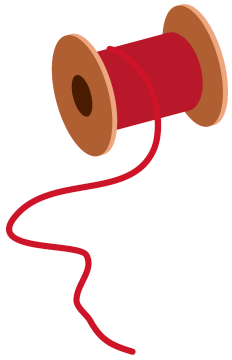
The sPool Programming Language

A General-Purpose Programming Language with Concurrency Support



Final Report

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

This document introduces sPool, a general-purpose, statically-typed programming language. It supports and incorporates multiple language features, including concurrent programming via multithreading, homogeneous first-class lists, automatic memoization for dynamic programming, and functions as first-class citizens.

Being a multithreaded language, sPool allows users to write efficient, concurrent, high-performance algorithms and to utilize the automatic memoization feature to enhance the performance of certain algorithms. In sPool, functions can be assigned to variables, sent in to other functions as parameters, and returned from functions. Functions also capture the environment where they were defined in. With the combination of the aforementioned general features, users who want to create highly effective general-purpose algorithms that take advantage of concurrency should consider using sPool.

2 Language Tutorial

In this section, a simple language tutorial will be given to users who are interested in programming with sPool for the first time. We provide a short tutorial on compiling and running a sPool program, writing a hello-world program, defining variables with different types, using lambda expressions and defining higher order functions, utilizing the store feature as well as writing some basic concurrent programs.

2.1 How to compile and run a sPool program

To compile a sPool source program, one needs to compile the sPool compiler first. Make sure you are in the src directory and type `make` to compile the compiler. If you have the OCaml environments and all the required tools installed already, you will see the sPool compiler being built. After the compiler is compiled, you can use the following command to compile a sPool program that you wrote:

```
./compile.sh [-stdlib] <sP file> <executable file>
```

Where the “sP file” is the name of the sP file you want to compile, and the output file is the name of the executable you want it to be. Notice the optional flag -stdlib: if it is included, the standard library functions will be available for you to use in your program automatically. For more information on what functions are available in the sPool standard library, see section **3.11**.

When you have your executable ready, run it as you run any other executable. If you named your executable my_executable while compiling your sPool program, type the following to run it:

```
./my_executable
```

2.2 Hello world!

To write a hello-world program in sPool, you only need to utilize string literals (see section **3.5**) and the built-in `println` function (see section **3.10.1**). Just type:

```
# this is a single-line comment
println("Hello world!")
```

The # character can be used to create single-line comments. If you compile this program and run it as specified in the previous section, you will see:

```
$user : ./compile.sh helloworld.sP hello
$user : ./hello
Hello world!
```

2.3 Types and Definitions

The primitive types in sPool are `int`, `float`, `bool`, and `string`. Two concurrency-related types are `thread` and `mutex`. Because sPool is a statically typed language that supports functions to be stored in variables, function types are supported as well in the form of arrow types. Lists can be created to store elements of the same type. Here are some examples for defining and assigning variables of each type:

```
int a = 2
a = 2993 - 12

float b = 3.1415
bool c = false
string d = "hello sPool"

thread t1 = { 1 + 2 }
mutex lock = Mutex()

def quack printNum(int x):
    println(int_to_string(x))
    return;

list<int> l1 = [1, 3, 5, 7, 9]
list<list<string>> l2 = [["a", "d", "g"], ["b", "e", "h"], ["c", "f", "i"]]
```

In definitions of basic variables, the type of the variable is first stated, followed by the name of the variable, an equal sign and then an expression (see section **3.7** for more on expressions) to be assigned as the value of the variable.

In function definitions, the type after `def` is the return type, and the types in the parentheses are the types of the parameters that the function takes. Notice the `quack` type is used in this case because the function does not return any value – it is analogous to the `void` type in languages like C and C++.

Function types are constructed using arrows connecting the input and output types. List types are constructed using angle brackets, such as `list<aType>` where `aType` is the type of the list's elements.

2.4 Lambdas and Higher-Order Functions

sPool is a functional programming language! Let's write some lambda expressions and higher order functions. Let's start by writing the function `isEven` using a lambda expression:

```
(int -> bool) isEven = lambda bool (int x):
    return (x % 2) == 0;

println(bool_to_string(isEven(15)))
println(bool_to_string(isEven(4)))
```

The `%` is used for modulo and the `==` is used for equality comparison (see section **3.6.2** for more on binary operators). Compile and run it:

```
$user : ./compile.sh -stdlib tutorial.sP tutorial
$user : ./tutorial
false
true
```

That's correct! Let's move on to write a higher-order function called `flipBool`, which takes in a function of type `(int -> bool)` and returns a function of the same type – but as the name implies, the returned function behaves exactly the opposite way as the input function:

```
def (int -> bool) flipBool((int -> bool) aFunction):

    (int -> bool) flippedFunc = lambda bool (int x):
        return !aFunction(x);

    return flippedFunc;
```

The `!` is used for boolean negation (see section **3.6.1** for more on unary operators). Let's try calling `flipBool` with `isEven`. We expect it to return a function which, when called, returns `false` when the input is an even number, and `true` when the input is an odd number:

```
(int -> bool) isOdd = flipBool(isEven)

println(bool_to_string(isOdd(15)))
println(bool_to_string(isOdd(4)))
```

Compile and run it:

```
$user : ./compile.sh -stdlib tutorial.sP tutorial
$user : ./tutorial
true
false
```

Nice, just as expected!

2.5 Automatic Memoization

Another core feature of sPool is the `store` keyword in function definition, which enables automatic memoization (see section **3.8.4** for more on `store` and function definition). It can be added as in the definition of any `(int->int)` function and the defined function will automatically cache its result for the supplied argument. Any computed argument values will just be retrieved from the cache, rather than running the function body again. Let's implement a Fibonacci calculator using `store`:

```
def store int fibonacci(int n):
    int result = 0
    if (n <= 1):
        result = n
    else
        result = fibonacci(n - 1) + fibonacci(n - 2);
    return result;

def int fib_nostore(int n):
    int result = 0
    if (n <= 1):
        result = n
    else
        result = fib_nostore(n - 1) + fib_nostore(n - 2);
    return result;
```

The difference between `fibonacci` and `fib_nostore` is the `store` keyword. Let's time the two functions on the 45th fibonacci number:

```
println(int_to_string(fib_nostore(45)))
```

```
$user : ./compile.sh -stdlib tutorial.sP without_store
$user : time ./without_store
1134903170

real    0m4.917s
user    0m4.910s
sys     0m0.000s
```

```
println(int_to_string(fibonacci(45)))
```

```
$user : ./compile.sh -stdlib tutorial.sP with_store
$user : time ./with_store
1134903170

real    0m0.009s
```

```
user    0m0.004s
sys     0m0.000s
```

That is over 1000x faster between the two processes!

2.6 Concurrency

Concurrency in sPool is facilitated with threads and mutexes. The first program is a simple illustration of defining and invoking new threads:

```
def quack sayHi():
   println("Howdy folks!")
   return;

thread t1 = { sayHi() }
thread t2 = { sayHi() }
```

The curly braces are used to create a thread. The statements enclosed by the curly braces are dispatched asynchronously to a new thread and executed. By default, a thread is invoked when it is defined. Let's compile and run the program:

```
$user : ./compile.sh -stdlib tutorial.sP tutorial
$user : ./tutorial
$user :
```

Well, there isn't any output seen, but we were expecting two "Howdy folks!" to be shown on the terminal. This is because the main thread ends before `t1` or `t2` ends. To wait for `t1` and `t2` end before we end the main thread, we use `Thread_join`:

```
def quack sayHi():
   println("Howdy folks!")
   return;

thread t1 = { sayHi() }
thread t2 = { sayHi() }

Thread_join(t1)
Thread_join(t2)
```

And let's compile and run:

```
$user : ./compile.sh -stdlib tutorial.sP tutorial
$user : ./tutorial
Howdy folks!
Howdy folks!
```

Now it works. The next example shows how to use mutexes to prevent race conditions. Mutexes, short for "mutual exclusion," are synchronization primitives used in concurrent programming. They protect shared resources from being accessed by multiple threads simultaneously, which could lead to data inconsistency, race conditions, or other undesirable behaviors which are very hard to debug. Mutexes help ensure that at most only one thread can access a shared resource or critical section of code at a time, providing a mechanism for managing concurrent access and maintaining data integrity.

We first initialize a list to be modified, a shared integer recording the number of invocations being made, as well as a mutex variable `lock`. The `shared` keyword used when declaring this integer just means it can be modified inside functions and threads, and those modifications will persist outside of the functions/threads (see section **3.8.1** for more on shared variables). Since we used the shared keyword, this integer will be captured by reference via the closure of the thread, and modifications made within the thread's code will be preserved after the thread has finished executing.

```
list<int> mySharedList = [1, 2, -1, -2, 0, 5]
shared int num_invocations = 0
mutex lock = Mutex()
```

Then, we define an `addToList` function, which sets `index`, the first argument, to be `value`, the second argument, in the list `mySharedList`. Notice that we use `Mutex_lock(lock)` to lock the section of code where we modify global variables `mySharedList` and `num_invocations`. This prevents other threads from accessing and modifying the same global variables concurrently. After we have modified `num_invocations` and replaced an element in `mySharedList`, we unlock the mutex `lock` so that other threads may access and modify these variables.

```
def quack addToList(int index, int value):
   # modifying global variables requires a lock to prevent race conditions
   Mutex_lock(lock)
   num_invocations = num_invocations + 1 # one more thread invoked this function
   List_replace(mySharedList, index, value)
   Mutex_unlock(lock)
   return;
```

Next, we invoke two threads and wait for their completion in the main thread. Both threads call `addToList`, which modifies global variables. This means use of `lock` is necessary within `addToList` in order to prevent both of the threads from modifying `mySharedList` or `num_invocations` at the same time.

```
thread t1 = { addToList(0, 99) }
thread t2 = { addToList(0, 88) }

Thread_join(t1)
```

```
Thread_join(t2)
```

Finally, we print out the modified list as well as the number of threads invoked. We expect the head of the list to be either 99 or 88 (because the threads operate concurrently, this program is non-deterministic), but we don't expect there to be any memory accessing errors since we have used lock to prevent that from happening.

```
println(int_to_string(List_at(mySharedList, 0))) # 99 or 88
print("# of threads that called addToList: ")
println(int_to_string(num_invocations))
```

Let's compile and run:

```
$user : ./compile.sh -stdlib tutorial.sP tutorial
$user : ./tutorial
99
# of threads that called addToList: 2
```

Our program printed that the number of threads which called `addToList` is 2, which is indeed the case.

That concludes our tutorial. You now have everything you need to get started in sPool. Please refer to the Language Reference Manual (section **3**) for answers to further questions.

3 Language Manual

3.1 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 section (section **3**) 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 **3.2.1**). It then goes on to explain the lexical conventions of the language (section **3.2.2**) and the scoping rules in sPool (section **3.3**), followed by the essential language features like the types used in sPool (section **3.4**), literals in sPool (section **3.5**), permitted arithmetic and logical operations (section **3.6**), and expressions and statements (sections **3.7** and **3.8**) that help define a complete sPool program (section **3.9**). Finally, this manual introduces the built-in functions that are provided to users when they write a sPool program (section **3.10**), followed by the functions included in the sPool standard library (section **3.11**).

3.2 Conventions

3.2.1 Manual Conventions

Throughout this manual, text written in black, Arial font should be taken as prose. 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 **3.2.2.2**. 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.

3.2.2 Lexical Conventions

sPool is not a free-form language, meaning that certain delimiters have special semantic meanings. Specifically, these special delimiters are discussed in section **3.2.2.1** below with their semantic significance.

3.2.2.1 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...
```

Most whitespace 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 **3.8**.

3.2.2.2 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	
shared			bool	list	

There are also certain names used in our built-in functions, listed in section **3.10**, 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 alphabetical characters 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_]*`

3.3 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 **3.7** and **3.8** 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. There is an exception to this rule: if *statement*s, which only have a single set of colon and semicolon, may introduce two scopes, if there is an else branch associated with the if statement. Each branch gets its own scope.

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).

New threads and anonymous functions created capture the scope when defined. All variables in-scope at definition-time can be considered as in-scope for the thread's (and the lambda's) lifetime, except they are only *copies* of the values at their definition time. For more information about threads, see section **3.4.4**. For more information about anonymous functions, see section **3.7.3**.

There are a few exceptions to this: Lists and mutexes are heap-allocated and mutable by multiple separate threads. Additionally, variables marked as **`shared`** at definition-time are also

heap-allocated, shared between all threads and can be modified by any thread or function. See section **3.8.1** to learn more about variable definition and sharing.

For the sake of example, we will use if statements, variable definition, and variable evaluation. Please see section **3.8** to learn more about *statement*s, and section **3.7** 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
```

Defining the same name multiple times in the same scope will cause an error.

3.4 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>
```

3.4.1 Integers, Floats, and Booleans

`int`, `bool`, and `float` are primitive types in sPool. A value having the `int` type is a 32-bit integer value in sPool. Similarly, a value having the `float` type is a IEEE 64-bit floating point number in sPool. Values having the type `bool` represent Boolean values in sPool.

3.4.2 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 **3.10.3**. To learn more about the standard library functions in sPool associated with strings, see section **3.11.2**.

3.4.3 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.

3.4.4 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 shared variables at any given time. Threads are invoked immediately upon definition.

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 **3.10.4** and **3.10.5** respectively.

3.4.5 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 **3.7.2**, **3.7.3**, and **3.8.4**, 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.

3.4.6 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 can be created of any type, however arrow type lists are undefined. 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 **3.10.6**. To learn more about the standard library functions in sPool associated with lists, see section **3.11.1**.

3.5 Literals

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

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

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 **3.7**. Lists, and all of their elements, are stored on the heap. When passed to functions, they are passed by reference. 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<alpha> 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>>
```

Thread literals are created with an opening and closing curly brace, containing a *statement_list*, which is zero or more newline-delimited statements. This is known as the body of the thread. For more on *statement* and *statement_list*s, see section **3.8**. The *statement_list* between the curly braces is the code which will be run concurrently on a

newly spawned thread. Threads are invoked on creation, so creating a thread literal immediately spawns a thread running the *statement_list* written as the body of the thread literal. For more examples of using threads in sPool, please see the code listing provided in section **3.10.5**.

3.6 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.

3.6.1 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.

3.6.2 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 respectively. Integer division is handled with floor, ala C. The % arithmetic operator, which performs modulo, must have operands of the type `int` only, and evaluates to a value of the `int`. Signed remainder is used for modulo calculation.

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. The **only** permitted types for the operands are `int`, `bool`, and `float`. The comparison operators always evaluate to a boolean value. If two `string` values are to be compared, use the `String_eq` built-in function instead (see section **3.10.3**).

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 **3.6.1**, all unary operations have higher precedence than binary operations in sPool.

3.7 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 **3.7.2** for more information about function calls.

For more information on expressions containing *literal*, *unop*, and *binop*, see sections **3.5, 3.6.1,** and **3.6.2** respectively. In these sections, references to “operand” are the value received after evaluating the expression *expr*.

3.7.1 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 **3.8.1** for more on variable definition and assignment.

3.7.2 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 **3.8.4**. All arguments except lists and shared variables are passed by value in sPool; lists and shared variables are passed by reference. For more information on shared variables, see section **3.8.1**.

In the function call, the *expr*s passed must be of the same type as the parameters expected by the function's definition. There must be the same number of *expr*s passed during the call as the parameters expected by the function's definition. 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. For more information on scope, see sections **3.3**. For more information on built-in and library functions, see sections **3.10** and **3.11**, 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
```

3.7.3 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 **3.8**. 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)
```

In this example, notice that the return statement is on the same line as the lambda definition. This is not necessary, and the return statement can be written on a new line too and can have

any number of newlines immediately after the colon and before the actual statement list of the lambda body.

Functions capture their environment at definition time. A copy is made of every variable in scope at definition time. All these values are in scope for the entirety of the function. Modifying these copies does not change the values outside of the scope, except when it comes to the heap-allocated shared variables and list variables. To learn more about scope and shared variables, see sections **3.3** and **3.8.1** respectively.

3.7.4 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
```

3.8 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 ::= (shared)? 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 **3.4**.

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. For example:

```
# valid statement_list:
int x = 1 # you do not need a newline before starting a statement_list!
# another valid statement_list:


int y = 1


int z = 1

# another valid statement_list:
int a = 1
int b = 1
int c = 1

int d = 1

# just newlines, but valid statement_list:




# empty, but valid statement_list:
```

3.8.1 Variable Definition and Assignment

```
    | (shared)? 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. Redefinition of a given within the same scope is prohibited. Variables of quack type or lists of quack type may never be defined explicitly, and those types must never appear in variable definition on the left-hand side.

The optional **shared** keyword indicates whether this variable is mutable across closures captured by functions and threads. Since shared variables are declared on the heap, they can be mutated by concurrently executing threads, and also by functions that capture these

variables. By default, list and mutex values are shared even when the **shared** keyword is absent in their definition. There is an exception, however – all **shared** variables (except lists and mutexes) lose their **shared** property when passed as arguments to functions; they are downgraded to local, non-shared variables when this happens.

```
# Shared Variable Definition
shared int x = 1
       int y = 1
shared int z = 1


thread              t = { x = x + 1
                          y = y + 1
                          z = z + 1 }
(int -> quack) func = lambda quack (int local):
                           x = x + 2
                           y = y + 2
                           local = local + 2
                           return;
Thread_join(t) # this is a blocking call; wait for the thread to be done

func(z) # when z is passed, it gets downgraded to a local variable
        # within the function since it is passed as an argument
x # shared   int, evaluates to 4
y # unshared int, evaluates to 1 (notice that it did not change!)
z # shared   int, evaluates to 2 (was only modified once by the thread)
```

| *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 the *name* on the left-hand side was never defined, an “unidentified flying name X” error will be thrown. Reassignment to values having the `thread` type is undefined behavior in sPool. For more information on `threads`, see section **3.4.4** and end of section **3.5**. Assignment of one list type variable to another does *not* create a new list, but rather has both variables pointing to the same list literal on the heap.

```
# Variable Definition
int    x = 1
string a = "hi"
list<int> l1 = [0, 1, 2]
list<int> l2 = l1        # not a copy, just another reference to l1

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

```
# Reassignment of the value of a variable
x = 3
a = "world"
List_remove(l2, 0) # remove 0 from the list
# x evaluates to 3, a evaluates to “world” now
# l1 and l2 both evaluate to [1, 2] now
```

3.8.2 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. 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 **3.2.2.1**.

3.8.3 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:

```
int x = 0
int y = 20
int z = 0
while (x < 10):
    while (y > 10):
        z = x * y
        y = y - 1
    ;
    x = x + 1;
```

3.8.4 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. Functions capture their environment at definition time by copying the current scope. To learn more about environment capture and closure, see section **3.7.3**.

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. The **`store`** functionality only works for functions which are of `(int -> int)` type, meaning they take an integer argument, and return an integer result. Otherwise this functionality will do nothing. This helps reduce extra

computation by minimizing redundant function calls. Each function declared with **store** will get its own unique cache. Cached results persist between individual function calls, and the store of one function will not interfere with the store of another function. At most, 32 results may be stored for each function before subsequent new results evict older results to make space. For `(int -> int)` functions that utilize the store functionality, returning the minimum integer value for 32 bit signed integers will cause the result to not be cached, as this number is used under-the-hood for low-level cache query purposes. The **store** feature is only available with function definitions using the keyword **def**; anonymous **lambda** functions do not have this option. Functions declared with **store** lose their **store** property when passed as arguments; they are downgraded to regular functions without caching when used in that context.

```
# fibonacci that utilizes automatic memoization using the 'store' keyword
def store int fibonacci(int n):
    int result = 0
    if (n <= 1):
        result = n
    else
        result = fibonacci(n - 1) + fibonacci(n - 2);
    return result
;
# fibonacci that does not utilize automatic memoization
def int fib_nostore(int n):
    int result = 0
    if (n <= 1):
        result = n
    else
        result = fib_nostore(n - 1) + fib_nostore(n - 2);
    return result
;

fibonacci(45)   # terminates in ~0.15s, for results in recursive calls
                # are obtained directly from the cache
fib_nostore(45) # terminates in ~13s, for every recursive call has to
                # perform the entire computation
```

store functions can also be utilized to implement idempotent functions with side-effects. However, since **store** functions lose their **store** property when passed as arguments, this idempotence will be lost if the **store** functions are passed as higher-order functions:

```
def store int print_once(int i):
```

```
    println("Hello world!")
    return i
;

# a higher order function that takes in an (int -> int) function and calls it
def int print_once_hof((int -> int) my_print_func):
    print("HOF Printing: ")
    my_print_func(0)
    return 0
;

int    x = 0
while (x < 10):
    print_once(1)              # will only print once, for results for argument
                               # 1 are cached in store. No actual function computation
                               # needs to be done for successive calls
    print_once_hof(print_once) # will print every time the loop runs because
                               # passing the print_once function to print_once_hof
                               # will downgrade it to a non-store function in the
                               # function body
    x = x + 1;
```

Next, after the optional **store** keyword, 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 **3.4.3**). 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 **3.4.5**), 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**, sPool semantically requires that there be exactly one **return** statement, in the toplevel of the function, rather than in a branch of an if, else, or while. Additionally, no statements may follow the return statement, as they will never be reached. 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
```

3.9 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.

3.10 Built-In Functions

sPool implements several built-in functions. The following subsections contain the names, types, and contracts for such functions.

For any of these functions mentioned in sections **3.10** and **3.11**, 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.

3.10.1 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.

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
```

3.10.2 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.

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
```

3.10.3 String Built-In Functions

Name	Function Type	Function Contract
`String_len`	`(string -> int)`	Returns the length of a string as an integer.
`String_eq`	`(string, string -> bool)`	Compares the two strings supplied as arguments, and returns true if they are the same string otherwise returns false.
`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.

These string built-in functions can be used like this in sPool:

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

3.10.4 Mutex Built-In Functions

Name	Function Type	Function Contract
`Mutex`	`(quack -> mutex)`	Initializes and returns a value 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 (in terms of the "text" of the program source code) may be run concurrently by any number of threads at any given time, until `Mutex_lock` is invoked.

3.10.5 Thread Built-In Functions

Name	Function Type	Function Contract
`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

# A simple function that prohibits concurrent threads from executing its
# body by using a Mutex; the function body restricts the number of
# concurrent threads running it to at most one at all times, thanks to
# the Mutex_lock and Mutex_unlock calls
def 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
    return;
```

```
# Now we will create two threads, t1 and t2. In each thread, we will
# call myfunc with certain arguments. The observable output should
# be nondeterministic in terms of which thread acquires the lock
# in the body of myfunc first, but the main thing to note here is
# that at no point will both threads concurrently execute the
# function body of myfunc due to the critical region being locked
# with the mutex myLock. The two possible outputs of running this
# program are:

# Inside thread: 4
# Inside thread: 0
# Main thread: done.

# OR

# Inside thread: 0
# Inside thread: 4
# Main thread: done.

# depending on whether t1 or t2 acquires the lock first.

# 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. It is invoked immediately after definition.
thread t1 = {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. It is invoked immediately after definition.
thread t2 = {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.

