `List_insert`, and `List_remove` functions do not return any value. Instead, since lists are passed by reference in sPool, direct modification of the lists passed as arguments are done by these aforementioned functions. All other functions mentioned in the table do not directly modify the input list.

Some examples of using list functions in sPool are shown below:

```
list<int> l = [2, 3, 5, 7, 11]
List(3, "a")  # returns ["a", "a", "a"]
List_at(l, 3)  # returns 7
List_replace(l, 3, 13)  # l is now [2, 3, 5, 13, 11]
List_insert(l, 3, 0)    # l is now [2, 3, 5, 0, 13, 11]
List_remove(l, 3)       # l is now [2, 3, 5, 13, 11]
List_len([["a"], ["b", ""], [""], ["c"], ["d", "e"]]) # returns 5
```

11 sPool Standard Library

The standard library in sPool is automatically imported to every sPool program.

11.1 Standard Library List Functions for Integer Lists

sPool comes with standard list functions for operating on integer lists. The functions along with their signature and contracts are depicted in the following table:

Name	Function Type	Function Contract
`List_int_rev`	`(list<int> -> list<int>)`	Takes in a list of integers and returns a list of integers which contains exactly the same elements present in the original list but in reversed order.
`List_int_map`	`(list<int>, (int -> int) -> list<int>)`	Takes in a list of integers and a function that takes in an integer and returns an integer, and returns a list of integers by applying the function supplied by the second argument to every element of the list supplied by the first argument.
`List_int_pmap`	`(list<int>, (int -> int) -> list<int>)`	Takes in a list of integers and a function that takes in an integer and returns an integer, and returns a list of integers by applying the function supplied by the second argument to every element of the list supplied by the first argument. This function concurrently applies the provided function to each element of the list to get the final result.

`List_int_filter`	`(list<int>, (int-> bool) -> list<int>)`	Takes in a list of integers and a predicate on integers, and returns a list of integers from the original list that satisfies the predicate.
`List_int_fold`	`((int, int -> int), int, list<int> -> int)`	Folds a list from left to right, given a function of type (int, int -> int), an initial accumulator of type integer and a list of integers. It returns the final accumulated value of type integer as a result of the fold over the supplied list.
`List_int_append`	`(list<int>, list<int> -> list<int>)`	Takes in two lists of integers, and appends the second list to the first list and returns the resulting list.

None of these functions modify the original input list. The implementation for `List_int_pmap`, `List_int_filter`, and `List_int_fold` in sPool are provided in **Appendix I**. Some examples of using the integer list functions from the sPool standard library are:

```
List_int_rev([2, 3, 5, 7, 11]) # returns [11, 7, 5, 3, 2]
List_int_map([2, 3, 5, 7, 11], lambda int (int x): return x + 1;) #
returns [3, 4, 6, 8, 12]
List_int_pmap([2, 3, 5, 7, 11], lambda int (int x): return x + 1;) #
returns [3, 4, 6, 8, 12]
List_int_filter([2, 3, 5, 7, 11], lambda bool (int x): return 0 == (x
% 2);) # returns [2]
List_int_fold(lambda int (int acc, int x): return acc + x;, 0, [2, 3,
5, 7, 11]) # returns 28
List_int_append([2, 3, 5, 7, 11], [13, 17, 19]) # returns [2, 3, 5, 7,
11, 13, 17, 19]
```

11.2 Standard Library String Functions

Name	Function Type	Function Contract
`String_rev`	`(string -> string)`	Reverses a string and returns the reversed string.
`String_find`	`(string, string -> int)`	Takes in a string value as the first argument and another substring as the second argument and searches for the second substring in the first string. If the substring is found in the first string, the function returns the index of the substring within the first string. Otherwise, it returns -1. If the substring occurs more than once in the first string, the index of the first occurrence is returned.

The implementation for `String_rev` in sPool is provided in **Appendix I**. Some examples of using string manipulating functions from the sPool standard library are shown below:

```
String_rev("rukna") # returns "ankur"
String_find("COMP107 is fun", "fun") # returns 11
String_find("COMP107 is fun", "funny") # returns -1
```

Appendix I
The following are implementations of some standard library functions in sPool:

```
# List_int_pmap implementation
def list<int> List_int_pmap(list<int> l, (int -> int) f):
   int i = 0
   int j = 0
   int len = List_len(l)

   # create a list of threads and the resulting int list
   list<thread> threads = []
   list<int> result = List(len, -1)

   # helper function to carry function computation
   (int -> quack) pmap_helper = lambda quack (int index):
       List_replace(result, index, f(List_at(l, index)));

   while (i < len):
    # designate each thread to compute a value asynchronously
    List_insert(threads, i, Thread("pmap_helper", int_to_string(i)))
    i = i + 1;

   # wait for the threads to be done
   while (j < List_len(threads)):
       Thread_join(List_at(threads, j))
       j = j + 1;
   return result;
```

```
# List_int_fold implementation
def int List_int_fold((int, int -> int) accFunc, int acc, list<int>
iList):
    int i = 0
    while (i < List_len(iList)):
        acc = accFunc(acc, List_at(iList, index))
        i = i + 1;
    return acc;
```

```
# String_rev implementation
def string String_rev(string word):
    int    len = String_len(word)
    string rev = ""

    while (length > 0):
       rev = String_concat(rev, String_substr(word, len - 1, len))
       len = len - 1
    ;

    return rev;
```

```
# List_int_filter implementation
def list<int> List_int_filter(list<int> l, (int -> bool) p):
   int i = 0
   int j = 0
   int elem = -1
   int len = List_len(l)
   list<int> result = []

   while (i < len):
       elem = List_at(l, i)
       if (p(elem)):
           List_insert(result, j, elem)
           j = j + 1;
       i = i + 1;
   return result;
```