The sPool Programming Language

```
let get_details name = match name with
 | “Ankur Dahal”    -> “ankur.dahal@tufts.edu”
 | “Max Mitchell”   -> “maxwell.mitchell@tufts.edu”
 | “Etha Hua”       -> “tianze.hua@tufts.edu”
 | “Yuma Takahashi” -> “yuma.takahashi@tufts.edu”
 | “Richard Townsend” | _ -> SPAM
```

Introduction

sPool is a statically-typed general-purpose programming language. sPool incorporates multiple features including polymorphism using templates, concurrent programming via multithreading, automatic memoization/caching for dynamic programming, functions as first-class citizens, and structures allowing basic object-oriented capabilities.

Language Features:

1. ***Structures and Duck Typing***

sPool incorporates simple structures which allow basic OOP programming and user-defined types. They behave similarly to C++ structs and can be polymorphic, like functions. In each `struct`, there are two required components: a `member` field, where the instance members and instance methods are defined, and an `init` field, where the constructor is defined. In the `member` field, users can extend primitive functions, such as addition, allowing their types to `duck` for other types in certain cases. This is similar to operator overloading in C++. We include this to conveniently allow for interoperability between different types.

```
# this is a comment
# say we want to average a CS student’s grades but some of our grades are stored
# as integers, others as `CS_Grade` type
struct CS_Grade(string s): # colon opens scope à la `{` in C++
     member:
          string text = s
          int grade = 0

          # duck describes how this type behaves when used with the `+` operator
          duck int + (x):
               return grade + x;; # semicolon closes scope à la `}` in C++
          # two semicolons used here – one to close duck scope,
          # another to close member scope
     init:
          if s == “Excellent”: grade = 100;
```

```
        else if s == "Good": grade = 80;
        else if s == "Fair": grade = 60;
        else:                grade = 0;;;

# example raw student data, and how we could calculate the average
int avg = (((67 + CS_Grade("Good")) + CS_Grade("Excellent")) + 83) / 4
```

2. ***Templates and Functions***

Functions are first-class citizens. They can be assigned to variables, sent as parameters to functions, and returned from functions. They can also be anonymous and polymorphic. To define a polymorphic function, the user writes (uninstantiated) type variables in places of specific types. Before calling a polymorphic function, the user first instantiates the function with a specific type within the angle brackets (e.g. `len<string>`) and then calls the function.

```
# functions are defined using the def keyword – return type of function is
# specified before the function identifier, and argument list contains
# comma-separated list of arguments along with their types
def int double(int i):
     return i * 2;

# polymorphic functions! (we also have polymorphism for structs)
# sPool is statically-typed, so types have to be explicitly written. In this
# case, a polymorphic anonymous function parameterized on the type variable T
# is assigned to the identifier `len`, whose type is (T -> int).
# The arrow -> separates the argument types from the return type.
func<T> (T -> int) len = lambda (T iter):
    # this function takes advantage of the fact that any
    # primitive type can be indexed with []
    int i = 0
    while iter[i] != quack:  # quack is our version of null
        i = i + 1;           # this semicolon closes the while loop's scope
    return i;                # last semicolon closes the function's scope

len<list<int>>([1, 2, 3])     # evaluates to 3
len<string>("how long am i?") # evaluates to 14
len<int>(271828)              # evaluates to 6
len<bool>(true)  # evaluates to 1; true[0] = true, elsewhere undefined
len<bool>(false) # evaluates to 0; false is undefined everywhere
```

3. ***Concurrency***

sPool implements concurrency using multithreading and the shared memory model, where multiple threads can concurrently read from and write to shared memory locations. While this takes away some of the safety features that other models (like the actor model) provide, it provides the users with more flexibility and control.

```
# parallel_map is a templated function parameterized over two type
# variables, `A` and `B`. The return type list<B> is also polymorphic.
def<A, B> list<B> parallel_map(func (A -> B) f, list<A> l):
     int i = 0
     int j = 0
     # threads is a list of thread objects
     list<thread> threads = []
     list<B> result = list(len(l))
     func (int -> quack) pmap_helper = lambda (int i):
          result[i] = f(l[i]);

     while i < len(l):
          # items can be added to lists using the add() method.
          # thread(pmap_helper(i)) spawns and starts a new thread, where the
          # function `pmap_helper` is executed with the argument `i`. This is
          # asynchronous from the perspective of the main thread
          threads.add(thread(pmap_helper(i)))
           i = i + 1;

     # synchronization: threads have the wait() method which blocks the main
     # thread until the thread is done executing
     while j < len(threads):
          threads[j].wait()
          j = j + 1;

     return result;
```

`lock` is used as a synchronization primitive to implement mutual exclusion (i.e., to prevent multiple threads from running a region of code concurrently). In some cases, we may need to prevent shared variables from being modified by multiple threads concurrently – for this, we introduce the `lock` type.

```
# Declare a lock named mutex
lock mutex = lock()

def quack modifyGlobalVar():
      mutex.lock()
      # This is the critical region where shared memory is modified.
      # For example, modifying a global counter would go here.
      mutex.unlock();

# create two threads targeting modifyGlobalVar function and
# start execution in the background
thread t1 = thread(modifyGlobalVar())
thread t2 = thread(modifyGlobalVar())

# wait for the threads to finish executing
t1.wait()
t2.wait()
```

When a thread, (either **t1** or **t2**) calls the `lock()` function on the `lock` instance **mutex**, it holds **mutex** and prevents any other thread from entering the critical region until the holding thread calls `unlock()` on **mutex**. Therefore, it is now safe for that thread to modify any shared mutable variables between these two function calls, because at any given time, at most one thread will be running in this region. This is how one can use locks to modify shared mutable state in sPool.

4. ***Dynamic Programming***

Instead of the user having to cache computed results manually, sPool automates memoization by using the keyword `store`. The `store` keyword indicates that any results generated by the function should be cached. Furthermore, it indicates that any subsequent call to this function (including recursive calls) should first check if those passed parameters map to a value in the store. If so, the function will not be called, and instead the value stored will be retrieved. `store` can be parameterized using an integer, indicating the maximum number of values that should be stored before eviction. If no value is given, at most 16 values will be stored by default. Maps generated by `store` persist after individual function calls, so this feature can be useful even with non-recursive functions.

```
def int store<> fib(int i):
      if i == 0 or i == 1:
            return i;
      # under the hood, compiler checks the store/cache before
      # computing fresh results
      return fib(i - 2) + fib(i - 1);
```