3.10.6 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 **3.4** 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_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_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
```

3.11 sPool Standard Library

The standard library in sPool is imported to the sPool program when one compiles the program with the `-stdlib` flag added when using the compile.sh script.

3.11.1 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_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,`	Folds a list from left to right, given a

	`list<int> -> int)`	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. Implementations for these functions are provided in section **8.2.1** in the **Appendix**. 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_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]
```

3.11.2 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.

None of these functions modify the original input string. Implementations are provided in section **8.2.2** in the **Appendix**. 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
```

4 Project Plan

4.1 Planning, specification and development

During the planning, specification, and development phases, our team implemented a structured approach to ensure progress and collaboration. We conducted Zoom meetings three times a week, where we worked collectively to tackle the problem at hand, promoting effective communication, brainstorming, and the exchange of ideas. These frequent meetings facilitated a deeper understanding of the project requirements, enabling us to refine our strategies and solutions.

In addition to the collaborative sessions, each of us occasionally focused on writing test cases (for all lexer/parser, semantic analysis and codegen phrases) for the compiler individually. This independent work allowed us to contribute to the project's quality assurance by identifying potential issues and ensuring that the compiler met the desired specifications. Combining both collaborative and individual efforts enabled us to achieve a well-rounded development process that addressed various aspects of the project, ultimately leading to a more robust and reliable outcome.

4.2 Project timeline

The following table summarizes the completion date of our deliverables and implementation of features of sPool.

Completion Date	Deliverables/Features
02/01/2023	Project Proposal
02/23/2023	Scanner / Parser
02/27/2023	Language Reference Manual
03/15/2023	Semantic Analysis
03/29/2023	Hello World!
04/11/2023	Threads
04/13/2023	Lists
04/26/2023	Closure and HOFs
04/27/2023	Mutex
05/02/2023	Store for `(int->int)` functions

4.3 Roles and responsibilities

Initially, we assigned the following roles and responsibilities to each member of Team Nautilus:
Ankur - Tester
Max - Manager
Etha - System Architect
Yuma - Language Guru

As the project progressed, we discovered that strictly adhering to the initially assigned roles was not the most effective approach for our team. Instead, we shifted towards a more engaging and collaborative working style that prioritized teamwork.

In our new approach, we do not assign specific tasks to individual members; rather, we all work together to tackle the same problem. By holding meetings three times a week, each lasting 2 to 3 hours, we create ample opportunities for engaging in collective coding sessions using Live Share in VSCode. This real-time collaboration allows each of us to actively participate in and contribute to the development of every feature in sPool.

The experience of working together in such a closely-knit manner facilitates a deep understanding of each feature and promotes instant communication of thoughts and ideas related to the development process. As a result, we benefit from a cohesive, unified approach that uses the diverse skills and perspectives of each member, fostering an environment that encourages creativity, problem-solving, and shared ownership of the project.

4.4 Development environment

We used the following development/version-control software:

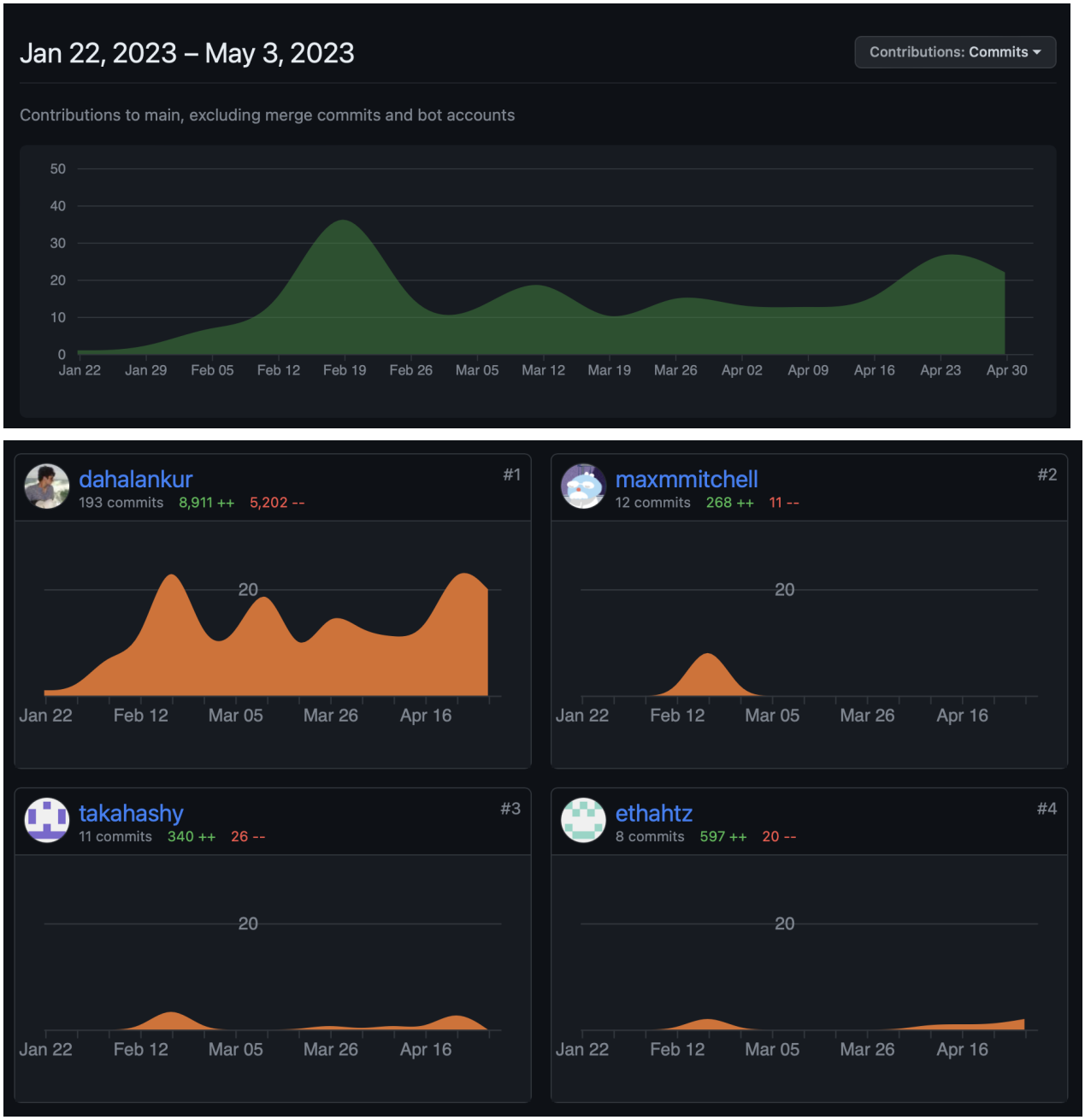
- Opam 2.1.2
- LLVM 14.0.6
- Python3 >=3.8.10
- gcc
- llc >= 14.0
- git for version control

The development of the sPool compiler relies on the basic OCaml toolchain, which includes OCaml, ocamlbuild, ocamllex, ocamlyacc, ocamlfind, and opam. Additionally, LLVM is utilized, both on the host machine and through the OCaml LLVM API installed via opam. Python3 is used for creating and running testing scripts, with compatibility confirmed for versions 3.8.10 and higher (although lower versions should still function adequately).

4.5 Commit history visualization

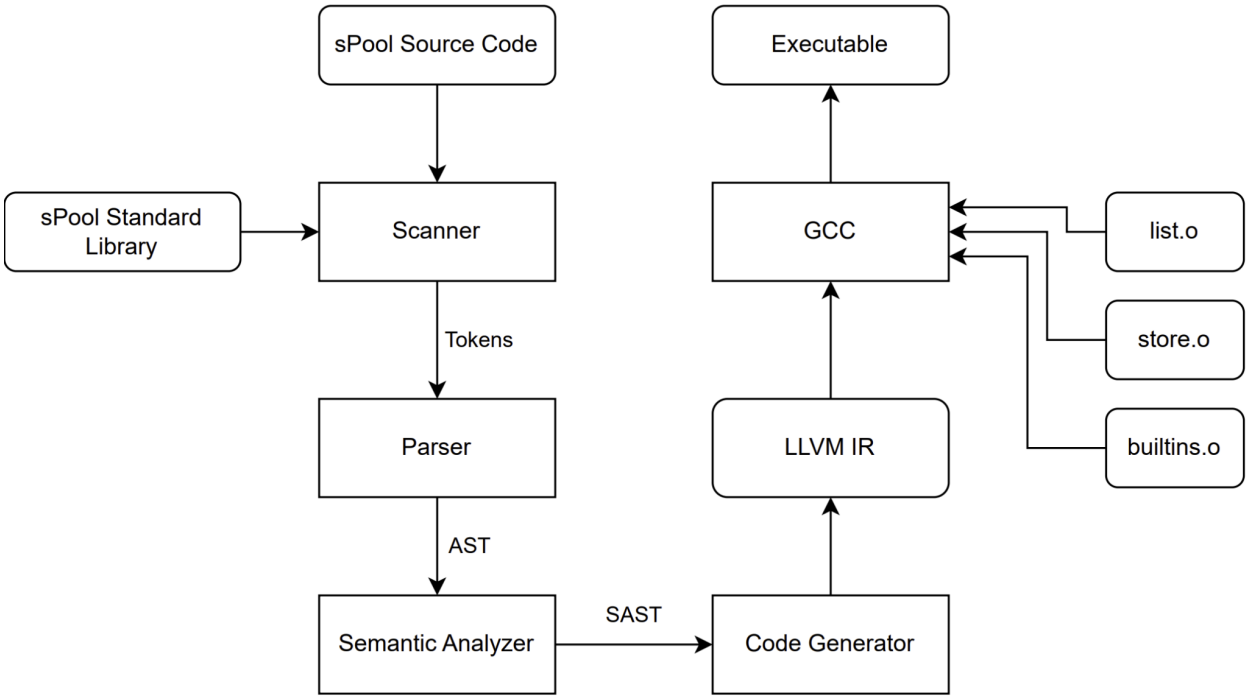
The visualization below represents our commit history. A majority of the commits are attributed to Ankur's GitHub account because we primarily collaborated using Live Share on Ankur's

machine. The remaining commits showcase individual contributions, including minor code modifications and the creation of test files for various features.



5 Architectural Design

This section discusses the architectural design of the sPool compiler and discusses how different compiler phases are linked with each other in the entire compilation process. Furthermore, section **5.1** goes on to describe how sPool's interesting language features were implemented.



The components with rounded rectangles in the aforementioned block diagram refer to data files (source files, intermediate files, output files, etc.) and the rectangular components with sharp corners refer to the individual phase of compilation. *Each and every component was implemented together by Team Nautilus.*

When a program written in sPool is compiled, it first passes through the scanner along with the standard library, if specified with the -stdlib flag during compilation. The compiler reads the source code character by character and parses them into tokens which are then passed on to the sPool parser. In the parsing stage, the tokens are analyzed by the compiler in order to create an abstract syntax tree (AST) that represents the program's structure. This AST is passed to the semantic analyzer where it checks for semantic errors and performs type checking to ensure that the program is well-formed and semantically correct, and produces a semantically checked abstract syntax tree (SAST). Finally, in the code generation phase, it translates the SAST into LLVM IR (our low-level intermediate representation) where it is then linked with other object files (list.o, store.o, builtins.o) using gcc, producing the final executable.

5.1 Interesting Language Features

This section of the document describes how sPool's interesting language features are implemented. *All of the interesting features in sPool were implemented as a team by all members of Team Nautilus.*

5.1.1 Lists

Under the hood, a list data structure in sPool is just a singly linked list, where each node is a struct containing a pointer to a value on the heap and a pointer to the next node in the list. Most of the bookkeeping for lists is done in the code generation phase, which directly generates the corresponding LLVM IR. Functions that manipulate a list, such as inserting an element into a list and removing an element from a list are implemented in C in the list.c file and are later linked against the generated LLVM IR. Furthermore, the code generation phase is responsible for generating LLVM instructions to ensure that lists are passed by reference to functions and captured by reference in closures. The code generation phase also handles successful preservation of references when list variables are being reassigned to other list variables or list literals.

5.1.2 Concurrency

In order to save time and avoid having to generate code to interface with syscalls in LLVM, we leveraged the POSIX Threads (commonly known as pthreads) library to implement concurrency via multithreading in sPool. Variables of the thread type are actually instances of pthread_t under the hood. Invocation and joining for threads are simply mappings to pthread_create and pthread_join functions from the pthread library. For all statements written inside the curly braces of a thread instance in sPool, we do the following:

1. Create a function, with its body containing the `statement_list` wrapped within curly braces in the sPool code.
2. Generate LLVM instructions to initialize a pthread_t instance.
3. Cast the function created in step 1 to a function that returns a void pointer and takes in a void pointer in order to comply with the signature of pthread_create.
4. Generate a call instruction to pthread_create with the function pointer obtained from 3 and pthread_t instance obtained from step 2.

This sequence of events successfully creates an anonymous function for each thread construct in sPool and dispatches it to an asynchronous thread using pthread_create. This is how threads are implemented in sPool – they interface with the pthread library and do some bookkeeping before dispatching statements surrounded by curly braces to an asynchronous thread using pthread_new. pthread_create by default invokes thread when called, so it all works out for how threads are invoked in sPool (which is on creation by default).

Similarly, to wait for threads we keep track of the pthread_t instance associated with each created thread. Therefore, generating an LLVM call instruction to pthread_join with this pthread_t instance is enough to wait on that particular thread to be done running.

For synchronization primitives, sPool uses mutexes (or mutual exclusion locks). As before, the pthread library's pthread_mutex_t instance is mapped to each mutex value in sPool. Therefore, any calls made in a sPool source file to `Mutex_lock` and `Mutex_unlock` functions directly map to the pthread library's pthread_mutex_lock and pthread_mutex_unlock functions respectively.

5.1.3 Store (Automatic Memoization)

The store functionality in sPool is implemented in two phases. The first phase occurs when the function is defined, and handles initialization of a global LLVM struct which will store the 32 most recent, unique results returned from the function. The second phase happens during the actual function call to a function that has store enabled, and does the following steps:

1. Look up the global store struct associated with the function that is being called.
2. Look for a cached result, using the parameter passed to the function as a key.
    a. If a hit is found, extract the result from the struct. Return the result in the struct and do **not** call the function.
    b. If a hit was **not** found, call the function to get the result. Store it into the struct for future use, along with the parameter passed, before returning the result as a normal function would.

For step 1, we generate LLVM IR directly in the code generation phase to initialize a struct for a store function that has the following structure:

```
typedef struct Elem {
    int param;
    int result;
} Elem;

typedef struct Store {
    int curr_index;
    int full; // used as bool to record whether it is full
    Elem stored[STORE_SIZE]; // where STORE_SIZE is 32 by default
} Store;
```

For step 2, to look up and update cache during runtime, the `store_lookup` and `store_insert` functions implemented in C are used. `store_lookup` takes in a global store struct and an integer (the actual parameter), and it goes through the stored array to find whether the actual parameter has already been called before. If so, it returns the returned value from the array, otherwise it returns `INT_MIN` as a default value to indicate that the parameter for that function has not been inserted in the cache. `store_insert`, on the other hand, takes in a global struct, a parameter and a returned value from the function, and it inserts the `<param, result> Elem` struct at the next available space in the cache, updating old values when the cache is full.

Lastly, the code generation phase generates a conditional branch instruction to make decisions at runtime whether to return a cached result, or call the function, based on results from `store_lookup`. Due to time constraints, we only support store functions for functions of type

`(int->int)` but theoretically, our approach could be used to enable automatic memoization for any function type in sPool.

5.1.4 First-class Functions and Closures

Functions as first-class citizens are implemented through the use of function pointers. This allows functions to be passed as arguments, stored in variables, and returned as values from other functions. This means that whenever functions are passed as parameters or assigned to variables, the underlying function pointer pointed to by the variable is passed instead to facilitate functions as first-class citizens.

For closures, we used the traditional approach of capturing the environment when a function is defined. All the variables present in the environment are captured in an explicit closure struct for each function defined. This struct is then unpacked when generating LLVM IR for the body of the function. This can be summarized in a two-step process shown below:

1. When a function is defined, it captures all variables in the current environment and stores their values (or references to the heap location of these values for shared variables, lists, mutexes, and functions) in a **globally** accessible LLVM closure structure specific to that function.
2. When the function is called, it first retrieves the captured values stored in the global closure structure and adds them into the scope of the function body, like any other local variable defined inside the function body. After this step of unpacking the values from the closure struct is done, instructions for the function body are generated normally. The free variables in the function can then be referenced. They are now treated as any locally-defined variables would be from the perspective of the function.

6 Test Plan

This section explains how we approached unit and integration testing and what automation was used in the testing process, along with providing some example sPool programs and its corresponding LLVM IR.

6.1 Representative sPool Programs

6.1.1 Higher Order Function

The following shows an example sPool program where functions are being passed as arguments to other functions.

```
def int square(int a):
    return a * a
;

def int add5(int a):
    return a + 5
;

def quack callFunc((int->int) fun):
    println(int_to_string(fun(10)))
    return
;

callFunc(square)
callFunc(add5)
```

The corresponding LLVM IR generated for the aforementioned sPool source file is shown below:

```
; ModuleID = 'sPool'
source_filename = "sPool"

%"square_closure_struct#" = type {}
%"add5_closure_struct#" = type { %"(add5):square:" }
%"(add5):square:" = type { i32 (i32)* }
%"callFunc_closure_struct#" = type { %"(callFunc):add5:", %"(callFunc):square:" }
%"(callFunc):add5:" = type { i32 (i32)* }
%"(callFunc):square:" = type { i32 (i32)* }
%Node = type opaque
```

```
%pthread_t = type opaque
%pthread_mutex_t = type opaque

@strfmt = private unnamed_addr constant [3 x i8] c"%s\00", align 1
@strfmtendline = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@"global_square_closure#" = global %"square_closure_struct#" zeroinitializer
@"global_add5_closure#" = global %"add5_closure_struct#" zeroinitializer
@"global_callFunc_closure#" = global %"callFunc_closure_struct#" zeroinitializer

define i32 @main() {
entry:
 %add5__struct__square = alloca %"(add5):square:", align 8
 %v = getelementptr inbounds %"(add5):square:", %"(add5):square:"*
%add5__struct__square, i32 0, i32 0
 store i32 (i32)* @square, i32 (i32)** %v, align 8
 %"(add5):square:" = load %"(add5):square:", %"(add5):square:"* %add5__struct__square,
align 8
 store %"(add5):square:" %"(add5):square:", %"(add5):square:"* getelementptr inbounds
(%"add5_closure_struct#", %"add5_closure_struct#"* @"global_add5_closure#", i32 0, i32
0), align 8
 %callFunc__struct__add5 = alloca %"(callFunc):add5:", align 8
 %v1 = getelementptr inbounds %"(callFunc):add5:", %"(callFunc):add5:"*
%callFunc__struct__add5, i32 0, i32 0
 store i32 (i32)* @add5, i32 (i32)** %v1, align 8
 %"(callFunc):add5:" = load %"(callFunc):add5:", %"(callFunc):add5:"*
%callFunc__struct__add5, align 8
 %callFunc__struct__square = alloca %"(callFunc):square:", align 8
 %v2 = getelementptr inbounds %"(callFunc):square:", %"(callFunc):square:"*
%callFunc__struct__square, i32 0, i32 0
 store i32 (i32)* @square, i32 (i32)** %v2, align 8
 %"(callFunc):square:" = load %"(callFunc):square:", %"(callFunc):square:"*
%callFunc__struct__square, align 8
 store %"(callFunc):add5:" %"(callFunc):add5:", %"(callFunc):add5:"* getelementptr
inbounds (%"callFunc_closure_struct#", %"callFunc_closure_struct#"*
@"global_callFunc_closure#", i32 0, i32 0), align 8
```

```
 store %"(callFunc):square:" %"(callFunc):square:", %"(callFunc):square:"*
getelementptr inbounds (%"callFunc_closure_struct#", %"callFunc_closure_struct#"*
@"global_callFunc_closure#", i32 0, i32 1), align 8
 call void @callFunc(i32 (i32)* @square)
 call void @callFunc(i32 (i32)* @add5)
 ret i32 0
}

declare i32 @printf(i8*, ...)

declare i8* @int_to_string(i32)

declare i8* @float_to_string(double)

declare i8* @bool_to_string(i1)

declare double @int_to_float(i32)

declare i32 @float_to_int(double)

declare i32 @strlen(i8*)

declare i8* @string_concat(i8*, i8*)

declare i8* @string_substr(i8*, i32, i32)

declare i1 @string_eq(i8*, i8*)

declare void @List_insert({ i8*, %Node* }**, i32, i8*)

declare i32 @List_len({ i8*, %Node* }**)

declare void @List_remove({ i8*, %Node* }**, i32)

declare void @List_replace({ i8*, %Node* }**, i32, i8*)
```

```
declare i8* @List_at({ i8*, %Node* }**, i32)

declare void @store_insert(i8*, i32, i32)

declare i32 @store_lookup(i8*, i32)

declare i32 @pthread_create(%pthread_t**, i8*, i8* (i8*)*, i8*)

declare i32 @pthread_join(%pthread_t*, i8**)

declare %pthread_mutex_t** @Mutex_init()

declare i32 @pthread_mutex_lock(%pthread_mutex_t*)

declare i32 @pthread_mutex_unlock(%pthread_mutex_t*)

define i32 @square(i32 %a) {
entry:
 %a1 = alloca i32, align 4
 store i32 %a, i32* %a1, align 4
 %a2 = load i32, i32* %a1, align 4
 %a3 = load i32, i32* %a1, align 4
 %tmp = mul i32 %a2, %a3
 ret i32 %tmp
}

define i32 @add5(i32 %a) {
entry:
 %individual_data_struct = load %"(add5):square:", %"(add5):square:"* getelementptr
inbounds (%"add5_closure_struct#", %"add5_closure_struct#"* @"global_add5_closure#",
i32 0, i32 0), align 8
 %0 = alloca %"(add5):square:", align 8
 store %"(add5):square:" %individual_data_struct, %"(add5):square:"* %0, align 8
 %1 = getelementptr inbounds %"(add5):square:", %"(add5):square:"* %0, i32 0, i32 0
 %square = load i32 (i32)*, i32 (i32)** %1, align 8
 %a1 = alloca i32, align 4
```

```
 store i32 %a, i32* %a1, align 4
 %a2 = load i32, i32* %a1, align 4
 %tmp = add i32 %a2, 5
 ret i32 %tmp
}

define void @callFunc(i32 (i32)* %fun) {
entry:
 %individual_data_struct = load %"(callFunc):add5:", %"(callFunc):add5:"*
getelementptr inbounds (%"callFunc_closure_struct#", %"callFunc_closure_struct#"*
@"global_callFunc_closure#", i32 0, i32 0), align 8
 %0 = alloca %"(callFunc):add5:", align 8
 store %"(callFunc):add5:" %individual_data_struct, %"(callFunc):add5:"* %0, align 8
 %1 = getelementptr inbounds %"(callFunc):add5:", %"(callFunc):add5:"* %0, i32 0, i32
0
 %add5 = load i32 (i32)*, i32 (i32)** %1, align 8
 %individual_data_struct1 = load %"(callFunc):square:", %"(callFunc):square:"*
getelementptr inbounds (%"callFunc_closure_struct#", %"callFunc_closure_struct#"*
@"global_callFunc_closure#", i32 0, i32 1), align 8
 %2 = alloca %"(callFunc):square:", align 8
 store %"(callFunc):square:" %individual_data_struct1, %"(callFunc):square:"* %2,
align 8
 %3 = getelementptr inbounds %"(callFunc):square:", %"(callFunc):square:"* %2, i32 0,
i32 0
 %square = load i32 (i32)*, i32 (i32)** %3, align 8
 %fun_result = call i32 %fun(i32 10)
 %int_to_string = call i8* @int_to_string(i32 %fun_result)
 %println = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
i8]* @strfmtendline, i32 0, i32 0), i8* %int_to_string)
 ret void
}
```

6.1.2 Threads

The following shows an example sPool program where two threads target the same function and asynchronously print to stdout. The output is nondeterministic, for any thread can finish executing the function first.

```
def quack printHello(int id):
   println(String_concat(("Hello! I am thread "), int_to_string(id)))
   return;

# invoke two threads that asynchronously target the "printHello" function
int i = 1
thread t1 = { printHello(i) }
i = i + 1
thread t2 = { printHello(i) }

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

The corresponding LLVM IR generated for this sPool program is given below:

```
; ModuleID = 'sPool'
source_filename = "sPool"

%"printHello_closure_struct#" = type {}
%"#anon_1_closure_struct#" = type { %"(#anon_1):i:", %"(#anon_1):printHello:" }
%"(#anon_1):i:" = type { i32 }
%"(#anon_1):printHello:" = type { void (i32)* }
%"#anon_2_closure_struct#" = type { %"(#anon_2):t1:", %"(#anon_2):i:",
%"(#anon_2):#anon_1:", %"(#anon_2):printHello:" }
%"(#anon_2):t1:" = type { %pthread_t* }
%pthread_t = type opaque
%"(#anon_2):i:" = type { i32 }
%"(#anon_2):#anon_1:" = type { void ()* }
%"(#anon_2):printHello:" = type { void (i32)* }
%Node = type opaque
%pthread_mutex_t = type opaque

@strfmt = private unnamed_addr constant [3 x i8] c"%s\00", align 1
@strfmtendline = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@"global_printHello_closure#" = global %"printHello_closure_struct#" zeroinitializer
```

```
@strlit = private unnamed_addr constant [20 x i8] c"Hello! I am thread \00", align 1
@"global_#anon_1_closure#" = global %"#anon_1_closure_struct#" zeroinitializer
@"global_#anon_2_closure#" = global %"#anon_2_closure_struct#" zeroinitializer

define i32 @main() {
entry:
 %i = alloca i32, align 4
 store i32 1, i32* %i, align 4
 %i1 = load i32, i32* %i, align 4
 %"#anon_1__struct__i" = alloca %"(#anon_1):i:", align 8
 %v = getelementptr inbounds %"(#anon_1):i:", %"(#anon_1):i:"* %"#anon_1__struct__i",
i32 0, i32 0
 store i32 %i1, i32* %v, align 4
 %"(#anon_1):i:" = load %"(#anon_1):i:", %"(#anon_1):i:"* %"#anon_1__struct__i", align
4
 %"#anon_1__struct__printHello" = alloca %"(#anon_1):printHello:", align 8
 %v2 = getelementptr inbounds %"(#anon_1):printHello:", %"(#anon_1):printHello:"*
%"#anon_1__struct__printHello", i32 0, i32 0
 store void (i32)* @printHello, void (i32)** %v2, align 8
 %"(#anon_1):printHello:" = load %"(#anon_1):printHello:", %"(#anon_1):printHello:"*
%"#anon_1__struct__printHello", align 8
 store %"(#anon_1):i:" %"(#anon_1):i:", %"(#anon_1):i:"* getelementptr inbounds
(%"#anon_1_closure_struct#", %"#anon_1_closure_struct#"* @"global_#anon_1_closure#",
i32 0, i32 0), align 4
 store %"(#anon_1):printHello:" %"(#anon_1):printHello:", %"(#anon_1):printHello:"*
getelementptr inbounds (%"#anon_1_closure_struct#", %"#anon_1_closure_struct#"*
@"global_#anon_1_closure#", i32 0, i32 1), align 8
 %pthread_t = alloca %pthread_t*, align 8
 %0 = call i32 @pthread_create(%pthread_t** %pthread_t, i8* null, i8* (i8*)* bitcast
(void ()* @"#anon_1" to i8* (i8*)*), i8* null)
 %pthread_t3 = load %pthread_t*, %pthread_t** %pthread_t, align 8
 %t1 = alloca %pthread_t*, align 8
 store %pthread_t* %pthread_t3, %pthread_t** %t1, align 8
 %i4 = load i32, i32* %i, align 4
 %tmp = add i32 %i4, 1
 store i32 %tmp, i32* %i, align 4
```

```
 %i5 = load i32, i32* %i, align 4
 %t16 = load %pthread_t*, %pthread_t** %t1, align 8
 %"#anon_2__struct__t1" = alloca %"(#anon_2):t1:", align 8
 %v7 = getelementptr inbounds %"(#anon_2):t1:", %"(#anon_2):t1:"*
%"#anon_2__struct__t1", i32 0, i32 0
 store %pthread_t* %t16, %pthread_t** %v7, align 8
 %"(#anon_2):t1:" = load %"(#anon_2):t1:", %"(#anon_2):t1:"* %"#anon_2__struct__t1",
align 8
 %"#anon_2__struct__i" = alloca %"(#anon_2):i:", align 8
 %v8 = getelementptr inbounds %"(#anon_2):i:", %"(#anon_2):i:"* %"#anon_2__struct__i",
i32 0, i32 0
 store i32 %i5, i32* %v8, align 4
 %"(#anon_2):i:" = load %"(#anon_2):i:", %"(#anon_2):i:"* %"#anon_2__struct__i", align
4
 %"#anon_2__struct__#anon_1" = alloca %"(#anon_2):#anon_1:", align 8
 %v9 = getelementptr inbounds %"(#anon_2):#anon_1:", %"(#anon_2):#anon_1:"*
%"#anon_2__struct__#anon_1", i32 0, i32 0
 store void ()* @"#anon_1", void ()** %v9, align 8
 %"(#anon_2):#anon_1:" = load %"(#anon_2):#anon_1:", %"(#anon_2):#anon_1:"*
%"#anon_2__struct__#anon_1", align 8
 %"#anon_2__struct__printHello" = alloca %"(#anon_2):printHello:", align 8
 %v10 = getelementptr inbounds %"(#anon_2):printHello:", %"(#anon_2):printHello:"*
%"#anon_2__struct__printHello", i32 0, i32 0
 store void (i32)* @printHello, void (i32)** %v10, align 8
 %"(#anon_2):printHello:" = load %"(#anon_2):printHello:", %"(#anon_2):printHello:"*
%"#anon_2__struct__printHello", align 8
 store %"(#anon_2):t1:" %"(#anon_2):t1:", %"(#anon_2):t1:"* getelementptr inbounds
(%"#anon_2_closure_struct#", %"#anon_2_closure_struct#"* @"global_#anon_2_closure#",
i32 0, i32 0), align 8
 store %"(#anon_2):i:" %"(#anon_2):i:", %"(#anon_2):i:"* getelementptr inbounds
(%"#anon_2_closure_struct#", %"#anon_2_closure_struct#"* @"global_#anon_2_closure#",
i32 0, i32 1), align 4
 store %"(#anon_2):#anon_1:" %"(#anon_2):#anon_1:", %"(#anon_2):#anon_1:"*
getelementptr inbounds (%"#anon_2_closure_struct#", %"#anon_2_closure_struct#"*
@"global_#anon_2_closure#", i32 0, i32 2), align 8
```

```
 store %"(#anon_2):printHello:" %"(#anon_2):printHello:", %"(#anon_2):printHello:"*
getelementptr inbounds (%"#anon_2_closure_struct#", %"#anon_2_closure_struct#"*
@"global_#anon_2_closure#", i32 0, i32 3), align 8
 %pthread_t11 = alloca %pthread_t*, align 8
 %1 = call i32 @pthread_create(%pthread_t** %pthread_t11, i8* null, i8* (i8*)* bitcast
(void ()* @"#anon_2" to i8* (i8*)*), i8* null)
 %pthread_t12 = load %pthread_t*, %pthread_t** %pthread_t11, align 8
 %t2 = alloca %pthread_t*, align 8
 store %pthread_t* %pthread_t12, %pthread_t** %t2, align 8
 %t113 = load %pthread_t*, %pthread_t** %t1, align 8
 %2 = call i32 @pthread_join(%pthread_t* %t113, i8** null)
 %t214 = load %pthread_t*, %pthread_t** %t2, align 8
 %3 = call i32 @pthread_join(%pthread_t* %t214, i8** null)
 ret i32 0
}

declare i32 @printf(i8*, ...)

declare i8* @int_to_string(i32)

declare i8* @float_to_string(double)

declare i8* @bool_to_string(i1)

declare double @int_to_float(i32)

declare i32 @float_to_int(double)

declare i32 @strlen(i8*)

declare i8* @string_concat(i8*, i8*)

declare i8* @string_substr(i8*, i32, i32)

declare i1 @string_eq(i8*, i8*)
```

```
declare void @List_insert({ i8*, %Node* }**, i32, i8*)

declare i32 @List_len({ i8*, %Node* }**)

declare void @List_remove({ i8*, %Node* }**, i32)

declare void @List_replace({ i8*, %Node* }**, i32, i8*)

declare i8* @List_at({ i8*, %Node* }**, i32)

declare void @store_insert(i8*, i32, i32)

declare i32 @store_lookup(i8*, i32)

declare i32 @pthread_create(%pthread_t**, i8*, i8* (i8*)*, i8*)

declare i32 @pthread_join(%pthread_t*, i8**)

declare %pthread_mutex_t** @Mutex_init()

declare i32 @pthread_mutex_lock(%pthread_mutex_t*)

declare i32 @pthread_mutex_unlock(%pthread_mutex_t*)

define void @printHello(i32 %id) {
entry:
 %id1 = alloca i32, align 4
 store i32 %id, i32* %id1, align 4
 %id2 = load i32, i32* %id1, align 4
 %int_to_string = call i8* @int_to_string(i32 %id2)
 %string_concat = call i8* @string_concat(i8* getelementptr inbounds ([20 x i8], [20 x
i8]* @strlit, i32 0, i32 0), i8* %int_to_string)
 %println = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
i8]* @strfmtendline, i32 0, i32 0), i8* %string_concat)
 ret void
}
```

```
define void @"#anon_1"() {
entry:
 %individual_data_struct = load %"(#anon_1):i:", %"(#anon_1):i:"* getelementptr
inbounds (%"#anon_1_closure_struct#", %"#anon_1_closure_struct#"*
@"global_#anon_1_closure#", i32 0, i32 0), align 4
 %0 = alloca %"(#anon_1):i:", align 8
 store %"(#anon_1):i:" %individual_data_struct, %"(#anon_1):i:"* %0, align 4
 %1 = getelementptr inbounds %"(#anon_1):i:", %"(#anon_1):i:"* %0, i32 0, i32 0
 %i = load i32, i32* %1, align 4
 %i_ptr = alloca i32, align 4
 store i32 %i, i32* %i_ptr, align 4
 %individual_data_struct1 = load %"(#anon_1):printHello:", %"(#anon_1):printHello:"*
getelementptr inbounds (%"#anon_1_closure_struct#", %"#anon_1_closure_struct#"*
@"global_#anon_1_closure#", i32 0, i32 1), align 8
 %2 = alloca %"(#anon_1):printHello:", align 8
 store %"(#anon_1):printHello:" %individual_data_struct1, %"(#anon_1):printHello:"*
%2, align 8
 %3 = getelementptr inbounds %"(#anon_1):printHello:", %"(#anon_1):printHello:"* %2,
i32 0, i32 0
 %printHello = load void (i32)*, void (i32)** %3, align 8
 %i2 = load i32, i32* %i_ptr, align 4
 call void %printHello(i32 %i2)
 ret void
}

define void @"#anon_2"() {
entry:
 %individual_data_struct = load %"(#anon_2):t1:", %"(#anon_2):t1:"* getelementptr
inbounds (%"#anon_2_closure_struct#", %"#anon_2_closure_struct#"*
@"global_#anon_2_closure#", i32 0, i32 0), align 8
 %0 = alloca %"(#anon_2):t1:", align 8
 store %"(#anon_2):t1:" %individual_data_struct, %"(#anon_2):t1:"* %0, align 8
 %1 = getelementptr inbounds %"(#anon_2):t1:", %"(#anon_2):t1:"* %0, i32 0, i32 0
 %t1 = load %pthread_t*, %pthread_t** %1, align 8
 %t1_ptr = alloca %pthread_t*, align 8
```

```
 store %pthread_t* %t1, %pthread_t** %t1_ptr, align 8
 %individual_data_struct1 = load %"(#anon_2):i:", %"(#anon_2):i:"* getelementptr
inbounds (%"#anon_2_closure_struct#", %"#anon_2_closure_struct#"*
@"global_#anon_2_closure#", i32 0, i32 1), align 4
 %2 = alloca %"(#anon_2):i:", align 8
 store %"(#anon_2):i:" %individual_data_struct1, %"(#anon_2):i:"* %2, align 4
 %3 = getelementptr inbounds %"(#anon_2):i:", %"(#anon_2):i:"* %2, i32 0, i32 0
 %i = load i32, i32* %3, align 4
 %i_ptr = alloca i32, align 4
 store i32 %i, i32* %i_ptr, align 4
 %individual_data_struct2 = load %"(#anon_2):#anon_1:", %"(#anon_2):#anon_1:"*
getelementptr inbounds (%"#anon_2_closure_struct#", %"#anon_2_closure_struct#"*
@"global_#anon_2_closure#", i32 0, i32 2), align 8
 %4 = alloca %"(#anon_2):#anon_1:", align 8
 store %"(#anon_2):#anon_1:" %individual_data_struct2, %"(#anon_2):#anon_1:"* %4,
align 8
 %5 = getelementptr inbounds %"(#anon_2):#anon_1:", %"(#anon_2):#anon_1:"* %4, i32 0,
i32 0
 %"#anon_1" = load void ()*, void ()** %5, align 8
 %individual_data_struct3 = load %"(#anon_2):printHello:", %"(#anon_2):printHello:"*
getelementptr inbounds (%"#anon_2_closure_struct#", %"#anon_2_closure_struct#"*
@"global_#anon_2_closure#", i32 0, i32 3), align 8
 %6 = alloca %"(#anon_2):printHello:", align 8
 store %"(#anon_2):printHello:" %individual_data_struct3, %"(#anon_2):printHello:"*
%6, align 8
 %7 = getelementptr inbounds %"(#anon_2):printHello:", %"(#anon_2):printHello:"* %6,
i32 0, i32 0
 %printHello = load void (i32)*, void (i32)** %7, align 8
 %i4 = load i32, i32* %i_ptr, align 4
 call void %printHello(i32 %i4)
 ret void
}
```

6.1.3 Automatic Memoization

The following shows an example sPool program where automatic memoization is used. Specifically, a factorial function is written with the store keyword which instructs the compiler to

cache the return values for this function. 120 is the expected output of the following sPool program, for 5! = 120.

```
def store int factorial(int n):
   int result = 1
   if (n <= 1):
       result = 1
   else
       result = n * factorial(n - 1);
   return result
;

int result = factorial(5)
println(int_to_string(result))
```

The corresponding LLVM IR generated for this sPool source file is shown below. Notice the factorial_store_struct created which caches at most 32 results for calls to the automatically memoized factorial function.

```
; ModuleID = 'sPool'
source_filename = "sPool"

%"factorial_store_struct#" = type { i32, i32, [32 x %"factorial_store_elem_struct#"] }
%"factorial_store_elem_struct#" = type { i32, i32 }
%"factorial_closure_struct#" = type {}
%Node = type opaque
%pthread_t = type opaque
%pthread_mutex_t = type opaque

@strfmt = private unnamed_addr constant [3 x i8] c"%s\00", align 1
@strfmtendline = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@"global_factorial_store#" = global %"factorial_store_struct#" zeroinitializer
@"global_factorial_closure#" = global %"factorial_closure_struct#" zeroinitializer

define i32 @main() {
entry:
 %lookup_result = call i32 @store_lookup(i8* bitcast (%"factorial_store_struct#"*
@"global_factorial_store#" to i8*), i32 5)
```

```
 %0 = icmp eq i32 %lookup_result, -2147483648
 %result_stackaddr = alloca i32, align 4
 br i1 %0, label %then, label %else

then:                                             ; preds = %entry
 %factorial_result = call i32 @factorial(i32 5)
 call void @store_insert(i8* bitcast (%"factorial_store_struct#"*
@"global_factorial_store#" to i8*), i32 5, i32 %factorial_result)
 store i32 %factorial_result, i32* %result_stackaddr, align 4
 br label %merge

else:                                             ; preds = %entry
 store i32 %lookup_result, i32* %result_stackaddr, align 4
 br label %merge

merge:                                            ; preds = %else, %then
 %result = load i32, i32* %result_stackaddr, align 4
 %result1 = alloca i32, align 4
 store i32 %result, i32* %result1, align 4
 %result2 = load i32, i32* %result1, align 4
 %int_to_string = call i8* @int_to_string(i32 %result2)
 %println = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
i8]* @strfmtendline, i32 0, i32 0), i8* %int_to_string)
 ret i32 0
}

declare i32 @printf(i8*, ...)

declare i8* @int_to_string(i32)

declare i8* @float_to_string(double)

declare i8* @bool_to_string(i1)

declare double @int_to_float(i32)
```

```
declare i32 @float_to_int(double)

declare i32 @strlen(i8*)

declare i8* @string_concat(i8*, i8*)

declare i8* @string_substr(i8*, i32, i32)

declare i1 @string_eq(i8*, i8*)

declare void @List_insert({ i8*, %Node* }**, i32, i8*)

declare i32 @List_len({ i8*, %Node* }**)

declare void @List_remove({ i8*, %Node* }**, i32)

declare void @List_replace({ i8*, %Node* }**, i32, i8*)

declare i8* @List_at({ i8*, %Node* }**, i32)

declare void @store_insert(i8*, i32, i32)

declare i32 @store_lookup(i8*, i32)

declare i32 @pthread_create(%pthread_t**, i8*, i8* (i8*)*, i8*)

declare i32 @pthread_join(%pthread_t*, i8**)

declare %pthread_mutex_t** @Mutex_init()

declare i32 @pthread_mutex_lock(%pthread_mutex_t*)

declare i32 @pthread_mutex_unlock(%pthread_mutex_t*)

define i32 @factorial(i32 %n) {
entry:
```

```
 %n1 = alloca i32, align 4
 store i32 %n, i32* %n1, align 4
 %result = alloca i32, align 4
 store i32 1, i32* %result, align 4
 %n2 = load i32, i32* %n1, align 4
 %tmp = icmp sle i32 %n2, 1
 br i1 %tmp, label %then, label %else

merge:                                            ; preds = %merge8, %then
 %result11 = load i32, i32* %result, align 4
 ret i32 %result11

then:                                             ; preds = %entry
 store i32 1, i32* %result, align 4
 br label %merge

else:                                             ; preds = %entry
 %n3 = load i32, i32* %n1, align 4
 %n4 = load i32, i32* %n1, align 4
 %tmp5 = sub i32 %n4, 1
 %lookup_result = call i32 @store_lookup(i8* bitcast (%"factorial_store_struct#"*
@"global_factorial_store#" to i8*), i32 %tmp5)
 %0 = icmp eq i32 %lookup_result, -2147483648
 %result_stackaddr = alloca i32, align 4
 br i1 %0, label %then6, label %else7

then6:                                            ; preds = %else
 %factorial_result = call i32 @factorial(i32 %tmp5)
 call void @store_insert(i8* bitcast (%"factorial_store_struct#"*
@"global_factorial_store#" to i8*), i32 %tmp5, i32 %factorial_result)
 store i32 %factorial_result, i32* %result_stackaddr, align 4
 br label %merge8

else7:                                            ; preds = %else
 store i32 %lookup_result, i32* %result_stackaddr, align 4
 br label %merge8
```

```
merge8:                                           ; preds = %else7, %then6
 %result9 = load i32, i32* %result_stackaddr, align 4
 %tmp10 = mul i32 %n3, %result9
 store i32 %tmp10, i32* %result, align 4
 br label %merge
}
```

6.2 Testing Workflow and Scripts

All work on testing was done together by Team Nautilus. Our testing included two test suites of cases: one for the Scanning and Parsing phases and another for the Semantic Analysis and Code Generation phases. For both test suites, we create a separate directory under the tests directory labeled lexerparser and codegen, respectively. In each of those directories there are subdirectories that unit test all features of our sPool program, including but not limited to if, strings, while, functions, lambdas, closures, threads, etc. Additionally, each of those subdirectories are composed of two files for each test case: one file specifying the sPool source code and another specifying the expected output. We added unit tests for each new component as we created. Our filenames for test cases followed this naming convention:

- Tests that are expected to succeed are named with
    - test-{testname[n]}.sP with their corresponding successful output as
    - test-{testname[n]}.out
- Tests that are expected to fail are named with
    - fail-{testname[n]}.sP with their corresponding error output as
    - fail-{testname[n]}.err

where **“**testname” is the feature being tested and “n” is the test number relative to the number of tests for that feature.

Our test script, runtests.py was written in Python3 and works together with the Makefile and the shell script compile.sh to perform testing in batches or individually.

1. **Run all tests:** To run all tests in a testing suite, type:
   make {testphase}
   where {testphase} is “testparser” or “testcodegen”. This automatically compiles the Compiler and runs the runtests.py python script

2. **Run all tests in a specific subdirectory:** To run all tests in a testname directory, type:
   python3 runtests.py {testphase} all {testname}
   where {testphase} is the name of the directory in sPool/tests/
   and {testname} is the name of the directory in sPool/tests/{testphase}/

3. **Run a single test:** To run a single test, type:
   python3 runtests.py {testphase} {testname} {filename}
   where {testphase} is the name of the directory in sPool/tests/,
   and {testname} is the name of the directory in sPool/tests/{testphase}/,
   and {filename} is the name of the file without the ".sP" extension in {testname}

When running these commands, the program will output whether the test/tests was/were successful or failed in some cases. In addition to these tests, we re-run all our lexerparser and codegen tests on every push to GitHub using GitHub's CI/CD workflow. This is an additional layer of regression testing that helps catch any untested additions made to the compiler source code.

6.2.1 Makefile

```
# Makefile
# Compiles the sPool compiler and runs tests
# Written by: Team Nautilus (Ankur, Yuma, Max, Etha)
#

EXEC := toplevel.native
CXX  := gcc

.PHONY: all clean testparser testcodegen testall

all: $(EXEC) builtins.o list.o store.o

testall: testparser testcodegen

testparser: $(EXEC) runtests.py
   @python3 runtests.py lexerparser all _

testcodegen: $(EXEC) runtests.py
   @python3 runtests.py codegen all _

%.o: %.c
   @$(CXX) -O2 -c -pthread $^ -o $@

$(EXEC): toplevel.ml parser.mly scanner.mll ast.ml semant.ml sast.ml codegen.ml
```

```
	@ocamlbuild $@ -package llvm -package llvm.analysis -cflags -w,+a-4-70-42

clean:
	@ocamlbuild -clean
	@rm -f *.o *.s *.ll *.exe
```

6.2.2 compile.sh

```
#!/bin/bash

# This script is used to compile a given sP file to the executable,
# generating any intermediate files in the process
# Usage: compile.sh [-stdlib] <sP file> <output file>
# Note: This script generates the .ll and .s files in the current working
#       directory and not in the directory where the sP file is present
#
# Written by: Team Nautilus (Ankur, Yuma, Max, Etha)

set -euo pipefail

LLC=llc

# check that llc is installed. If not, look for llc-14
if ! command -v $LLC &> /dev/null
then
    LLC=llc-14
    if ! command -v $LLC &> /dev/null
    then
        echo "llc could not be found. Please make sure llc or llc-14 is installed and
in your PATH."
        exit 1
    fi
fi

import=0

# check arguments
```

```
if [ $# -lt 2 ] || [ $# -gt 3 ]; then
   echo "Usage: compile.sh [-stdlib] <file.sP> <exec>"
   exit 1
fi

sP_file=$1
exec=$2

if [ $# -eq 3 ] && [ "$1" = "-stdlib" ]; then
   import=1
   sP_file=$2
   exec=$3
elif [ $# -eq 3 ] && [ "$1" != "-stdlib" ]; then
   echo "Usage: compile.sh [-stdlib] <file.sP> <exec>"
   exit 1
fi

if [ ! -f "$sP_file" ]; then
   echo "File $sP_file does not exist"
   exit 1
fi

sP_file_no_ext="${sP_file%.*}"
sP_file_no_ext="${sP_file_no_ext##*/}"
script_dir="$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null 2>&1 && pwd )"

# run make to compile the compiler
rm -f "$script_dir"/*.o
make -C "$script_dir"

# run the compiler on the sP file
if [ $import -eq 1 ]; then
   "$script_dir"/toplevel.native -i "$sP_file" > "$sP_file_no_ext".ll
else
   "$script_dir"/toplevel.native "$sP_file" > "$sP_file_no_ext".ll
fi
```

```
# compile the llvm file to assembly
"$LLC" -relocation-model=pic "$sP_file_no_ext".ll -o "$sP_file_no_ext".s

# link it with builtins.o and the pthread library
gcc -O2 -pthread "$sP_file_no_ext".s "$script_dir"/builtins.o "$script_dir"/list.o
"$script_dir"/store.o -o "$exec"
```

6.2.3 runtests.py

```
#  runtests.py
#  Run the tests for sPool
#  This script can be run as a standalone script, or from the Makefile
#  Usage: python3 runtests.py dir [all|testname] [filename],
#  where testname is the name of a test directory in tests/ and dir is the
#  directory containing the subdirectories of tests. filename is the individual
#  test to run if the second argument is not "all".
#
#  Written by Team Nautilus (ankur, yuma, max, etha) on 02/20/2023

import sys
import os
import glob
from subprocess import Popen, PIPE

SCRIPT_DIR = os.path.dirname(os.path.realpath(__file__))
EXECUTABLE = SCRIPT_DIR + "/toplevel.native"
ARGS = ""
FAILED = False

# Takes a string as an argument, the path to a test directory
# Runs the tests in the directory and reports the results
def run_test(codegen, test):
   global FAILED

   print("Testing " + test.split("/")[-1] + ":\n")
```

```
  if ".sP" in test:
      # individual testing
      if "test-" in test:
          success_tests = [test]
          success_expected = [test.replace(".sP", ".out")]
          fail_tests = []
          fail_expected = []
      else:
          success_tests = []
          success_expected = []
          fail_tests = [test]
          fail_expected = [test.replace(".sP", ".err")]
  else:
      # batch testing
      success_tests = sorted(glob.glob(test + "/test-*.sP"))
      success_expected = sorted(glob.glob(test + "/test-*.out"))
      fail_tests = sorted(glob.glob(test + "/fail-*.sP"))
      fail_expected = sorted(glob.glob(test + "/fail-*.err"))

  # Run the tests that should succeed
  for test, expected in zip(success_tests, success_expected):
      name = test.split("/")[-1]

      output = Popen([EXECUTABLE, ARGS], stderr=PIPE, stdout=PIPE, stdin=PIPE).\
          communicate(input=open(test, "rb").read())
      stdout = output[0].decode("utf-8")
      stderr = output[1].decode("utf-8")

      if stderr != "":
          print(f"\033[91mTest {name} FAILED.\033[0m Expected no errors,
got:\n{stderr}")
          FAILED = True
          continue

      if codegen:
```

```
            # run compile.sh script on input file to get the executable
            output = Popen([f"{SCRIPT_DIR}/compile.sh", test, name + ".exe"],
stderr=PIPE, stdout=PIPE).communicate()

            # run the executable and get the output
            output = Popen([f"{SCRIPT_DIR}/{name}.exe"], stderr=PIPE,
stdout=PIPE).communicate()
            stdout = output[0].decode("utf-8")
            stderr = output[1].decode("utf-8")
            if stderr != "":
                print(f"\033[91mTest {name} FAILED.\033[0m Expected no errors,
got:\n{stderr}")
                FAILED = True
                continue

        with open(expected, "r") as f: expected_output = f.read()

        # diff expected and actual output
        if stdout != expected_output:
            print(f"\033[91mTest {name} FAILED.\033[0m Expected output:
{expected_output}, got:\n{stdout}")
            FAILED = True
        else:
            print(f"\033[92mTest {name} PASSED.\033[0m")

    # Run the tests that should fail
    for test, expected in zip(fail_tests, fail_expected):
        name = test.split("/")[-1]

        # read from stderr instead of stdout
        output = Popen([EXECUTABLE, ARGS], stderr=PIPE, stdout=PIPE, stdin=PIPE).\
            communicate(input=open(test, "rb").read())[1].decode("utf-8")
        with open(expected, "r") as f: expected_output = f.read()

        # empty stderr indicates the test was expected to fail, but didn't
        if output == "" or output != expected_output:
```

```
            print(f"\033[91mTest {name} FAILED.\033[0m Expected output:
{expected_output}, got:\n{output}")
            FAILED = True
        else:
            # print this in GREEN color in the terminal
            print(f"\033[92mTest {name} PASSED.\033[0m")


    print("\n" + "-----------------" * 3 + "\n")

def cleanup(base_dir):
    # remove all the .ll and .exe files that were created
    for f in glob.glob(f"{base_dir}/*.ll", recursive=True):
        os.remove(f)
    for f in glob.glob(f"{base_dir}/*.exe", recursive=True):
        os.remove(f)
    for f in glob.glob(f"{base_dir}/*.s", recursive=True):
        os.remove(f)

# Takes a string as an argument, either "all" or the name of a test directory
# in tests/ and runs and reports the results of the tests
def main(to_test):
    global ARGS

    base_test_dir = to_test[0]
    to_test = to_test[1:]

    tests = []
    tests_dir = os.path.dirname(os.path.realpath(__file__)) +
f"/../tests/{base_test_dir}/"

    if to_test[0] == "all":
        if to_test[1] == "_":
            print("Running all tests\n")
            tests = glob.glob(tests_dir + "*")
```

```
        else:
            if os.path.exists(tests_dir + to_test[1]):
                tests = glob.glob(tests_dir + to_test[1])
            else:
                print(f"\033[91mThe requested directory does not exist.\033[0m\n")
                sys.exit(1)
    else:
        if os.path.exists(tests_dir + to_test[0] + "/" + to_test[1] + ".sP"):
            filename = tests_dir + to_test[0] + "/" + to_test[1] + ".sP"
            tests.append(filename)
        else:
            print("The requested test does not exist. Please ensure you typed the
correct name. \nTo double check the name of the test, look at the tests under the
tests/ directory.\n\nFor example, for running codegen tests on the test-if1.sP file,
run python3 runtests.py codegen if test-if1\n")

    if not os.path.exists(EXECUTABLE):
        print("toplevel.native not found. Please run 'make' first in the src
directory.")
        sys.exit(1)

    # send correct arguments to toplevel.native
    if base_test_dir == "lexerparser":
        ARGS += "-a"
    elif base_test_dir == "codegen":
        ARGS += "-c"

    for test in tests:
        run_test(base_test_dir == "codegen", test)

    cleanup(SCRIPT_DIR) # clean up intermediate files and executables generated by the
tests

    if FAILED:
        print(f"\033[91mSome tests failed. Please check the test output above.\033[0m")
        sys.exit(1)
```

```
    else:
        print(f"\033[92mAll tests passed.\033[0m")


if __name__ == '__main__':
    if len(sys.argv) != 4:
        print("Usage: python3 runtests.py testdir [all|testname] [filename]")
        # if all is given, filename is ignored
        # if testname is given, filename is the individual test file to run
        sys.exit(1)
    main(sys.argv[1:])
```

7 Lessons Learned

7.1 Ankur Dahal

One of the major lessons I learned from working on this project was the importance of having a clear and well-thought out implementation plan in mind before starting work on the project, especially for the codegen phase of the compiler. Whilst the microC compiler was a helpful representation of the entire “building a compiler” process, sPool had too many features not implemented in microC which made it difficult to organize everything at the beginning. Moreover, I also learned the motivation behind using a functional language like OCaml to build compilers and I can not imagine using imperative languages like C++ or Java to implement all phases of a compiler.

Another important takeaway from working on this project was understanding how programming language features like closures, structs, OOP (even though we do not have OOP in sPool), references, and garbage collection work under the hood. Some of the discoveries I made while researching for this project are simply mind boggling and have helped further my understanding of programming languages and language translators as a whole.

An advice I would give to future teams is to start early on codegen but only start meddling with the code once you have a solid understanding of what’s going on (i.e., what a module is, what types you are dealing with, what a builder is, what basic blocks are, etc.). This will be immensely helpful once you implement the “interesting features” of your language, for you will need to understand the ins and outs of the produced LLVM module from the codegen phase.

Moreover, a very helpful advice that I can give is to implement the feature you are trying to implement in your language in C first. Then, use gcc to compile the C source code to LLVM, observe the LLVM produced, and try producing the same target code in OCaml for your language. This saved our team a lot of time while working on closures, lists, and automatic memoization!

7.2 Max Mitchell

The most valuable lesson I learned from this project is that planning is a very powerful tool for tackling large-scale projects. A task that may seem herculean and near-impossible becomes much more reasonable when broken up into pieces. Setting up tangible deliverables that require more modest amounts of work and setting deadlines for each of them made the amount of effort required far more reasonable. Furthermore, it’s really key to know what you want before you start doing something. At every step of the compiler this proved valuable – considering what tokens we wanted before lexing, what tree we wanted before parsing, what LLVM we wanted before generating code, etc.

Another important takeaway for me was the value of team programming. Almost all of the code for this project was written with all four team members present, and the code that wasn’t was reviewed by all team members. Given the scale of this project, I’m proud we were able to manage that. The debugging process was very efficient with all four minds working together,

and even though it's unlikely I'll get the chance to program like this on future projects, it was a really fun and interesting experience.

The best advice I could give to future teams is to make a schedule and stick to it – even when you don't want to. We blocked out three meetings a week at 9am, every week, for a total of six or more hours weekly. There were a lot of weeks where it was exhausting to get up that early and just start working, but it proved incredibly helpful in staying on task. We never felt overwhelmed by the workload and we always felt like we could meet deadlines reasonably. Plus, we rarely had to meet or do work for the group project outside of those scheduled times. Scheduling meetings like this is really helpful in keeping everyone accountable and making sure progress is being made consistently. Not only that, but it also helped us develop good group chemistry and learn how to work together based on everyone's strengths.

7.3 Etha Hua

The biggest lesson I have learned from this project is that implementing a programming language is a very systematic process, and a fun one as well. Before taking compilers, I would think of the work of implementing a programming language to be super complicated. But when the compiler's work is divided into lexical analysis, syntactical analysis, type checking (with semantic analysis) and code generation, as long as each module works on its own, a compiler will automatically work! Although the entire compiler looked like a giant task in the beginning, when we implemented modules after modules one by one, making a working, functional compiler did not take a huge amount of time or code.

An advice I would give to future teams is to choose interesting language features during the design phase. With interesting programs that you could write in your programming language, you will be constantly motivated to implement your language with the aspiration to see it work at the end. We did see our programming language evolve from a simple grammar to an actual, working language, and it is super exciting that we implemented most of the features we were originally thinking of. This was a very rewarding experience.

7.4 Yuma Takahashi

Working on this compiler project was a challenging but very rewarding experience. Personally, I think the most important takeaway from this project was understanding the project's scope, the objectives to be accomplished, and the schedule for development. Creating a general plan for developing a compiler was crucial and that was done by breaking the project down to smaller more manageable components. The general structure of our compiler consisted of lexical analysis, parsing, semantic analysis, and code generation to LLVM IR. Each of these parts were further divided into smaller jobs or sub-parts, such as implementing a specific feature of sPool. We as a team discussed a development roadmap after identifying the key components and their sub-components. This worked very well for us because we could identify potential roadblocks or challenges early on during each phase.

To future teams working on compiler projects, I would say to prioritize communication and collaboration. Building a compiler is a complex task that requires contributions from all team members. Therefore, it's critical to create open lines of communication and make sure that everyone is aware of their responsibilities. In our case, we set up regular meetings of at least three times per week, where everyone worked together on the same file instead of dividing and delegating tasks. This way, everyone knew what was being implemented and how. However, since this might not be possible, communication is key when dividing up tasks.

8 Appendix

8.1 Translator

This section contains code listing for the entire sPool translator. To view the entire source code on GitHub, visit this link.

8.1.1 toplevel.ml

```
(* toplevel.ml
  The top level driver program for the sPool compiler.
  Allows the user to select which stage of the compiler to run, and
  which file to compile.
  Usage: ./toplevel.native [-a|-l|-s|-c|i] [file.sP]

  Written by: Team Nautilus (Ankur, Yuma, Max, Etha)
*)

type action = Ast | Sast | LLVM_IR | Compile | ImportStdlib

let () =
 let action = ref Compile in
 let set_action a () = action := a in
 let usage_msg = "Usage: ./toplevel.native [-a|-l|-s|-c] [file.sP]" in
 let speclist = [
   ("-a", Arg.Unit (set_action Ast), "Print the AST");
   ("-s", Arg.Unit (set_action Sast), "Print the SAST");
   ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
   ("-c", Arg.Unit (set_action Compile),
     "Check and print the generated LLVM IR (default)");
   ("-i", Arg.Unit (set_action ImportStdlib), "Import the standard library, and check
and print the generated LLVM IR");] in
 let channel = ref stdin in
   Arg.parse speclist (fun file -> channel := open_in file) usage_msg;
 let lexbuf = Lexing.from_channel !channel in
   match !action with
     Ast -> print_endline (Ast.ast_of_program (Parser.program Scanner.token lexbuf))
   | _   ->
       match !action with
```

```
        Sast    -> print_endline (Sast.sast_of_sprogram (Semant.check (Parser.program
Scanner.token lexbuf)))
      | LLVM_IR -> print_endline (Llvm.string_of_llmodule (Codegen.translate
(Semant.check (Parser.program Scanner.token lexbuf))))
      | Compile -> let m = Codegen.translate (Semant.check (Parser.program
Scanner.token lexbuf)) in Llvm_analysis.assert_valid_module m;
                   print_endline (Llvm.string_of_llmodule m)
      | ImportStdlib ->
        let stdlib_list = open_in "../stdlib/list.sP" in
        let stdlib_string = open_in "../stdlib/string.sP" in

        let temp_file_name = "__temp__.sP_" in
        let temp_file = open_out temp_file_name in

        (* copy contents of list.sP and string.sP to temp.sP *)
        let rec copy_file_to_temp_file file =
          try
            let line = input_line file in
            Printf.fprintf temp_file "%s\n" line;
            copy_file_to_temp_file file
          with End_of_file -> ()
        in
        copy_file_to_temp_file stdlib_list;
        copy_file_to_temp_file stdlib_string;

        let input_file_string = ref "" in
        let rec read_input_file file =
          try
            let line = input_line file in
            input_file_string := !input_file_string ^ line ^ "\n";
            read_input_file file
          with End_of_file -> ()
        in
        read_input_file !channel;

        (* write contents of input file to temp.sP *)
```

```
        Printf.fprintf temp_file "%s" !input_file_string;
        close_in !channel;
        close_in stdlib_list;
        close_in stdlib_string;
        close_out temp_file;

        let temp_lexbuf = Lexing.from_channel (open_in temp_file_name) in
        let m = Codegen.translate (Semant.check (Parser.program Scanner.token
temp_lexbuf)) in Llvm_analysis.assert_valid_module m;
        print_endline (Llvm.string_of_llmodule m);

        (* delete temp file *)
        Sys.remove temp_file_name;

      | _ -> raise (Failure "Internal error: no action selected")
```

8.1.2 scanner.mll

```
(* scanner.mll
  Scanner generator for the sPool programming language.

  Written by: Team Nautilus (Ankur, Yuma, Max, Etha)
*)

{ open Parser }

let digit = ['0' - '9']
let digits = digit+

rule token = parse
 [' ' '\t' '\r'] { token lexbuf   } (* ignore non-significant whitespace *)
| "#"             { comment lexbuf }
| '('             { LPAREN         }
| ')'             { RPAREN         }
| '['             { LSQUARE        }
| ']'             { RSQUARE        }
```

```
| '{'             { LBRACE         }
| '}'             { RBRACE         }
| ':'             { COLON          }
| ';'             { SEMI           }
| ','             { COMMA          }
| "->"            { ARROW          }
| "return"        { RETURN         }
| "shared"        { SHARED         }
| "int"           { INT            }
| "list"          { LIST           }
| "bool"          { BOOL           }
| "string"        { STRING         }
| "float"         { FLOAT          }
| "quack"         { QUACK          }
| "thread"        { THREAD         }
| "mutex"         { MUTEX          }
| "false"         { BLIT(false)    }
| "true"          { BLIT(true)     }
| "def"           { DEF            }
| "lambda"        { LAMBDA         }
| "store"         { STORE          }
| '"'(("\\"_)|[^'\n' '"'])*'"' as str { STRINGLIT(str) }
| "if"            { IF             }
| "else"          { ELSE           }
| "while"         { WHILE          }
| '\n'            { NEWLINE        }
| '='             { ASSIGN         }
| '+'             { PLUS           }
| '-'             { MINUS          }
| '*'             { TIMES          }
| '/'             { DIVIDE         }
| "=="            { EQ             }
| "!="            { NEQ            }
| '<'             { LT             }
| "<="            { LEQ            }
| ">"             { GT             }
```

```
| ">="            { GEQ            }
| "&&"            { AND            }
| "||"            { OR             }
| "!"             { NOT            }
| "%"             { MOD            }
| digits as lxm   { LITERAL(int_of_string lxm) }
| digits '.' digit* as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { NAME(lxm) }
| eof             { EOF            }
| _ as char       { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
 '\n'           { NEWLINE         }
| eof            { EOF             }
| _              { comment lexbuf  }
```

8.1.3 ast.ml

```
(* ast.ml
  Defines the abstract syntax tree (AST) of the sPool programming language,
  along with functions for pretty-printing the AST.

  Written by: Team Nautilus (Ankur, Yuma, Max, Etha)
*)

type unaryop = Not | Neg
type binop   = And | Or | Add | Sub | Mod | Mult | Div | Equal | Neq | Less
            | Leq | Greater | Geq

type typ = Quack | Int | Bool | Float | Mutex | Thread | String
        | Arrow of typ list * typ
        | List of typ
        | Alpha (* Used internally for polymorphism. Quack is to null as Alpha is to
void pointer *)

type bind = typ * string
```

```
type expr =
 Literal of int
| BoolLit of bool
| ListLit of expr list
| Fliteral of string
| StringLiteral of string
| Thread of statement list
| Var of string
| Binop of expr * binop * expr
| Unop of unaryop * expr
| Lambda of bool * typ * bind list * statement list
| Call of string * expr list
| Noexpr
and statement =
 Expr of expr
| Assign of string * expr
| Define of bool * typ * string * expr (* first bool indicates whether the variable is
shared across threads *)
| If of expr * statement list * statement list
| While of expr * statement list
| FunDef of bool * typ * string * bind list * statement list (* first bool indicates
whether store is present *)
| Return of expr

type program = Program of statement list

(* Pretty-printing functions *)

let rec string_of_bindings = function
 []         -> ""
| [(t, x)]   -> string_of_type t ^ " " ^ x
| (t, x)::xs -> string_of_type t ^ " " ^ x ^ ", " ^ string_of_bindings xs
and string_of_expr = function
 Literal(l)          -> string_of_int l
| Fliteral(f)         -> f
```

```
| BoolLit(b)          -> string_of_bool b
| StringLiteral(s)    -> s
| Thread(_)           -> "{...}"
| Var(s)              -> s
| Binop(e1, o, e2)    -> string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^
string_of_expr e2
| Unop(o, e)          -> string_of_uop o ^ string_of_expr e
| Lambda(_, t, bs, _) -> "lambda " ^ string_of_type t ^ " (" ^ string_of_bindings bs ^
"): " ^ " ... " ^ ";"
| Call(name, args)    -> name ^ "(" ^ (List.fold_left (fun acc e ->  (if acc = "" then
string_of_expr e else acc ^ ", " ^ string_of_expr e)) "" args) ^ ")"
| ListLit(es)         -> "[" ^ (List.fold_left (fun acc e ->  (if acc = "" then
string_of_expr e else acc ^ ", " ^ string_of_expr e)) "" es) ^ "]"
| Noexpr              -> ""
and string_of_type = function
 Int          -> "int"
| Bool         -> "bool"
| Quack        -> "quack"
| Float        -> "float"
| Mutex        -> "mutex"
| Thread       -> "thread"
| String       -> "string"
| Arrow(ts, t) -> (List.fold_left (fun acc t -> (if acc = "" then acc else acc ^ ", ")
^ string_of_type t) "" ts) ^ " -> " ^ string_of_type t ^ ")"
| List(t)      -> "list<" ^ string_of_type t ^ ">"
| Alpha        -> "alpha"
and string_of_op = function
 Add          -> "+"
| Sub          -> "-"
| Mult         -> "*"
| Mod          -> "%"
| Div          -> "/"
| Equal        -> "=="
| Neq          -> "!="
| Less         -> "<"
| Leq          -> "<="
```

```
| Greater      -> ">"
| Geq          -> ">="
| And          -> "&&"
| Or           -> "||"
and string_of_uop = function
 Neg          -> "-"
| Not          -> "!"

let rec ast_of_op = function
 Add          -> "PLUS"
| Sub          -> "MINUS"
| Mult         -> "TIMES"
| Mod          -> "MOD"
| Div          -> "DIV"
| Equal        -> "EQUALS"
| Neq          -> "NOTEQUALS"
| Less         -> "LESS"
| Leq          -> "LEQ"
| Greater      -> "GREATER"
| Geq          -> "GEQ"
| And          -> "AND"
| Or           -> "OR"
and ast_of_uop = function
 Neg          -> "NEG"
| Not          -> "NOT"
and ast_of_ty = function
 Int          -> "INTTY"
| Bool         -> "BOOLTY"
| Quack        -> "QUACKTY"
| Float        -> "FLOATTY"
| Mutex        -> "MUTEXTY"
| Thread       -> "THREADTY"
| String       -> "STRINGTY"
| Arrow(ts, t) -> "ARROWTY(" ^ (List.fold_left (fun acc t -> (if acc = "" then acc
else acc ^ " * ") ^ ast_of_ty t) "" ts) ^ " -> " ^ ast_of_ty t ^ ")"
| List(t)      -> "LISTTY(" ^ ast_of_ty t ^ ")"
```

```
| Alpha        -> "ALPHATY"
and ast_of_bindings = function
 []           -> ""
| [(t, x)]     ->  "(" ^ ast_of_ty t ^ ", " ^ x ^ ")"
| (t, x)::xs   -> "(" ^ ast_of_ty t ^ ", " ^ x ^ "), " ^ ast_of_bindings xs
and ast_of_expr n = function
 Literal(l)   -> "INT(" ^ string_of_int l ^ ")"
| Fliteral(f)  -> "FLOAT(" ^ f ^ ")"
| BoolLit(b)   -> "BOOL(" ^ string_of_bool b ^ ")"
| Thread(s)    -> "THREAD(" ^ (ast_of_s_list (n + 1) s) ^ ")"
| Var(s)       -> "VAR(" ^ s ^ ")"
| Unop(o, e)   -> "UNOP(" ^ ast_of_uop o ^ ", " ^ ast_of_expr n e ^ ")"
| ListLit(es)  -> "LIST(" ^ List.fold_left (fun acc ex -> (if acc = "" then acc else
acc ^ ", ") ^ ast_of_expr n ex) "" es ^ ")"
| Noexpr       -> "NOEXPR"
| StringLiteral(s)    -> "STRING(" ^ s ^ ")"
| Lambda(_, t, bs, s) -> "LAMBDA(" ^ ast_of_ty t ^ ", " ^ "FORMALS(" ^ ast_of_bindings
bs ^ "), " ^ ast_of_s_list (n + 1) s ^ ")"
| Call(name, args)    -> "CALL(" ^ name ^ "," ^ " ARGS(" ^ List.fold_left (fun acc ex
-> (if acc = "" then acc else acc ^ ", ") ^ ast_of_expr n ex) "" args ^ "))"
| Binop(e1, o, e2)    -> "BINOP(" ^ ast_of_expr n e1 ^ ", " ^ ast_of_op o ^ ", " ^
ast_of_expr n e2 ^ ")"
and ast_of_s_list n s = "[" ^ (List.fold_left (fun acc st -> acc ^ " " ^
ast_of_statement n st) "" s) ^ "]"
and ast_of_statement n statement =
 let statement_str =
   match statement with
     Expr(e)           -> ast_of_expr n e
   | Assign(v, e)      -> "ASSIGN(" ^ v ^ ", " ^ ast_of_expr n e ^ ")"
   | Define(s, t, v, e) -> "DEFINE(" ^ string_of_bool s ^ ", " ^ ast_of_ty t ^ ", " ^
v ^ ", " ^ ast_of_expr n e ^ ")"
   | Return(e)         -> "RETURN(" ^ ast_of_expr n e ^ ")"
   | If(e, s1, s2)     -> "IF(" ^ ast_of_expr n e ^ ", " ^ ast_of_s_list (n + 1) s1 ^
", " ^ ast_of_s_list (n + 1) s2 ^ ")"
   | While(e, s)       -> "WHILE(" ^ ast_of_expr n e ^ ", " ^ ast_of_s_list (n + 1 )
s ^ ")"
```

```
   | FunDef(s, t, fname, fs, b) ->
       "FUN(" ^ (if s then "STORE, " else "NOSTORE, ")
       ^ ast_of_ty t ^ ", " ^ fname ^ ", " ^
       "FORMALS(" ^ ast_of_bindings fs ^ "), " ^ ast_of_s_list (n + 1) b ^ ")"
 in "\n" ^ n_tabs n ^ statement_str
and n_tabs n = if n = 0 then "" else "    " ^ (n_tabs (n - 1))

let ast_of_program (Program(s)) = "PROGRAM {" ^ ast_of_s_list 1 s  ^ "}"
```

8.1.4 parser.mly

```
// parser.mly
// Parser generator for the sPool programming language.
//
// Written by: Team Nautilus (Ankur, Yuma, Max, Etha)

%{
open Ast
%}

%token PLUS MINUS TIMES DIVIDE NOT EQ NEQ LT LEQ GT GEQ AND OR MOD
%token ASSIGN ELSE IF WHILE
%token INT BOOL FLOAT QUACK MUTEX THREAD STRING LIST ARROW
%token DEF STORE RETURN LAMBDA SHARED
%token NEWLINE LPAREN RPAREN LBRACE RBRACE COLON SEMI COMMA LSQUARE RSQUARE EOF
%token <int> LITERAL
%token <bool> BLIT
%token <string> NAME
%token <string> STRINGLIT
%token <string> FLIT

%start program
%type <Ast.program> program

%nonassoc ELSE
%right ASSIGN
%left OR
```

```
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT

%%

program:
 d_opt statement_list EOF            { Program($2) }

d_opt:
 /* nothing */                      {}
| delimiter                          {}

store_opt:
 /* nothing */                      { false }
| STORE                              { true  }

args_opt:
 /* nothing */                      { []          }
| args_list                          { List.rev $1 }

args_list:
 expr                               { [$1]     }
| args_list COMMA expr               { $3 :: $1 }

formals_opt:
 /* nothing */                      { []          }
| formal_list                        { List.rev $1 }

formal_list:
 typ NAME                           { [($1,$2)]     }
| formal_list COMMA typ NAME         { ($3,$4) :: $1 }
```

```
delimiter:
 NEWLINE                              {}
| delimiter NEWLINE                    {}

typ_list:
 typ                                  { [$1]     }
| typ_list COMMA typ                   { $3 :: $1 }

typ:
 INT                                  { Int                    }
| BOOL                                 { Bool                   }
| QUACK                                { Quack                  }
| FLOAT                                { Float                  }
| STRING                               { String                 }
| MUTEX                                { Mutex                  }
| THREAD                               { Thread                 }
| LIST LT typ GT                       { List($3)               }
| LPAREN typ_list ARROW typ RPAREN     { Arrow(List.rev $2, $4) }

expr_opt:
 /* nothing */                        { Noexpr }
| expr                                 { $1     }

statement_list:
/* nothing */                          { []       }
| statement delimiter statement_list   { $1 :: $3 }
| statement                            { [$1]     }

statement:
 expr                                 { Expr($1)           }
| typ NAME ASSIGN expr                 { Define(false, $1, $2, $4) }
| SHARED typ NAME ASSIGN expr          { Define(true, $2, $3, $5)  }
| NAME ASSIGN expr                     { Assign($1, $3)     }
| RETURN expr_opt                      { Return($2)         }
| DEF store_opt typ NAME LPAREN formals_opt RPAREN COLON d_opt statement_list SEMI
                                       { FunDef($2, $3, $4, $6, $10) }
```

```
| WHILE LPAREN expr RPAREN COLON d_opt statement_list SEMI { While($3, $7)   }
| IF LPAREN expr RPAREN COLON d_opt statement_list SEMI    { If($3, $7, [])  }
| IF LPAREN expr RPAREN COLON d_opt statement_list ELSE d_opt statement_list SEMI
                                                          { If($3, $7, $10) }

expr:
 FLIT                               {      Fliteral($1)      }
| BLIT                               {      BoolLit($1)       }
| LITERAL                            {      Literal($1)       }
| STRINGLIT                          {      StringLiteral($1) }
| NAME                               {      Var($1)           }
| LSQUARE args_opt RSQUARE           {      ListLit($2)       }
| LBRACE d_opt statement_list RBRACE  {      Thread($3)        }
| NOT expr                           {      Unop(Not, $2)     }
| MINUS expr %prec NOT               {      Unop(Neg, $2)     }
| NAME LPAREN args_opt RPAREN        {      Call($1, $3)      }
| LPAREN expr RPAREN                 {      $2                }
| expr PLUS   expr                   { Binop($1, Add,   $3)   }
| expr MINUS  expr                   { Binop($1, Sub,   $3)   }
| expr MOD    expr                   { Binop($1, Mod,   $3)   }
| expr TIMES  expr                   { Binop($1, Mult,  $3)   }
| expr DIVIDE expr                   { Binop($1, Div,   $3)   }
| expr EQ     expr                   { Binop($1, Equal, $3)   }
| expr NEQ    expr                   { Binop($1, Neq,   $3)   }
| expr LT     expr                   { Binop($1, Less,  $3)   }
| expr LEQ    expr                   { Binop($1, Leq,   $3)   }
| expr GT     expr                   { Binop($1, Greater, $3) }
| expr GEQ    expr                   { Binop($1, Geq,   $3)   }
| expr AND    expr                   { Binop($1, And,   $3)   }
| expr OR     expr                   { Binop($1, Or,    $3)   }
| LAMBDA typ LPAREN formals_opt RPAREN COLON d_opt statement_list SEMI { Lambda(false,
$2, $4, $8) }
```

8.1.5 sast.ml

```
(* sast.ml
  Defines the semantically-checked abstract syntax tree (SAST) of the
```

```
  sPool programming language, along with functions for pretty-printing the SAST.

  Written by: Team Nautilus (Ankur, Yuma, Max, Etha)
*)

open Ast

type sexpr = typ * sx
and sx =
 SLiteral of int
| SBoolLit of bool
| SListLit of sexpr list
| SFliteral of string
| SStringLiteral of string
| SThread of sstatement list
| SVar of bool * string
| SBinop of sexpr * binop * sexpr
| SUnop of unaryop * sexpr
| SLambda of bool * typ * bind list * sstatement list
| SCall of string * sexpr list
| SNoexpr
and sstatement =
 SExpr of sexpr
| SAssign of string * sexpr
| SDefine of bool * typ * string * sexpr
| SIf of sexpr * sstatement list * sstatement list
| SWhile of sexpr * sstatement list
| SReturn of sexpr

type sprogram = SProgram of sstatement list

(* Pretty-printing functions *)

let shared_str s = if s then "shared " else ""

let rec sast_of_sexpr n (t, e) =
```

```
 "(" ^ ast_of_ty t ^ " : " ^
 (match e with
   SLiteral(l)              -> "SINT(" ^ string_of_int l ^ ")"
 | SFliteral(f)             -> "SFLOAT(" ^ f ^ ")"
 | SBoolLit(b)              -> "SBOOL(" ^ string_of_bool b ^ ")"
 | SStringLiteral(s)        -> "SSTRING(" ^ s ^ ")"
 | SThread(s)               -> "STHREAD(" ^ (sast_of_s_list (n + 1) s) ^ ")"
 | SVar(shared, s)          -> "SVAR(" ^ shared_str shared ^ ast_of_ty t ^ ", " ^ s ^
")"
 | SBinop(e1, o, e2)        -> "SBINOP(" ^ sast_of_sexpr n e1 ^ ", " ^ ast_of_op o ^
", " ^ sast_of_sexpr n e2 ^ ")"
 | SUnop(o, e)              -> "SUNOP(" ^ ast_of_uop o ^ ", " ^ sast_of_sexpr n e ^
")"
 | SLambda(store, t1, bs, s) -> "SLAMBDA(STORE(" ^ string_of_bool store ^ "), " ^
ast_of_ty t1 ^ ", " ^ "FORMALS(" ^ ast_of_bindings bs ^ "), " ^ sast_of_s_list (n + 1)
s ^ ")"
 | SCall(name, args)        -> "SCALL(" ^ name ^ "," ^ " ARGS(" ^ List.fold_left (fun
acc ex -> (if acc = "" then acc else acc ^ ", ") ^ sast_of_sexpr n ex) "" args ^ "))"
 | SListLit(es)             -> "SLIST(" ^ List.fold_left (fun acc ex -> (if acc = ""
then acc else acc ^ ", ") ^ sast_of_sexpr n ex) "" es ^ ")"
 | SNoexpr                  -> "SNOEXPR") ^ ")"
and sast_of_s_list n s = "[" ^ (List.fold_left (fun acc st -> acc ^ " " ^
sast_of_sstatement n st) "" s) ^ "]"
and sast_of_sstatement n statement =
 let statement_str =
   match statement with
     SExpr(e)           -> sast_of_sexpr n e
   | SAssign(v, e)      -> "SASSIGN(" ^ v ^ ", " ^ sast_of_sexpr n e ^ ")"
   | SDefine(s, t, v, e) -> "SDEFINE(" ^ shared_str s ^ ast_of_ty t ^ ", " ^ v ^ ", "
^ sast_of_sexpr n e ^ ")"
   | SReturn(e)         -> "SRETURN(" ^ sast_of_sexpr n e ^ ")"
   | SIf(e, s1, s2)     -> "SIF(" ^ sast_of_sexpr n e ^ ", " ^ sast_of_s_list (n + 1)
s1 ^ ", " ^ sast_of_s_list (n + 1) s2 ^ ")"
   | SWhile(e, s)       -> "SWHILE(" ^ sast_of_sexpr n e ^ ", " ^ sast_of_s_list (n +
1 ) s ^ ")"
 in "\n" ^ Ast.n_tabs n ^ statement_str
```

```
 let sast_of_sprogram (SProgram(s)) = "SPROGRAM {" ^ sast_of_s_list 1 s ^ "}"
```

8.1.6 semant.ml

```
(* semant.ml
  Walks over the AST generated by the sPool parser and generates a
  semantically-checked AST (SAST) with extra information added to each node
  of the AST.

  Written by: Team Nautilus (Ankur, Yuma, Max, Etha)
*)

open Ast
open Sast

exception SemanticError of string
exception NameNotFound of string
exception TypeError of string

module StringMap = Map.Make(String)

type symbol_table = {
 (* Variables bound in current scope *)
 variables : typ StringMap.t;
  (* For each variable bound in current scope, are they shared across thread? *)
 shared : bool StringMap.t;

 (* Enclosing scope *)
 parent : symbol_table option;
}

let rec find_variable (scope : symbol_table) name =
 try
   StringMap.find name scope.variables
 with Not_found ->
   match scope.parent with
     Some(parent) -> find_variable parent name
```

```
   | _            -> raise (NameNotFound ("unidentified flying name " ^ name))

let rec find_shared (scope : symbol_table) name =
 try
   StringMap.find name scope.shared
 with Not_found ->
   match scope.parent with
     Some(parent) -> find_shared parent name
   | _            -> raise (NameNotFound ("unidentified flying name " ^ name))

type translation_environment = { scope : symbol_table }

(* initial env *)
let env : translation_environment ref = ref { scope = { variables = StringMap.empty;
shared = StringMap.empty; parent = None } }

let add_to_scope (s, t, n) =
 let new_scope = {variables = StringMap.add n t !env.scope.variables; shared =
StringMap.add n s !env.scope.shared; parent = !env.scope.parent}
 in env := {scope = new_scope}

let rec eqType = function
   (Arrow(ts, t), Arrow(ts2, t2)) ->
     let sametype = eqType(t, t2) in
     if (List.length ts) != (List.length ts2) then false else
     let zipped_ts = List.combine ts ts2 in
     let sametypes = List.for_all eqType zipped_ts in
       sametype && sametypes
 | (List(t1), List(t2)) -> t1 = Alpha || t2 = Alpha || eqType(t1, t2)
 | (Alpha, _) | (_, Alpha) -> true
 | (List(_), _) | (_, List(_)) | (Arrow(_, _), _) | (_, Arrow(_, _)) -> false
 | (t1, t2) -> t1 = t2  (* primitive type equality *)

(* list of built-in function names *)
let builtin_functions =
 let builtins = [
```

```
                            (* List built-ins *)
                ("List_len", Arrow([List(Alpha)], Int)); ("List_at",
Arrow([List(Alpha); Int], Alpha));
                ("List_replace", Arrow([List(Alpha); Int; Alpha], Quack));
                ("List_insert", Arrow([List(Alpha); Int; Alpha], Quack));
                ("List_remove", Arrow([List(Alpha); Int], Quack));

                            (* Printing built-ins *)
                ("print", Arrow([String], Quack)); ("println", Arrow([String],
Quack));

                            (* Type conversion built-ins *)
                ("int_to_string", Arrow([Int], String)); ("float_to_string",
Arrow([Float], String));
                ("bool_to_string", Arrow([Bool], String)); ("int_to_float",
Arrow([Int], Float));
                ("float_to_int", Arrow([Float], Int));

                            (* String built-ins *)
                ("String_len", Arrow([String], Int)); ("String_concat",
Arrow([String; String], String));
                ("String_substr", Arrow([String; Int; Int], String)); ("String_eq",
Arrow([String; String], Bool));

                            (* Mutex built-ins *)
                ("Mutex", Arrow([Quack], Mutex)); ("Mutex_lock", Arrow([Mutex],
Quack));
                ("Mutex_unlock", Arrow([Mutex], Quack));

                            (* Thread built-ins *)
                ("Thread_join", Arrow([Thread], Quack));

                ] in
 let _ = List.iter (fun (n, t) -> add_to_scope(false, t, n)) builtins in
 List.map fst builtins
```

```
(* returns the specific sPool type that should replace ALPHA for each list builtin
function *)
let alpha_to_builtin (fname, argtypes) = match (fname, argtypes) with
   ("List_at", [List(t); Int])                              -> t
 | ("List_len", [List(t)])                                  -> t
 | ("List_replace", [List(t); Int; t2]) when eqType(t, t2) -> t
 | ("List_insert", [List(t); Int; t2])  when eqType(t, t2) -> t
 | ("List_remove", [List(t); Int])                          -> t
 | _                                                        -> raise (Failure
("InternalError: Mistyped arguments to list builtin function " ^ fname))

let check (Program(statements)) =
 let rec eqTypes = function
     []             -> true
   | [_]            -> true
   | t1 :: t2 :: ts -> (eqType(t1, t2)) && (eqTypes (t2 :: ts)) in
 let rec quack_type = function
     Quack    -> true
   | List(t) -> quack_type t
   | _        -> false in
 let rec alpha_type = function
     Alpha         -> true
   | Arrow(ts, t) -> (List.exists alpha_type ts) || (alpha_type t)
   | List(t)      -> alpha_type t
   | _            -> false in
 let push_scope () =
   let new_scope = {variables = StringMap.empty; shared = StringMap.empty; parent =
Some(!env.scope)}
   in env := {scope = new_scope}
 in
 let pop_scope () =
   let parent_scope = match !env.scope.parent with
       Some(parent) -> parent
     | _            -> raise (Failure "Internal Error: no parent scope")
   in env := {scope = parent_scope}
 in
```

```
 let str_of_scope scope =
   let str_of_bindings bindings = String.concat ", " (List.map (fun (k, v) ->
(string_of_type v) ^ " : " ^ k) (StringMap.bindings bindings))
   in "{" ^ str_of_bindings scope.variables ^ "}"
 in
 let defined_in_current_scope s = StringMap.mem s !env.scope.variables in
 let rec check_bool_expr e =
   let (t, se) = check_expr e in
   let err = "expected " ^ string_of_expr e ^ " to be of type bool, but it is type " ^
string_of_type t ^ " instead"
   in if t != Bool then raise (TypeError err) else (t, se)
 and construct_arrow_type formals t =
   let ts = if formals = [] then [Quack] else List.map fst formals in Arrow(ts, t)
 and is_function = function
   | Arrow(_, _) -> true | _ -> false
 and check_statement = function
     Expr(expr)               -> SExpr (check_expr expr)
   | Define(s, t, name, expr) ->
       let _ = if List.mem name builtin_functions then raise (SemanticError ("name " ^
name ^ " is a built-in function and may not be redefined")) else () in
       let _ = if defined_in_current_scope name then raise (SemanticError ("name " ^
name ^ " is already defined in the current scope and may not be redefined in this
scope")) else () in
       let _ = if is_function t then let _ = add_to_scope(false, t, name) in () else
() in (* Support recursive calls *)
       (match check_expr expr with
         | _       when quack_type t -> raise (TypeError ("variables of type quack may
not be defined"))
         | (t1, _) when not (eqType(t1, t))  -> raise (TypeError ("expression " ^
string_of_expr expr ^ " of type " ^ string_of_type t1 ^ " may not be assigned to a
variable of type " ^ string_of_type t))
         | (t1, _) as sexpr ->
             let _ = (match t1 with
                 Arrow(_, _) | Thread when s -> raise (TypeError ("variables of type "
^ string_of_type t1 ^ " may not be defined as shared variables"))
               | _ -> ())
```

```
            in
            let is_shared = (match t with List(_) | Mutex -> true
                                          | _              -> s)
            in
            let _ = if not (is_function t) then add_to_scope (is_shared, t, name)
else ()
            in SDefine(is_shared, t, name, sexpr))
   | Assign(name, expr) ->
       let (t1, _) as sexpr = check_expr expr in
           (match find_variable !env.scope name with
              t when eqType (t1, t) -> SAssign(name, sexpr)
            | t -> raise (TypeError ("expression " ^ string_of_expr expr ^ " of type
" ^ string_of_type t1 ^ " may not be assigned to a variable of type " ^ string_of_type
t)))
   | While(expr, statements) ->
       let conditional = check_bool_expr expr in
       let _ = push_scope () in
       let sexpr = SWhile(conditional, List.map check_statement statements) in
       let _ = pop_scope () in sexpr
   | Return(_) -> raise (SemanticError "Return statement may not exist outside of a
function definition")
   | If(expr, statements1, statements2) ->
       let conditional = check_bool_expr expr in
       let _ = push_scope () in
       let ss1 = List.map check_statement statements1 in
       let _ = pop_scope () in
       let _ = push_scope () in
       let ss2 = List.map check_statement statements2 in
       let _ = pop_scope () in SIf(conditional, ss1, ss2)
   | FunDef(store, t, name, formals, statements) ->
       let arrow_ty = construct_arrow_type formals t in
       check_statement(Define(false, arrow_ty, name, Lambda(store, t, formals,
statements)))
 (* sanitize_string: remove first and last quotes, and remove any backslashes before
double quote or newlines *)
 and sanitize_string s =
```

```
  let len = String.length s in
  let rec sanitize i =
    if i = len then "" else
      let c = String.get s i in
      let c' =
        if c = '\\' then
          if i + 1 < len then
            let c'' = String.get s (i + 1) in
            if c'' = '"' || c'' = '\n'
              then "" else String.make 1 c
          else String.make 1 c
        else String.make 1 c
      in c' ^ sanitize (i + 1)
  in let result = sanitize 1 in String.sub result 0 (String.length result - 1)
and check_expr = function
    Literal(l)          -> (Int, SLiteral l)
  | BoolLit(b)          -> (Bool, SBoolLit(b))
  | Fliteral(f)         -> (Float, SFliteral(f))
  | StringLiteral(s)    -> (String, SStringLiteral(sanitize_string(s)))
  | Thread(statements)  ->
      let _ = push_scope () in
      let sts = List.map check_statement statements in
      let _ = pop_scope () in (Thread, SThread(sts))
  | ListLit(l) as expr  -> (match l with
        [] -> (List(Alpha), SListLit([]))
      | xs ->
        let sxs = List.map check_expr xs in
        let ts = List.map fst sxs in
          if eqTypes ts then (List(List.hd ts), SListLit(sxs))
          else raise (TypeError ("lists must only contain expressions of the same
type in expression: " ^ string_of_expr expr)))
  | Var(s)              -> (find_variable !env.scope s, SVar(find_shared !env.scope
s, s))
  | Unop(op, e) as expr ->
      let (t, _) as sexpr = check_expr e in
        (let ty = match op with
```

```
            Neg when (t = Int || t = Float) -> t
          | Not when t = Bool               -> Bool
          | _                               -> raise (TypeError ("illegal unary
operator " ^ string_of_uop op ^ " on type " ^ string_of_type t ^ " in expression: " ^
string_of_expr expr))
        in (ty, SUnop(op, sexpr)))
   | Binop(e1, op, e2) as expr ->
      let (t1, _) as sexpr1 = check_expr e1 in
      let (t2, _) as sexpr2 = check_expr e2 in
        if not (eqType(t1, t2)) then raise (TypeError ("binary operator " ^
string_of_op op ^ " must get identical types, not " ^ string_of_type t1 ^ " and " ^
string_of_type t2 ^ " in expression: " ^ string_of_expr expr)) else
          (let ty = match op with
            | Add | Sub | Mult | Div     when (t1 = Int || t1 = Float) -> t1
            | Mod                        when t1 = Int                -> Int
            | And | Or                   when t1 = Bool               -> Bool
            | Geq | Greater | Leq | Less when (t1 = Int || t1 = Float) -> Bool
            | Neq | Equal                when (t1 = Int || t1 = Bool || t1 = Float)
-> Bool
            | _ -> raise (TypeError ("illegal binary operator " ^ string_of_op op ^ "
between types " ^ string_of_type t1 ^ " and " ^ string_of_type t2 ^ " in expression: "
^ string_of_expr expr))
          in (ty, SBinop(sexpr1, op, sexpr2)))
   | Lambda(store, t, formals, statements) ->
      let count_return s =
        let rec count_return' s acc = match s with
            [] -> acc
          | Return(_) :: xs -> count_return' xs (acc + 1)
          | If(_, s1, s2) :: xs -> count_return' (s1 @ s2 @ xs) acc (* Only count
Return in IF or WHILE inside a function body *)
          | While(_, s) :: xs -> count_return' (s @ xs) acc
          | _ :: xs -> count_return' xs acc (* Other constructs allow returns in them
(nested functions) *)
        in count_return' s 0 in
      let num_returns = count_return statements in
```

```
        if not (num_returns = 1) then raise (SemanticError ("Function body must have
exactly 1 return statement, not " ^ string_of_int num_returns)) else
      let is_shared t = match t with Mutex | List(_) -> true | _ -> false in (* only
lists and mutexes are pointers; they are allocated on the heap *)
      let _ = push_scope () in
      let _ = List.map (fun (ft, fn) -> add_to_scope (is_shared ft, ft, fn)) formals
in
      (* check whether return statement is the last statement in function body *)
      let rec check_body xs acc = match xs with
          [Return(e)] -> let (t', _) as sx = check_expr e in
                               if (eqType (t', t)) then SLambda(store, t, formals,
(List.rev acc) @ [SReturn(sx)])
                                                   else raise (TypeError "Expression
in return statement must match type declared in function")
        | [_] -> raise (SemanticError "No statement(s) can follow a return
statement")
        | x :: xs_rest -> check_body xs_rest (check_statement x :: acc)
        | _ -> raise (Failure "InternalError: shouldn't happen in body")
      in let slam = check_body statements [] in
      let _ = pop_scope () in (construct_arrow_type formals t, slam)
   | Call(name, actuals) ->
      let funty = (match find_variable !env.scope name with
          Arrow(_, _) as t -> t
        | _ -> raise (TypeError ("name " ^ name ^ " is not a function and is
therefore not callable"))) in
      let rec num_args = function
          [Quack] | [] -> 0
        | _ :: xs      -> 1 + num_args xs in
      let (fty, retty) = match funty with Arrow(f, r) -> (f, r) | _ -> raise (Failure
"InternalError: shouldn't happen in call") in
      if List.length actuals != num_args fty then raise (SemanticError ("Function " ^
name ^ " called with the wrong number of arguments")) else
      let checked_actuals = List.map check_expr actuals in
      let arg_typs =
        let ca = List.map fst checked_actuals in
```

```
        if ca = [] then [Quack] else ca in (* If the arg list is empty, set it to
have 'quack' type *)

      (* if alphas exist in the function's return type, instantiate it so we get a
monomorphic type for the result of the call *)
        let sretty = if alpha_type retty then alpha_to_builtin (name, arg_typs) else
retty in
        let _ =
          (* Check well-typedness of polymorphic functions. This is done by
alpha_to_builtin function automatically during pattern matching. This check ensures
that calls like List_insert([1, 2], 0, false) are rejected because false != int *)
          if alpha_type funty then alpha_to_builtin (name, arg_typs) else Quack in
      if eqType (funty, Arrow(arg_typs, retty)) then (sretty, SCall(name,
checked_actuals))
      else raise (TypeError ("Function " ^ name ^ " called with the wrong argument
types"))
  | Noexpr -> (Quack, SNoexpr)
 in
  let _ = str_of_scope !env.scope in (* silence compiler warning for debugging
functions *)
  if statements = [] then SProgram([])
  else SProgram(List.map check_statement statements)
```

8.1.7 codegen.ml

```
(* codegen.ml
  Translates a semantically-checked AST to LLVM IR.

  Written by: Team Nautilus (Ankur, Yuma, Max, Etha)
*)

module L = Llvm
module A = Ast
module S = Stack
open Sast

module StringMap = Map.Make(String)
```

```
(* context and type defintions *)
let context    = L.global_context ()
let i32_t      = L.i32_type    context
and i8_t       = L.i8_type     context
and i1_t       = L.i1_type     context
and float_t    = L.double_type context
and quack_t    = L.void_type   context
and string_t   = L.pointer_type (L.i8_type context)
and voidptr_t  = L.pointer_type (L.i8_type context)
and nodeptr_t  = L.pointer_type (L.named_struct_type context "Node")
let list_t     = L.pointer_type (L.struct_type context [| voidptr_t; nodeptr_t |])
and pthread_t  = L.pointer_type (L.named_struct_type context "pthread_t")
and mutex_t    = L.pointer_type (L.named_struct_type context "pthread_mutex_t")
let thread_t   = pthread_t

let is_pointer = function
 A.List(_) | A.Mutex -> true
| _          -> false

let is_mutex = function
 A.Mutex -> true | _ -> false

let rec ltype_of_typ = function
 A.Int     -> i32_t
| A.Bool    -> i1_t
| A.Float   -> float_t
| A.Quack   -> quack_t
| A.String  -> string_t
| A.List(_) -> list_t
| A.Thread  -> thread_t
| A.Mutex   -> mutex_t
| A.Arrow(ts, t) -> ftype_from_t (A.Arrow(ts, t))
| A.Alpha   -> raise (Semant.SemanticError "Can not convert Alpha to lltype in
codegen")
and ftype_from_t = function
```

```
 A.Arrow(formals, retty) ->
   let formal_types = Array.of_list (List.map (fun (t) -> if ((is_pointer t && (not
(is_mutex t))) || (is_function t)) then L.pointer_type (ltype_of_typ t) else
ltype_of_typ t) formals) in
   let formal_types = Array.of_list (List.filter (fun (t) -> t <> quack_t)
(Array.to_list formal_types)) in
   let ret_llval =
     if ((is_function retty) || (is_pointer retty && (not (is_mutex retty)))) then
(L.pointer_type (ltype_of_typ retty)) else ltype_of_typ retty in L.function_type
ret_llval formal_types
| _ -> raise (Failure "Internal Error: ftype_from_t called on non-arrow type")
and is_function = function A.Arrow(_, _) -> true | _ -> false

let is_llval_pointer llval = (L.type_of llval = (L.pointer_type (L.pointer_type
list_t)))

type symbol_table = {
 variables : L.llvalue StringMap.t;
 functionpointers : L.llvalue StringMap.t;
 shared : bool StringMap.t;
 stored : (L.llvalue option) StringMap.t;
 parent : symbol_table option;
}

let rec find_variable (scope : symbol_table) name =
 try
   StringMap.find name scope.variables
 with Not_found ->
   match scope.parent with
     Some(parent) -> find_variable parent name
   | _            -> raise (Failure ("Internal Error: should have been caught in
semantic analysis (find_variable)"))

let rec find_shared (scope : symbol_table) name =
 try
   StringMap.find name scope.shared
```

```
 with Not_found ->
   match scope.parent with
     Some(parent) -> find_shared parent name
   | _            -> raise (Failure ("Internal Error: should have been caught in
semantic analysis (find_shared)"))

let rec find_stored (scope : symbol_table) name =
   try
     StringMap.find name scope.stored
   with Not_found ->
     match scope.parent with
       Some(parent) -> find_stored parent name
     | _            -> None

let env : symbol_table ref = ref { variables = StringMap.empty; shared =
StringMap.empty; stored = StringMap.empty; parent = None ; functionpointers =
StringMap.empty }
let seen_functions = ref StringMap.empty
let anon_counter = ref 1
let depth = ref 0

let add_to_scope (s, v, n) builder t =
 if not s then
   let local = L.build_alloca (ltype_of_typ t) n builder in
   let _     = L.build_store v local builder in
   let new_scope = {variables = StringMap.add n local !env.variables; shared =
StringMap.add n s !env.shared; stored = !env.stored; parent = !env.parent;
functionpointers = !env.functionpointers}
     in env := new_scope
 else
   if is_pointer t then (* this is for values that are passed by reference (aka raw
pointers) *)
     if is_mutex t then
       let local = L.build_alloca (ltype_of_typ t) n builder in
       let _     = L.build_store v local builder in
```

```
      let new_scope = {variables = StringMap.add n local !env.variables; shared =
StringMap.add n s !env.shared; stored = !env.stored; parent = !env.parent;
functionpointers = !env.functionpointers}
        in env := new_scope
    else
      let local = L.build_alloca (L.pointer_type (ltype_of_typ t)) n builder in
      let _     = L.build_store v local builder in
      let new_scope = {variables = StringMap.add n local !env.variables; shared =
StringMap.add n s !env.shared; stored = !env.stored; parent = !env.parent;
functionpointers = !env.functionpointers}
        in env := new_scope
  else
    let heap  = L.build_malloc (ltype_of_typ t) n builder in
    let _     = L.build_store v heap builder in
    let new_scope = {variables = StringMap.add n heap !env.variables; shared =
StringMap.add n s !env.shared; stored = !env.stored; parent = !env.parent;
functionpointers = !env.functionpointers}
      in env := new_scope

let push_scope () =
 let new_scope = {variables = StringMap.empty; shared = StringMap.empty; stored =
StringMap.empty; parent = Some(!env); functionpointers = StringMap.empty}
 in let _ = depth := !depth + 1 in env := new_scope

let pop_scope () =
 let parent_scope = match !env.parent with
    Some(parent) -> parent
  | _            -> raise (Failure "Internal Error: should not happen in pop_scope;
should have been caught in semantic analysis")
 in let _ = depth := !depth - 1 in env := parent_scope

let translate (SProgram(statements)) =
 let the_module = L.create_module context "sPool" in

 let main_t                = L.function_type i32_t [|  |] in
 let main_function         = L.define_function "main" main_t the_module in
```

```
 let builder               = L.builder_at_end context (L.entry_block main_function)
in

 let str_format_str        = L.build_global_stringptr "%s"   "strfmt"        builder
in
 let str_format_str_endline = L.build_global_stringptr "%s\n" "strfmtendline" builder
in
  (* ----------- beginning of stack-related bookkeeping functions -----------

    These stack-related functions are only for bookkeeping purposes.
    Specifically, when generating branching instructions (If and While),
    we need to know which function we are inside at that given point since
    we need to create new blocks like "merge", "then", "else", etc.
    For this, the the_function () function is useful since it returns the
    function definition for which the code is being generated at any
    given point. To maintain this invariant, whenever we see a new function
    being created (LAMBDA in our case), we need to ensure that its
    definition is pushed to the stack and then only its body is generated.
    The stack must be popped once we are done generating instructions for
    that function body.
 *)
  let curr_function     = ref (S.create ()) in
 let the_function   () = S.top      !curr_function in
 let push_function   f = S.push f   !curr_function in
 let pop_function   () = S.pop      !curr_function in
 let is_stack_empty () = S.is_empty !curr_function in

 (* ----------- end of stack-related bookkeeping functions ----------- *)
  (* dumps the symbol table to a list of (name, llvalue) pairs *)
 let rec list_of_llvals (scope : symbol_table) builder =
   let rec helper acc curr_scope =
     (match curr_scope.parent with
       None -> acc @ (StringMap.bindings curr_scope.variables)
     | Some(p) -> helper (acc @ (StringMap.bindings curr_scope.variables)) p)
   in
   let (list_llvals, builder) = (helper [] scope, builder) in
```

```
   let seen_names = ref StringMap.empty in
   let (fptrs, builder) = list_of_fptrs !env builder in
   let fptrs = List.map fst fptrs in
   let result = List.rev (List.fold_left (fun acc (name, llval) ->
     if List.mem name fptrs then acc else
     if StringMap.mem name !seen_functions then acc else
     if StringMap.mem name !seen_names then acc else
     let _ = seen_names := StringMap.add name true !seen_names in
     if (is_llval_pointer llval) then (name, L.build_load llval name builder) :: acc
else
     if (find_shared !env name) then (name, llval) :: acc
     else (name, L.build_load llval name builder) :: acc)
     [] list_llvals) in (result, builder)
and list_of_fptrs (scope : symbol_table) builder =
     let rec helper acc curr_scope =
       (match curr_scope.parent with
         None -> acc @ (StringMap.bindings curr_scope.functionpointers)
       | Some(p) -> helper (acc @ (StringMap.bindings curr_scope.functionpointers)) p)
     in
     let list_fptrs = helper [] scope in
     let result = List.rev (List.fold_left (fun acc bs -> bs :: acc) [] list_fptrs)
     in (result, builder) in

 (* ----- start of builtin function declarations ----- *)

 let printf_t            = L.var_arg_function_type i32_t [| string_t |] in
 let printf_func         = L.declare_function "printf" printf_t the_module in

 let int_to_string_t     = L.function_type string_t [| i32_t |] in
 let int_to_string_func  = L.declare_function "int_to_string" int_to_string_t
the_module in
  let float_to_string_t    = L.function_type string_t [| float_t |] in
 let float_to_string_func = L.declare_function "float_to_string" float_to_string_t
the_module in

 let bool_to_string_t     = L.function_type string_t [| i1_t |] in
```

```
 let bool_to_string_func  = L.declare_function "bool_to_string" bool_to_string_t
the_module in

 let int_to_float_t       = L.function_type float_t [| i32_t |] in
 let int_to_float_func    = L.declare_function "int_to_float" int_to_float_t
the_module in

 let float_to_int_t       = L.function_type i32_t [| float_t |] in
 let float_to_int_func    = L.declare_function "float_to_int" float_to_int_t
the_module in

 let strlen_t             = L.function_type i32_t [| string_t |] in
 let strlen_func          = L.declare_function "strlen" strlen_t the_module in

 let string_concat_t      = L.function_type string_t [| string_t; string_t |] in
 let string_concat_func   = L.declare_function "string_concat" string_concat_t
the_module in

 let string_substr_t      = L.function_type string_t [| string_t; i32_t; i32_t |] in
 let string_substr_func   = L.declare_function "string_substr" string_substr_t
the_module in

 let string_eq_t          = L.function_type i1_t [| string_t; string_t |] in
 let string_eq_func       = L.declare_function "string_eq" string_eq_t the_module in

 let list_insert_t        = L.function_type quack_t [| (L.pointer_type list_t); i32_t;
voidptr_t |] in
 let list_insert_func     = L.declare_function "List_insert" list_insert_t the_module
in

 let list_len_t           = L.function_type i32_t [| (L.pointer_type list_t) |] in
 let list_len_func        = L.declare_function "List_len" list_len_t the_module in

 let list_remove_t       = L.function_type quack_t [| (L.pointer_type list_t); i32_t
|] in
```

```
 let list_remove_func    = L.declare_function "List_remove" list_remove_t the_module
in

 let list_replace_t     = L.function_type quack_t [| (L.pointer_type list_t); i32_t;
voidptr_t |] in
 let list_replace_func  = L.declare_function "List_replace" list_replace_t the_module
in

 let list_at_t          = L.function_type voidptr_t [| (L.pointer_type list_t); i32_t
|] in
 let list_at_func       = L.declare_function "List_at" list_at_t the_module in

 let store_insert_t     = L.function_type quack_t [| voidptr_t; i32_t; i32_t |] in
 let store_insert_func  = L.declare_function "store_insert" store_insert_t the_module
in

 let store_lookup_t     = L.function_type i32_t [| voidptr_t; i32_t |] in
 let store_lookup_func  = L.declare_function "store_lookup" store_lookup_t the_module
in
 (* ----- end of builtin function declarations ----- *)

 (* ----- start of thread related function declarations ----- *)
 let pthread_create_t          = L.function_type i32_t [| (L.pointer_type pthread_t);
voidptr_t; (L.pointer_type (L.function_type voidptr_t [| voidptr_t |])); voidptr_t |]
in
 let pthread_join_t            = L.function_type i32_t [| pthread_t; (L.pointer_type
voidptr_t) |] in
 let pthread_mutex_init_t      = L.function_type (L.pointer_type mutex_t) [| |] in
 let pthread_mutex_lock_t      = L.function_type i32_t [| mutex_t |] in
 let pthread_mutex_unlock_t    = L.function_type i32_t [| mutex_t |] in

 let pthread_create_func       = L.declare_function "pthread_create" pthread_create_t
the_module in
 let pthread_join_func         = L.declare_function "pthread_join" pthread_join_t
the_module in
```

```
 let pthread_mutex_init_func   = L.declare_function "Mutex_init" pthread_mutex_init_t
the_module in
 let pthread_mutex_lock_func   = L.declare_function "pthread_mutex_lock"
pthread_mutex_lock_t the_module in
 let pthread_mutex_unlock_func = L.declare_function "pthread_mutex_unlock"
pthread_mutex_unlock_t the_module in
 (* ----- end of thread related function declarations ----- *)

 let rec statement builder = function
     SExpr e -> let _     = expr builder e in builder
   | SIf (predicate, then_stmt, else_stmt) ->
       let the_function   = the_function () in

       let bool_val       = expr builder predicate in

       let _ = push_scope () in

       let merge_bb       = L.append_block context "merge" the_function in
         let branch_instr = L.build_br merge_bb in

       let then_bb        = L.append_block context "then" the_function in
         let then_builder = List.fold_left statement (L.builder_at_end context
then_bb) then_stmt in
       let ()             = add_terminal then_builder branch_instr in

       let _ = pop_scope() in
       let _ = push_scope () in

       let else_bb        = L.append_block context "else" the_function in
         let else_builder = List.fold_left statement (L.builder_at_end context
else_bb) else_stmt in
       let ()             = add_terminal else_builder branch_instr in

       let _ = pop_scope() in
```

```
      let _              = L.build_cond_br bool_val then_bb else_bb builder in
L.builder_at_end context merge_bb
   | SWhile (predicate, body) ->
      let the_function      = the_function () in

      let pred_bb           = L.append_block context "while" the_function in
      let _                 = L.build_br pred_bb builder in

      let pred_builder      = L.builder_at_end context pred_bb in
      let bool_val          = expr pred_builder predicate in

      let _ = push_scope() in

      let body_bb           = L.append_block context "while_body" the_function in
          let while_builder = List.fold_left statement (L.builder_at_end context
body_bb) body in
      let ()                = add_terminal while_builder (L.build_br pred_bb) in

      let _ = pop_scope() in

      let merge_bb          = L.append_block context "merge" the_function in
      let _                 = L.build_cond_br bool_val body_bb merge_bb pred_builder
in L.builder_at_end context merge_bb
   | SAssign (name, ((t, _) as sx)) ->
      let e' = expr builder sx in
      (match t with
        A.Arrow(_, _) ->
          let fptr = e' in
          let new_scope = {variables = StringMap.add name fptr !env.variables; shared
= StringMap.add name false !env.shared; stored = !env.stored; parent = !env.parent;
functionpointers = StringMap.add name fptr !env.functionpointers}
          in let _ = env := new_scope in builder
        | _ ->
          if is_pointer t && (not (is_mutex t)) then
            let lhs = find_variable !env name in
            let _ = L.build_store e' lhs builder in builder
```

```
          else
            let _  = L.build_store e' (find_variable !env name) builder in builder)
   | SDefine(_, A.Arrow(formals, retty), name, (t, e)) ->
     (match e with
       (* there is no function to build here; assign to pre-existing function ptr *)
       SVar(_, rhs) -> let fptr = find_variable !env rhs in
         let new_scope = {variables = StringMap.add name fptr !env.variables; shared =
StringMap.add name false !env.shared; stored = StringMap.add name (find_stored !env
rhs) !env.stored; parent = !env.parent; functionpointers = StringMap.add name fptr
!env.functionpointers}
           in let _ = env := new_scope in builder
       | SCall(f, _) -> let fptr = expr builder (t , e) in
                          let new_scope = {variables = StringMap.add name fptr
!env.variables; shared = StringMap.add name false !env.shared; stored = StringMap.add
name (find_stored !env f) !env.stored; parent = !env.parent; functionpointers =
StringMap.add name fptr !env.functionpointers}
                          in let _ = env := new_scope in builder
     | SLambda (_, _, _, _) ->
       let ftype = ftype_from_t (A.Arrow(formals, retty)) in
       let f = L.define_function name ftype the_module in
       let fptr = L.build_bitcast f (L.pointer_type ftype) (name ^ "_ptr") builder in
         build_named_function name builder fptr e
     | _ -> raise (Failure "Invalid function definition"))
   | SDefine(s, typ, name, e) -> let _  = add_to_scope (s, (expr builder e), name)
builder typ in builder
   | SReturn (_, SNoexpr) ->  let _ = L.build_ret_void builder in builder
   | SReturn e -> let _ = L.build_ret (expr builder e) builder in builder
 and build_malloc builder llval =
     let heap = L.build_malloc (L.type_of llval) "heap" builder in
     let _    = L.build_store llval heap builder in heap
 and expr builder (t, e) = match e with
     SNoexpr     -> L.const_int i32_t 0
   | SLiteral i  -> L.const_int i32_t i
   | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
   | SFliteral l -> L.const_float_of_string float_t l
   | SListLit l  ->
```

```
    let llvals = List.map (expr builder) l in
    let malloced_ptrs = List.map (build_malloc builder) llvals in (* addresses of
malloc'd locations for the llvals on the heap *)
    let list_ptr = L.build_alloca (L.pointer_type list_t) "list_ptr" builder in
(*   Node **l;   *)
    let head = L.build_malloc list_t "head" builder in
    let _ = L.build_store head list_ptr builder in
    let _ = L.build_store (L.const_null list_t) head builder in
    let _ = List.fold_left (fun _ (i, llval) ->
      (* cast each llval to a void * before inserting it into the list *)
      let void_cast = L.build_bitcast llval voidptr_t "voidptr" builder in
      let listval = L.build_load list_ptr "listval" builder in
      L.build_call list_insert_func [| listval; L.const_int i32_t i; void_cast |] ""
builder
    ) list_ptr (List.mapi (fun i llval -> (i, llval)) malloced_ptrs) in L.build_load
list_ptr "listlit" builder
  | SVar (_, name)                -> let llval = find_variable !env name in
                                     if (is_function t) then llval
                                     else L.build_load llval name builder
  | SStringLiteral s              -> L.build_global_stringptr s "strlit" builder
  | SCall ("print", [e])          -> L.build_call printf_func [| str_format_str;
(expr builder e) |] "print" builder
  | SCall ("println", [e])        -> L.build_call printf_func [|
str_format_str_endline; (expr builder e) |] "println" builder
  | SCall ("int_to_string", [e])   -> L.build_call int_to_string_func [| (expr
builder e) |] "int_to_string" builder
  | SCall ("float_to_string", [e]) -> L.build_call float_to_string_func [| (expr
builder e) |] "float_to_string" builder
  | SCall ("bool_to_string", [e])  -> L.build_call bool_to_string_func [| (expr
builder e) |] "bool_to_string" builder
  | SCall ("int_to_float", [e])    -> L.build_call int_to_float_func [| (expr builder
e) |] "int_to_float" builder
  | SCall ("float_to_int", [e])    -> L.build_call float_to_int_func [| (expr builder
e) |] "float_to_int" builder
  | SCall ("String_eq", [e1; e2])  -> L.build_call string_eq_func [| (expr builder
e1); (expr builder e2) |] "string_eq" builder
```

```
  | SCall ("String_len", [e])      -> L.build_call strlen_func [| (expr builder e) |]
"strlen" builder
  | SCall ("List_len", [e])        -> L.build_call list_len_func [| expr builder e |]
"list_len" builder
  | SCall ("String_concat", [e1; e2])     -> L.build_call string_concat_func [| (expr
builder e1); (expr builder e2) |] "string_concat" builder
  | SCall ("String_substr", [e1; e2; e3]) -> L.build_call string_substr_func [| (expr
builder e1); (expr builder e2); (expr builder e3) |] "string_substr" builder
  | SCall ("List_insert", [e1; e2; e3])   -> L.build_call list_insert_func [| (expr
builder e1); (expr builder e2); (L.build_bitcast (build_malloc builder (expr builder
e3)) voidptr_t "voidptr" builder) |] "" builder
  | SCall ("List_remove", [e1; e2])       -> L.build_call list_remove_func [| (expr
builder e1); (expr builder e2) |] "" builder
  | SCall ("List_replace", [e1; e2; e3])  -> L.build_call list_replace_func [| (expr
builder e1); (expr builder e2); (L.build_bitcast (build_malloc builder (expr builder
e3)) voidptr_t "voidptr" builder) |] "" builder
  | SCall ("List_at", [((A.List(t1), _) as e1); e2]) ->
    let value = L.build_call list_at_func [| (expr builder e1); (expr builder e2) |]
"list_at" builder in
    let cast =
      if (is_pointer t1 && (not (is_mutex t))) then
        L.build_bitcast value (L.pointer_type (L.pointer_type (ltype_of_typ t1)))
"cast" builder
      else L.build_bitcast value (L.pointer_type (ltype_of_typ t1)) "cast" builder in
    if is_function t1 then cast else L.build_load cast "list_at" builder
  | SCall ("Thread_join", [e]) -> L.build_call pthread_join_func [| expr builder e;
L.const_null (L.pointer_type voidptr_t) |] "" builder
  | SCall ("Mutex", []) -> L.build_load (L.build_call pthread_mutex_init_func [| |]
"" builder) "mutex" builder
  | SCall ("Mutex_lock", [e]) ->   L.build_call pthread_mutex_lock_func   [| expr
builder e |] "" builder
  | SCall ("Mutex_unlock", [e]) -> L.build_call pthread_mutex_unlock_func [| expr
builder e |] "" builder
  | SCall (f, args) ->
    let global_store_struct_option = find_stored !env f in
```

```
    let is_storefunc = (match global_store_struct_option with Some _ -> true | None
-> false)in
    let fdef   = find_variable !env f in
    let retty  = L.return_type (L.element_type (L.type_of fdef)) in
    let llargs = (List.map (expr builder) args) in
    let result = if retty = quack_t then "" else f ^ "_result" in
    if is_storefunc then
      let global_store_struct = (match global_store_struct_option with Some a -> a |
None -> (L.const_int i32_t 0)) in
      if global_store_struct = (L.const_int i32_t 0) then L.build_call fdef
(Array.of_list llargs) result builder else

      let arg = (Array.of_list llargs).(0) in (* safe because functions with store
always have one argument *)

      let global_store_struct = L.build_bitcast global_store_struct voidptr_t
"voidptr" builder in

      let result = L.build_call store_lookup_func [| global_store_struct; arg |]
"lookup_result" builder in
      let int32_min = L.const_int i32_t (-2147483648) in
      let is_min = L.build_icmp L.Icmp.Eq result int32_min "" builder in

      let result_stackaddr = L.build_alloca i32_t "result_stackaddr" builder in

      (* if result = int32_min, then we want to store the result of this call to
        the store and return the result. Otherwise, we want to return the
        result we got from the store *)

      let the_function = the_function () in

      let then_bb = L.append_block context "then" the_function in
      let else_bb = L.append_block context "else" the_function in
      let merge_bb = L.append_block context "merge" the_function in
      let _ = L.build_cond_br is_min then_bb else_bb builder in
```

```
      let then_builder = L.builder_at_end context then_bb in
      let else_builder = L.builder_at_end context else_bb in
      let merge_builder = L.builder_at_end context merge_bb in

      (* so, inside then:, call the actual function and cache the result *)
      let retval = L.build_call fdef (Array.of_list llargs) (f ^ "_result")
then_builder in
      let _ = L.build_call store_insert_func [| global_store_struct; arg; retval |]
"" then_builder in

      (* store val from function call in stack address *)
      let _ = L.build_store retval result_stackaddr then_builder in
      let _ = L.build_br merge_bb then_builder in

      (* Otherwise, in else:, store val from cache in the above stack address *)
      let _ = L.build_store result result_stackaddr else_builder in
      let _ = L.build_br merge_bb else_builder in

      (* move the builder to the end of the merge block so further code can be added
*)
      let _ = L.position_at_end merge_bb builder in L.build_load result_stackaddr
"result" merge_builder
    else L.build_call fdef (Array.of_list llargs) result builder
  | SBinop (e1, op, e2) ->
    let (t, _) = e1
    and e1'    = expr builder e1
    and e2'    = expr builder e2 in
    if t = A.Float
    then (match op with
      A.Add     -> L.build_fadd
    | A.Sub     -> L.build_fsub
    | A.Mult    -> L.build_fmul
    | A.Div     -> L.build_fdiv
    | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
    | A.Neq     -> L.build_fcmp L.Fcmp.One
    | A.Less    -> L.build_fcmp L.Fcmp.Olt
```

```
    | A.Leq     -> L.build_fcmp L.Fcmp.Ole
    | A.Greater -> L.build_fcmp L.Fcmp.Ogt
    | A.Geq     -> L.build_fcmp L.Fcmp.Oge
    | A.Mod        -> raise (Failure "Internal Error: semant should have rejected mod
on float")
    | A.And | A.Or -> raise (Failure "internal Error: semant should have rejected
and/or on float")
         ) e1' e2' "tmp" builder
    else (match op with
    | A.Add     -> L.build_add
    | A.Sub     -> L.build_sub
    | A.Mult    -> L.build_mul
    | A.Div     -> L.build_sdiv
    | A.Mod     -> L.build_srem
    | A.And     -> L.build_and
    | A.Or      -> L.build_or
    | A.Equal   -> L.build_icmp L.Icmp.Eq
    | A.Neq     -> L.build_icmp L.Icmp.Ne
    | A.Less    -> L.build_icmp L.Icmp.Slt
    | A.Leq     -> L.build_icmp L.Icmp.Sle
    | A.Greater -> L.build_icmp L.Icmp.Sgt
    | A.Geq     -> L.build_icmp L.Icmp.Sge
         ) e1' e2' "tmp" builder
  | SUnop (op, e) ->
    let (t, _) = e in
    let e' = expr builder e in
    (match op with
        A.Neg when t = A.Float -> L.build_fneg
      | A.Neg                  -> L.build_neg
      | A.Not                  -> L.build_not
    ) e' "tmp" builder
  | SThread body ->
    let funtype  = A.Arrow([], A.Quack) in
    let fdef     = expr builder (funtype, SLambda(false, A.Quack, [], body)) in
    let fcast    = L.build_bitcast fdef (L.pointer_type (L.function_type voidptr_t
[|voidptr_t|])) "fptr_cast" builder in (* complying with pthread_create's signature *)
```

```
    let pthread_t_ref = L.build_alloca pthread_t "pthread_t" builder in
    let _ = L.build_call pthread_create_func [|pthread_t_ref; L.const_null voidptr_t;
fcast; L.const_null voidptr_t|] "" builder in
    L.build_load pthread_t_ref "pthread_t" builder
  | (SLambda (store, retty, formals, _)) as e ->
    let typ = A.Arrow((List.map fst formals), retty) in
    let name = generate_name () in
    let _ = statement builder (SDefine(store, typ, name, (typ, e))) in
    find_variable !env name
 and generate_name () =
   let name = "#anon_" ^ (string_of_int !anon_counter) in
   let _ = anon_counter := !anon_counter + 1 in
   name
 and add_params_to_scope (s, p, n) builder t =
 if is_pointer t then
   if is_mutex t then
    let local = L.build_alloca (ltype_of_typ t) n builder in
    let _     = L.build_store p local builder in
    let new_scope = {variables = StringMap.add n local !env.variables; shared =
StringMap.add n s !env.shared; stored = !env.stored; parent = !env.parent;
functionpointers = !env.functionpointers}
      in env := new_scope
   else
    let listptr = L.build_alloca (L.pointer_type (ltype_of_typ t)) (n ^ "listparam")
builder in
    let _ = L.build_store p listptr builder in
    let new_scope = {variables = StringMap.add n listptr !env.variables; shared =
StringMap.add n s !env.shared; stored = !env.stored; parent = !env.parent;
functionpointers = !env.functionpointers}
      in env := new_scope
 else if is_function t then
   let new_scope = {variables = StringMap.add n p !env.variables; shared =
StringMap.add n s !env.shared; stored = StringMap.add n None !env.stored; parent =
!env.parent; functionpointers = StringMap.add n p !env.functionpointers}
    in let _ = env := new_scope
    in seen_functions := StringMap.add n true !seen_functions
```

```
 else
   let _ = add_to_scope (s, p, n) builder t in ()
  (* creates a struct with field {llval}, fills it up with the provided
    llval argument and returns the created struct
 *)
 and struct_of_llval fname builder binding =
   let name_str = "(" ^ fname ^ "):" ^ (fst binding) ^ ":" in
   let str_type = L.named_struct_type context name_str in
   let v = snd binding in
   let _ = L.struct_set_body str_type [| L.type_of v |] false in
   let struct_alloc = L.build_alloca str_type (fname ^ "__struct__" ^ (fst binding))
builder in
   let vptr = L.build_struct_gep struct_alloc 0 "v" builder in
   let _ = L.build_store v vptr builder in
   L.build_load struct_alloc name_str builder
  (* dumps the current scope into a list of structs, each struct containing an llval.
    returns the list of structs and the updated builder
 *)
 and dump_scope fname builder =
   (* add this function to the seen list. This prevents functions from being
   captured by themselves and other functions. *)
   let _ = seen_functions := StringMap.add fname true !seen_functions in
   let (llval_bindings, builder) = list_of_llvals !env builder in
   let seen_names = ref StringMap.empty in
   let llval_bindings = List.fold_left (fun acc (n, v) ->
       (* skip ourself and any other functions *)
       if String.equal n fname then acc else if StringMap.mem n !seen_functions then
acc else
       let has_seen = StringMap.mem n !seen_names in
       let answer = if has_seen then acc else let _ = seen_names := StringMap.add n
true !seen_names in (n, v) :: acc
       in  answer) [] llval_bindings in (List.map (struct_of_llval fname builder)
llval_bindings, builder)
 and build_named_function name builder fptr = function
   SLambda (store, retty, formals, body) ->
```

```
    let can_use_store = store && (retty = A.Int) && ((List.length formals) = 1) &&
((fst (List.hd formals)) = A.Int) in
    let _ = if can_use_store then
      (* store only for int -> int functions *)

      let store_elem_struct = L.named_struct_type context (name ^
"_store_elem_struct#") in
      let _ = L.struct_set_body store_elem_struct [| i32_t ; i32_t |] false in

      let store_elem_arrayty = (L.array_type store_elem_struct 32) in
      let store_struct = L.named_struct_type context (name ^ "_store_struct#") in

      (* a store struct contains { latest_index, full_marker, element array } *)
      let _ = L.struct_set_body store_struct [| i32_t; i32_t; store_elem_arrayty |]
false in

      let global_store_struct = L.define_global ("global_" ^ name ^ "_store#")
(L.const_null store_struct) the_module in

      let new_scope = {variables = StringMap.add name fptr !env.variables; shared =
StringMap.add name false !env.shared; stored = StringMap.add name (Some
global_store_struct) !env.stored; parent = !env.parent; functionpointers =
StringMap.add name fptr !env.functionpointers}
      in let _ = env := new_scope in ()
    else
      let new_scope = {variables = StringMap.add name fptr !env.variables; shared =
StringMap.add name false !env.shared; stored = StringMap.add name None !env.stored;
parent = !env.parent; functionpointers = StringMap.add name fptr
!env.functionpointers}
      in let _ = env := new_scope in
      ()
    in
    let closure_struct = L.named_struct_type context (name ^ "_closure_struct#") in
    let (dumped_scope, builder) = dump_scope name builder in

    let (fptrs, builder) = list_of_fptrs !env builder in
```

```
    let fptrs = List.filter (fun (n, _) -> not (String.equal n name)) fptrs in
    let fptrs = List.fold_left (fun acc (n, v) -> if List.exists (fun (n2, v2) -> (v
= v2) || (n = n2)) acc then acc else (n, v) :: acc) [] fptrs in
    let struct_of_fptrs = List.map (struct_of_llval name builder) fptrs in
    let all_structs = dumped_scope @ struct_of_fptrs in
    let non_fptr_bound = List.length dumped_scope in

    let dumped_scope = all_structs in

    let dumped_scope_tys = List.map L.type_of dumped_scope in
    let _ = L.struct_set_body closure_struct (Array.of_list (dumped_scope_tys)) false
in
    let global_closure_struct = L.define_global ("global_" ^ name ^ "_closure#")
(L.const_null closure_struct) the_module in

    (* fill up global closure struct with individual structs of captured variables *)
    let _ = List.fold_left (fun index dumped_llval ->
      let ith_struct = L.build_struct_gep global_closure_struct index "" builder in
      let _ = L.build_store dumped_llval ith_struct builder in
    index + 1) 0 dumped_scope in ();

    (* build function body *)
    let fdef = find_variable !env name in
    let _ = push_function fdef in
    let _ = push_scope () in

    let fun_builder = L.builder_at_end context (L.entry_block fdef) in

    (* unpacking the variables in the closure *)
    let _ = List.iteri (fun index _ ->
      let var_struct_ptr = L.build_struct_gep global_closure_struct index ""
fun_builder in
      let struct_var = L.build_load var_struct_ptr "individual_data_struct"
fun_builder in
      let struct_typ = L.type_of struct_var in
```

```
      let struct_local = L.build_alloca struct_typ "" fun_builder in
      let _ = L.build_store struct_var struct_local fun_builder in

      let struct_name = match (L.struct_name struct_typ) with Some(s) -> s | None ->
"" in
      let var_name = List.nth (String.split_on_char ':' struct_name) 1 in

      let is_shared = find_shared !env var_name in

      let llval_ptr = L.build_struct_gep struct_local 0 "" fun_builder in
      let llval = L.build_load llval_ptr var_name fun_builder in
      let is_list = (L.type_of llval = (L.pointer_type list_t)) in

      let shared_and_not_list = is_shared && (not is_list) in

      if (index < non_fptr_bound) then
        let llval_local = L.build_alloca (L.type_of llval) (var_name ^ "_ptr")
fun_builder in
        let _ = L.build_store llval llval_local fun_builder in

        let address = if (shared_and_not_list) then (L.build_load llval_local ""
fun_builder) else llval_local in

        let _ = let new_scope = {variables = StringMap.add var_name address
!env.variables; shared = StringMap.add var_name is_shared !env.shared; stored =
!env.stored; parent = !env.parent; functionpointers = !env.functionpointers} in
        env := new_scope in ()
      else
         (* we are unpacking captured fptrs here now *)
        let _ = let new_scope = {variables = StringMap.add var_name llval
!env.variables; shared = StringMap.add var_name is_shared !env.shared; stored =
StringMap.add var_name (find_stored !env var_name) !env.stored; parent = !env.parent;
functionpointers = StringMap.add var_name llval !env.functionpointers} in
        env := new_scope in ()) dumped_scope in

    if List.length formals > 0 then
```

```
      (* add params to scope *)
      let _ = List.iter2 (fun (t, n) p ->
        let () = L.set_value_name n p in
        let is_shared = match t with A.List(_) | A.Mutex -> true | _ -> false in
        let _ = add_params_to_scope (is_shared, p, n) fun_builder t in ()) formals
(Array.to_list (L.params fdef)) in ()
    else ();

    (* build body *)
    let final_builder = List.fold_left statement fun_builder body in
    let instr = if retty = A.Quack then L.build_ret_void else L.build_ret
(L.const_int i32_t 0) in
    let _ = add_terminal final_builder instr in

    let _ = pop_scope () in
    let _ = pop_function () in builder
  | _ -> raise (Failure "Internal Error: non-lambda expression passed to
build_named_function")
 and build_main_function builder statements =
  (* Note to self: at this point, final_builder is pointing to the END of the main
function.
     The call to statement generates instructions for all statments in this main
function,
     which subsequently keeps on updating the instruction builder. Therefore, after
the last
     instruction in main's body is generated, the builder points to that instruction
and
     this is stored in final_builder. This is different that the `builder` in the
argument since that
     builder is still pointing to the beginning of the main function! *)

  (* Push main_function definition to the curr_function stack first *)
  let _ = push_function main_function in

  let final_builder = List.fold_left statement builder statements in
```

```
   (* End the main function's basic block with a terminal, returning 0 *)
   let _ = add_terminal final_builder (L.build_ret (L.const_int i32_t 0)) in

   (* Pop main_function definition from the curr_function stack *)
   let _ = pop_function () in
     if (not (is_stack_empty ())) then raise (Failure "Internal Error: stack should be
empty after building main function. A function was not popped from the stack after it
was built.")
     else ()

 and add_terminal builder instr =
   (match L.block_terminator (L.insertion_block builder) with
     Some _ -> ()
   | None -> ignore (instr builder))
in

(* We only have one top-level function, main.
  All statements of the sPool program reside within main *)
build_main_function builder statements;

(* Return the final module *)
the_module;
```

8.1.8 store.c

```
// store.c
// Implements functions related to the store feature for sPool that are later
// linked with the sPool compiler
//
// Written by: Team Nautilus (Ankur, Yuma, Max, Etha)

#include <stdio.h>
#include <stdint.h>
#include <pthread.h>
#include <assert.h>

#define nullptr NULL
```

```c
#define DEBUG 0
#define INCLUDEMAIN 0

static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

#define STORE_SIZE 32

typedef struct Elem {
   int param;
   int result;
} Elem;

typedef struct Store {
   int curr_index;
   int full; // bool -> is it full?
   Elem stored[STORE_SIZE];
} Store;

void store_insert(void *s, int param, int result) {
   assert(s);
   pthread_mutex_lock(&lock);
#if DEBUG
   printf("\nstore_insert\nInserting arg %d = result %d\n", param, result);
#endif
   Store *store = (Store *)s;
   int index = store->curr_index;
   Elem e = {param = param, result = result};
   store->stored[index] = e;
   store->curr_index = (index + 1) % STORE_SIZE;
   store->full = store->full || ((store->curr_index) == 0);
   pthread_mutex_unlock(&lock);
#if DEBUG
   printf("\ndone inserting arg %d = result %d\n", param, result);
#endif
}
```

```c
int store_lookup(void *s, int param) {
   assert(s);
   pthread_mutex_lock(&lock);
#if DEBUG
   printf("\nstore_lookup\nLooking up %d\n", param);
#endif
   Store *store = (Store *)s;
   int start = 0;
   int end = store->full ? (STORE_SIZE - 1) : store->curr_index;
   int result = INT32_MIN;

   while (start <= end) {
       Elem e = store->stored[start];
       if (e.param == param) {
           result = e.result;
           break;
       }
       start += 1;
   }
   pthread_mutex_unlock(&lock);
#if DEBUG
   printf("\nreturned %d\n", result);
#endif
   return result;
}

#if INCLUDEMAIN
int main() {

   Store stores;

   return 0;
}
#endif
```

8.1.9 builtins.c

```c
// builtins.c
// Implements functions for sPool that are later linked with the sPool compiler
//
// Written by: Team Nautilus (Ankur, Yuma, Max, Etha)

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <pthread.h>

#define DEBUG 0

const char *int_to_string(int num)
{
   int length = snprintf(NULL, 0, "%d", num);
   char *str = malloc(length + 1); assert(str);
   snprintf(str, length + 1, "%d", num);
   return str;
}

const char *float_to_string(double num)
{
   int length = snprintf(NULL, 0, "%f", num);
   char *str = malloc(length + 1); assert(str);
   snprintf(str, length + 1, "%f", num);
   return str;
}

const char *bool_to_string(int b)
{
   return b? "true": "false";
}

double int_to_float(int num)
```

```c
{
   return (double)num;
}

int float_to_int(double num)
{
   return (int)num;
}

const char *string_concat(const char *s1, const char *s2)
{
   int l1 = strlen(s1);
   int l2 = strlen(s2);
   int len = l1 + l2;
   char *result = malloc(len + 1); assert(result);

   for (int i = 0; i < strlen(s1); i++) result[i]      = s1[i];
   for (int j = 0; j < strlen(s2); j++) result[j + l1] = s2[j];
                                        result[len]    = '\0';

   return result;
}

const char *string_substr(const char *s1, int m, int n)
{
   int len = strlen(s1);
   assert((m < n) && (m >= 0 && m <= (len - 1)) && (n >= 0 && n <= len));
   char *result = malloc(n - m + 1); assert(result);

   int i = 0;
   while (m < n) {
       result[i++] = s1[m++];
   }
   result[i] = '\0';

   return result;
```

```
}

int string_eq(const char *s1, const char *s2)
{
   return strcmp(s1, s2) == 0;
}

// a helper for mutex to circumvent LLVM's "cannot allocate unsized type" error
pthread_mutex_t **Mutex_init()
{
   pthread_mutex_t **mutex = (pthread_mutex_t **)malloc(sizeof(pthread_mutex_t *));
assert(mutex);
   *mutex = (pthread_mutex_t *)malloc(sizeof(pthread_mutex_t)); assert(*mutex);
   assert(pthread_mutex_init(*mutex, NULL) == 0);
   return mutex;
}

#if DEBUG
int main(void) {
   printf("Enter a number: \n");
   int num;
   scanf("%d", &num);
   printf("Int_to_string: %s\n", int_to_string(num));
}
#endif
```

8.1.10 list.c

```
// list.c
// Implements list functions for sPool that are later linked with the sPool compiler
//
// Written by: Team Nautilus (Ankur, Yuma, Max, Etha)

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
```

```c
#define nullptr NULL
#define DEBUG 0

typedef struct Node
{
   void *data;
   struct Node *next;
} Node;

int List_len(Node **l)
{
   if (!l || !*l) {
       return 0;
   }

   int len = 0;
   Node *temp = *l;

   while (temp != nullptr) {
       len++;
       temp = temp->next;
   }
   return len;
}

void *List_at(Node **l, int index)
{
   if (!l || !*l) {
       return nullptr;
   }

   int len = List_len(l);
   assert((index >= 0) && (index < len));

   Node *temp = *l;
   for (int i = 0; i < index; i++) temp = temp->next;
```

```
    return temp->data;
}

// data to be inserted should have already been
// allocated on the heap by the time this function
// is called.
void List_insert(Node **head, int index, void *v)
{

    int len = List_len(head); assert((index >= 0) && (index <= len));

    Node *curr = *head; Node *prev = nullptr;

    for (int i = 0; i < index; i++) {
        prev = curr;
        curr = curr->next;
    }

    Node *node = malloc(sizeof(*node)); assert(node);
    node->data = v; node->next = curr;

    if (prev == nullptr) { // adding to the head of the list
        *head = node;
    } else {
        prev->next = node;
    }
}

void List_remove(Node **head, int index)
{
    int len = List_len(head); assert((index >= 0) && (index < len));

    Node *curr = *head; Node *prev = nullptr;

    for (int i = 0; i < index; i++) {
        prev = curr;
```

```
        curr = curr->next;
    }

    if (prev == nullptr) { // removing the head of the list
        *head = curr->next;
    } else {
        prev->next = curr->next;
    }

    if (curr) free(curr);
}

void List_replace(Node **head, int index, void *v)
{
    assert(head && *head);
    int len = List_len(head); assert((index >= 0) && (index < len));

    Node *curr = *head; void *old = nullptr;
    for (int i = 0; i < index; i++) {
        curr = curr->next;
    }
    old = curr->data;
    curr->data = v;

    if (old) free(old);
}

#if DEBUG
void List_int_print(Node **l)
{
    if (!l || !*l) {
        printf("[]\n");
        return;
    }

    Node *t = *l;
```

```
    int i = 0;
    int len = List_len(l);
    printf("[ ");
    while (i < len) {
        int *data = (int *)List_at(l, i);
        printf("%d ", *data);
        t = t->next;
        i = i + 1;
    }
    printf("]\n");
}
#endif

#if DEBUG
int main()
{
    int *a = malloc(sizeof(a));
    int *b = malloc(sizeof(b));
    int *c = malloc(sizeof(c));
    int *d = malloc(sizeof(d));

    *a = 1; *b = 2; *c = 3; *d = 4;

    Node *l = nullptr; // our list....

    List_insert(&l, 0, a);
    List_insert(&l, 1, b);
    List_insert(&l, 2, c);
    List_insert(&l, 0, d);

    printf("%d\n", List_len(&l));
    List_int_print(&l);

    List_replace(&l, 0, a);
    List_replace(&l, 3, b);
    List_int_print(&l);
```

```
}
#endif
```

8.2 The sPool Standard Library

This section contains the code listing for the sPool standard library, which includes two modules – the integer list standard library and the string standard library.

8.2.1 list.sP

```
##########################################
# start of sPool List_int Standard Library #
# Written by: Team Nautilus (Ankur, Yuma, Max, Etha) #
##########################################

# helper function to create an 'n' element
# list pre-filled with a default value
def list<int> List__helper__(int len, int value):
   int i = 0
   list<int> result = []
   while (i < len):
       List_insert(result, i, value)
       i = i + 1;
   return result;

def list<int> List_int_map(list<int> l, (int -> int) f):
   int i = 0
   int len = List_len(l)
   list<int> result = List__helper__(len, -1)

   while (i < len):
       List_replace(result, i, f(List_at(l, i)))
       i = i + 1;

   return result;


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;


def list<int> List_int_rev(list<int> l):
   int i = 0
   int len = List_len(l)
   list<int> result = List__helper__(len, -1)

   while (i < len):
       List_replace(result, i, List_at(l, len - i - 1))
       i = i + 1;
   return result;


def int List_int_fold((int, int -> int) combine, int zero, list<int> l):
   int i = 0
   int len = List_len(l)
   while (i < len):
       zero = combine(zero, List_at(l, i))
       i = i + 1;
   return zero;


def list<int> List_int_append(list<int> l1, list<int> l2):
   int i = 0
```

```
   int len1 = List_len(l1)
   int len2 = List_len(l2)
   list<int> result = List__helper__(len1 + len2, -1)

   while (i < len1):
       List_replace(result, i, List_at(l1, i))
       i = i + 1;
   while (i < len1 + len2):
       List_replace(result, i, List_at(l2, i - len1))
       i = i + 1;

   return result;

########################################
# end of sPool List_int Standard Library #
########################################
```

8.2.2 string.sP

```
########################################
# start of sPool String Standard Library #
# Written by: Team Nautilus (Ankur, Yuma, Max, Etha) #
########################################

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

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

   return rev;


def int String_find(string haystack, string needle):
   int l1 = String_len(haystack)
```

```
    int l2 = String_len(needle)
    int answer = -1

    if (l1 < l2):
        answer = -1
    else
        int i = 0
        while (i < l1 - l2 + 1):
            if (String_eq(String_substr(haystack, i, i + l2), needle)):
                answer = i
                i = l1 - l2 + 1 # Hack: 'break' the loop
            else
                i = i + 1;
        ;
    ;

    return answer;

########################################
# end of sPool String Standard Library #
########################################
```

8.3 Some Fun sPool Programs

This section contains some fun algorithms and programs implemented in the sPool programming language. Some of the programs shown below are included as part of our demo during the project presentation, and all source files can be found in the demos directory.

8.3.1 Merge Sort

An implementation of merge sort in sPool:

```
# sort.sP
# Implements the merge sort algorithm in sPool
# Written by Team Nautilus (ankur, etha, max, yuma)
#
# Compile: The standard library needs to be imported while compiling this source
#          file. To do so, use the compile.sh script with the -stdlib flag.
# Example: When inside the src/ directory, run the following command to compile:
```

```
#           ./compile.sh -stdlib ../demos/sort.sP a.out
#           This will create an executable file named a.out in the src/ directory,
#           which can be simply run to execute the program.

# helper to print the string representation of an integer list
def quack printList(list<int> l):
   int len = List_len(l)
   int i = 0
   print("[ ")
   while (i < len):
       print(int_to_string(List_at(l, i)))
       print(" ")
       i = i + 1;
   println("]")
   return
;

# Returns the tail of the provided integer list, like cdr in other languages
def list<int> cdr(list<int> l):
   list<int> result = []
   int len = List_len(l)
   int i = 1
   while (i < len):
       result = List_int_append(result, [List_at(l, i)])
       i = i + 1
   ;
   return result;

# Given two sorted lists a and b, returns a new sorted list that contains
# all elements of a and b
def list<int> merge(list<int> a, list<int> b):
   int len_a = List_len(a)
   int len_b = List_len(b)
   list<int> result = []
   if (len_a == 0):
       result = b
```

```
   else if (len_b == 0):
       result = a
       else
           if (List_at(a, 0) < List_at(b, 0)):
               result = List_int_append(result, [List_at(a, 0)])
               result = List_int_append(result, merge(cdr(a), b))
           else
               result = List_int_append(result, [List_at(b, 0)])
               result = List_int_append(result, merge(a, cdr(b)))
           ;;;
   return result
;

# Given an integer list l, returns a new list whose elements are sorted in
# ascending order
def list<int> mergeSort(list<int> l):
   int len = List_len(l)
   list<int> result = []
   if (len <= 1):
       result = l
   else
       int mid = len / 2
       list<int> left = []
       list<int> right = []
       int i = 0
       while (i < mid):
           left = List_int_append(left, [List_at(l, i)])
           i = i + 1
       ;
       while (i < len):
           right = List_int_append(right, [List_at(l, i)])
           i = i + 1
       ;
       result = merge(mergeSort(left), mergeSort(right))
   ;
   return result;
```

```
# Testing mergesort: (the output obtained is as expected, with all lists
#                     sorted in the correct order)
printList(mergeSort([]))
printList(mergeSort([1]))
printList(mergeSort([1, 3, 2, 4, 5, 6, 7, 8, 9, 10]))
printList(mergeSort([-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]))
printList(mergeSort([1, -3, 2, -4, 5, -6, 7, -8, 9, -10]))
printList(mergeSort([10, -9, 8, -7, 6, -5, 4, -3, 2, -1]))
printList(mergeSort([1, 3, 2, 4, 5, 6, 437, 8, 9, 10, 1112, 12, 13, 14, 15, 16, 17,
18, 19, 20]))
printList(mergeSort([1, -3, 2, -4, 5, -6, 7, -8, 9, -10, 0, 100, -100, 1000, -1000,
10000, -10000]))
```

8.3.2 Idempotence

In addition to improving code efficiency in terms of runtime, the store functionality in sPool enables programmers to write programs that produce side effects just once. One such program in sPool is shown below, along with the expected results in the comments:

```
# idempotence.sP
# Demos the "store" functionality of sPool.
# Written by Team Nautilus (ankur, etha, max, yuma)
#
# Compile: Use the compile.sh script to compile this source file.
# Example: When inside the src/ directory, run the following command to compile:
#          ./compile.sh ../demos/idempotence.sP a.out
#          This will create an executable file named a.out in the src/ directory,
#          which can be simply run to execute the program.

# shared variables are declared on the heap; they can be mutated by threads and
# functions, and are accessible by all threads and functions
shared int global = 0

# getGlobal is an (int -> int) function with "store" enabled -- it caches
# the result of the function call and returns the cached result if the same
# argument is passed in again
```

```
def store int getGlobal(int x):
   return global
;

println("------------------------------")
println("--------WITH STORE------------")
println("------------------------------")

print("getGlobal(1) = ")
println(int_to_string(getGlobal(1))) # returns 0

println("changing global to 1")
global = 1

print("getGlobal(1) = ")
println(int_to_string(getGlobal(1))) # still returns 0, because the result was cached
in the first call

print("getGlobal(2) = ")
println(int_to_string(getGlobal(2))) # returns 1, because for argument x = 2, the
result was not cached
            # so the function is actually called instead of performing a cache lookup



# to see how a non-store function would behave, observe the following:

println("")
println("------------------------------")
println("---------W/O STORE------------")
println("------------------------------")

def int getGlobal2(int x):
   return global;

print("getGlobal2(1) = ")
```

```
println(int_to_string(getGlobal2(1))) # returns 1

println("changing global to 2")
global = 2

print("getGlobal2(1) = ")
println(int_to_string(getGlobal2(1))) # returns 2, because the result was not cached
in the first call
```

8.3.3 Store and Higher-order functions

Since sPool in its current state only supports automatic memoization for `(int -> int)` functions, programmers can write their own higher-order function that makes automatic memoization work for other function types. One such interesting example is shown below. Please view the actual source file in the demos directory of the final project for an ASCII art representation of function transformations that make this possible (due to formatting, it did not fit cleanly in the margins of this document). The usage of automatic memoization is shown in the previous example (section **8.3.2**) where the side effect is only observed once as a result of the function result being cached for successive calls with the same argument.

```
# storedStringFunc.sP
# an illustration of how to leverage the stored int->int function to other
# data types. Here we choose to write a HOF that converts a string->string
# function to a stored string->string function with the help of int<->string
# encoding and decoding functions.
# Written by Team Nautilus (ankur, etha, max, yuma)
#
# Compile: The standard library needs to be imported while compiling this source
#          file. To do so, use the compile.sh script with the -stdlib flag.
# Example: When inside the src/ directory, run the following command to compile:
#          ./compile.sh -stdlib ../demos/storedStringFunc.sP a.out
#          This will create an executable file named a.out in the src/ directory,
#          which can be simply run to execute the program.


# define the default alphabet to be lower case a-z, with the first
# character to be an unused character
string alphabet = "/abcdefghijklmnopqrstuvwxyz"
```

```
# convert a letter to its integer code
def int intOfLetter(string letter):
   return String_find(alphabet, letter);

# convert an integer code to its corresponding letter
def string letterOfInt(int code):
   return String_substr(alphabet, code, code + 1);

# convert a string of letters to an integer
def int convertStrToInt(string aStr):
   int idx = 0
   int lenStr = String_len(aStr)
   int result = 0

   while (idx < lenStr):
       result = result * 27 + intOfLetter(String_substr(aStr, idx, idx + 1))
       idx = idx + 1
   ;

   return result
;

# decompose an integer to its corresponding string
def string decompIntToStr(int aNum):
   string result = ""
   int moded = -1

   while (aNum != 0):
       moded = aNum % 27
       result = String_concat(letterOfInt(moded), result)
       aNum = aNum / 27
   ;

   return result
;
```

```
# makeStr2Store
# Purpose: make a string to string function a "stored" function
#          such that it has the access to previously computed values stored in
#          the "cache"
# Parameter: a function of type string->string where the strings are limited to
#            be consisted of lowercase English letters 'a' to 'z'; string of
#            length 7 or shorter are supported
# Return:    a function of type string->string that does the same thing as the
#            input function, but has the store feature.
def (string->string) makeStr2Store((string->string) aFunc):

   # utilizes the store function of int->int under the hood
   def store int storeHelperFunc(int x):
       string strX = decompIntToStr(x)
       string output = aFunc(strX)
       return convertStrToInt(output)
       ;

   # this is the stored version of the original function to be returned
   def string storedFunc(string inputStr):
       int intArg = convertStrToInt(inputStr)
       int result = storeHelperFunc(intArg) # calls the stored helper function
       return decompIntToStr(result)
   ;

   return storedFunc
;


# a simple function that prints its argument and returns its argument
def string printStrIdt(string x):
   println(x)
   return x;

# make the stored-version of printStrIdt
```

```
(string->string) stored_printStrIdt = makeStr2Store(printStrIdt)



# should print hello three times and print hellow once
println("----------Non-stored print---------------")
printStrIdt("hello")
printStrIdt("hello") # prints "hello" for the second time
printStrIdt("hello") # prints "hello" for the third time
printStrIdt("hellow") # prints "hellow"
println("-----------------------------------------")

println("----------Stored print-------------------")
# should print hello exactly once, followed by hellow
stored_printStrIdt("hello")
stored_printStrIdt("hello") # no side effect, as this is a "converted" store function
and result is cached for "hello"
stored_printStrIdt("hello") # again, does not print since result is cached
stored_printStrIdt("hellow")
println("-----------------------------------------")
```

Another simple higher-order function that flips the argument to a binary function that takes in two integers and returns an integer is shown below:

```
# flip.sP
# Demos sPool's higher order functions
# Written by Team Nautilus (ankur, etha, max, yuma)
#
# Compile: Use the compile.sh script to compile this source file.
# Example: When inside the src/ directory, run the following command to compile:
#          ./compile.sh ../demos/flip.sP a.out
#          This will create an executable file named a.out in the src/ directory,
#          which can be simply run to execute the program.

# flips the order of the arguments of an (int, int -> int) function
def (int, int -> int) flip((int, int -> int) f):
   def int flipped(int a, int b):
       return f(b, a);
```

```
   return flipped;

# a generic subtract function
(int, int -> int) subtract = lambda int (int x, int y): return x - y;
(int, int -> int) flippedSubtract = flip(subtract)

println(int_to_string(subtract(100, 1)))        # 99  == 100 - 1
println(int_to_string(flippedSubtract(100, 1))) # -99 == 1 - 100
```

8.3.4 Synchronizing Threads using Mutexes

Programmers have to be careful while using threads to work on regions in code that actively mutate shared, mutable memory. One such example is given below, where a shared variable x is being mutated concurrently by 50 threads. The purpose of this program is to show how one can use synchronization primitives like mutexes to prevent multiple threads from executing a critical region of code, and what can happen if this is not done. When calls to `Mutex_lock` and `Mutex_unlock` are uncommented, the output is deterministic and x is always 550 by the time the program execution finishes. However, when these calls are commented out, the output is non-deterministic since multiple concurrent threads will be mutating x simultaneously. See comments in the code below for more information.

```
# mutex.sP
# Demos sPool's threads and synchronization primitives (mutexes)
# Written by Team Nautilus (ankur, etha, max, yuma)
#
# Compile: Use the compile.sh script to compile this source file.
# Example: When inside the src/ directory, run the following command to compile:
#          ./compile.sh ../demos/mutex.sP a.out
#          This will create an executable file named a.out in the src/ directory,
#          which can be simply run to execute the program.

shared int x = 0 # a "global" variable from the perspective of the threads
mutex lock = Mutex()

def quack add11ToX():
   # Uncommenting calls to Mutex_lock and Mutex_unlock will cause the program to work
as expected
```

```
   # because the lock will be acquired and released before the next thread can access
the shared variable
   # This is critical when synchronizing access to shared mutable variables

   # This acts like a "barrier", allowing only one thread to access the critical
region at any given time
   # other threads will wait until the lock is released before they can access the
critical region
   # Mutex_lock(lock) # To demo, uncomment this to prevent concurrent access

   # reassigning to temp variable to increase surface area for race conditions
   int temp = x
   temp = temp + 11
   x = temp
   print("X = ")
   println(int_to_string(x))

   # Mutex_unlock(lock)
   return
;

int i = 0
int num_threads = 50
list<thread> ts = []

while (i < num_threads):
   List_insert(ts, i, { add11ToX() }) # invoke the function add11ToX in a new thread
   i = i + 1
;

# wait for all threads to be done
i = 0
while (i < num_threads):
   Thread_join(List_at(ts, i))
   i = i + 1
;
```

```
# when calls to lock and unlock are commented out, this will not always print
# 550 because of possible race conditions
print("Final value of x = ")
println(int_to_string(x))
```

8.3.5 Unblackedges

This is a simple implementation of a breadth-first search algorithm that starts at the edge of a given two-dimensional array and removes all connected “black pixels” that are represented by a 1 in the integer grid.

```
# unblackedges.sP
# Demos sPool's threads by implementing a simple BFS algorithm
# Inspired by CS40's unblackededges assignment at Tufts University
# Written by Team Nautilus (ankur, etha, max, yuma)
#
# Compile: Use the compile.sh script to compile this source file.
# Example: When inside the src/ directory, run the following command to compile:
#          ./compile.sh ../demos/unblackedges.sP a.out
#          This will create an executable file named a.out in the src/ directory,
#          which can be simply run to execute the program.

def bool is_black(int row, int col, list<list<int>> grid):
   return List_at(List_at(grid, row), col) == 1
;

def quack print_grid(list<list<int>> grid):
   int i = 0
   while (i < List_len(grid)):
       int j = 0
       while (j < List_len(List_at(grid, 0))):
           print(int_to_string(List_at(List_at(grid, i), j)))
           print(" ")
           j = j + 1
       ;
       println("")
```

```
      i = i + 1
   ;
   println("")
   return
;

def quack check_neighbor(int row, int col, list<list<int>> grid, list<list<int>>
queue):
   if (row < List_len(grid) - 1):             if (is_black(row + 1, col, grid)):
List_insert(queue, List_len(queue), [row + 1, col]);;
   if (col < List_len(List_at(grid, 0)) - 1): if (is_black(row, col + 1, grid)):
List_insert(queue, List_len(queue), [row, col + 1]);;
   if (row > 0):                              if (is_black(row - 1, col, grid)):
List_insert(queue, List_len(queue), [row - 1, col]);;
   if (col > 0):                              if (is_black(row, col - 1, grid)):
List_insert(queue, List_len(queue), [row, col - 1]);;
   return
;

list<list<int>> grid =
[[1,1,1,1,1,1,1,1],[1,0,0,0,0,0,0,1],[1,0,0,1,1,0,0,1],[1,0,1,0,0,0,1,1],[1,0,1,1,1,0,
1,1],[1,0,1,1,1,0,0,1],[1,0,0,0,0,0,1,0],[1,1,1,1,1,1,0,1]]
list<list<int>> queue = []

int row = 0
int rows = List_len(grid)
int cols = List_len(List_at(grid, 0))

while (row < rows):
   int col = 0
   while (col < cols):
      if ((row == 0 || row == rows - 1 || col == 0 || col == cols - 1) &&
is_black(row, col, grid)):
          List_insert(queue, List_len(queue), [row, col])
      ;
      col = col + 1
```

```
   ;
   row = row + 1
;

println("BEFORE UNBLACKING: ")
print_grid(grid)
while (List_len(queue) > 0):
   int row = List_at(List_at(queue, 0), 0)
   int col = List_at(List_at(queue, 0), 1)
   List_replace(List_at(grid, row), col, 0)
   check_neighbor(row, col, grid, queue)
   List_remove(queue, 0)
;

println("AFTER UNBLACKING: ")
print_grid(grid)
```