Seaflow Language Reference Manual

Rohan Arora, Junyang Jin, Ho Sanlok Lee, Sarah Seidman
ra3091, jj3132, hl3436, ss5311

1 Introduction

Seaflow is an imperative language designed to address the asynchronous event conundrum by supporting some of the core principles of ReactiveX and reactive programming natively. Modern applications handle many asynchronous events, but it is difficult to model such applications using programming languages such as Java and JavaScript. One popular solution among the developers is to use ReactiveX implementations in their respective languages to architect event-driven reactive models. However, since Java and JavaScript are not designed for reactive programming, it leads to complex implementations where multiple programming styles are mixed-used.

Our goals include:

1. All data types are immutable, with the exception being observables, no pointers
2. The creation of an observable should be simple
3. Natively support core principles in the ReactiveX specification

2 Lexical Conventions

2.1 Comments

We use the characters /* to introduce a comment, which terminates with the characters */. Any tokens between /* and */ are considered part of a comment and are not parsed.

```
/*
* This is a comment
* with multiple lines
*/

int x = 4;    /* This is also a comment */
```

2.2 Identifiers

Identifiers must begin with a lowercase letter and should only contain ASCII letters, digits and underscores. For observables, the identifier should be prepended with the “$” symbol.

Identifiers: `['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']*`

Observable identifiers: `'$'['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']*`

2.3 Keywords

Seaflow keywords include:

`char, else, float, if, int, null, return, struct, void, ($)`

2.4 Literals

Literals represent strings or one of Seaflow's primitive types: float, char, and int.
float: `([0-9]+.[0-9]+) | ([0-9]+)`
int: `[0-9]+`
char: `['a'-'z'] | ['A'-'Z']`
string: `("[^"'\\]*(\\.[^"\\]*)*")|('[^"'\\]*(\\.[^'\\]*)*')`

2.5 Struct

A struct is a combined representation of constituent primitives. Structs cannot have observables as members. Struct declarators must begin with a capital letter; otherwise they have the same rules as identifiers for what characters are allowed. They are defined as follows:

```
struct <declarator> {[type-list parameter-list]};
```

Once a struct is created, it is immutable.

```
struct Foo {
    int field1;
    char field2;
    char[] name;
};

struct Foo f = { 1, 'c', "name" };
```

2.6 Array

An array is a collection of primitives. (Arrays cannot contain observables.) Arrays are declared as follows:

```
<type>[] <identifier>;
```

Once an array is declared, it is immutable. To initialize an array with values, include a comma-separated list of expressions between square brackets.

```
int[] arr = [ 1, 2, 3, 4, 5 ]

arr[0] = 10;    /* this is not allowed */
```

Arrays have a fixed `length` attribute, which can be accessed like this:

```
int[] arr = [ 1, 2, 3, 4, 5 ]
int len = length(arr);    /* len = 5 */
```

3 Expressions and Statements

3.1 Expressions

3.1.1 Primary Expressions

Followings are considered primary expressions and they are evaluated left to right:

<primary-expression>:
    <identifier>
    <literal>
    <primary-expression>[<expression>]   # array indexing
    <primary-expression>.<member-of-structure>
    <function-identifier>(<expression-list>) # function call

3.1.1 Operators

The four basic arithmetic <operator> are
    + addition
    - subtraction
    * multiplication
    / division.
These are all left-associative binary operators. Multiplication and division have a higher precedence than addition and subtraction.
The relational <operator> are
    == equality
    != inequality
    < less than
    > greater than
    <= less than or equal to
    >= greater than or equal to.
These are binary operators and have lower precedence than all arithmetic operators.
The assignment operator is =, and is right associative. On the left hand side must be either an observable or the declaration of an identifier.

<expression>:

<expression> <operator> <expression>

3.1.2 Conditional expression

Conditional expression in Seaflow looks like this:

<conditional-expression>:
    if (<expression$_c$>) <expression$_1$> else <expression$_2$>

If <expression$_c$> is evaluated to true, the <conditional-expression> is evaluated to <expression$_1$>, and otherwise it is evaluated to <expression$_2$>

<conditional-expression> can be chained:
    if (<condition$_c$>) <expression$_1$> else <conditional-expression>

And therefore following example is possible:

```
char letterGrade = if (score > 90) 'a' else if (score > 80) 'b' else 'f';
```

3.4 Statements

3.4.1 Declaration

Variables are declared with the following syntax:

<declaration>:
    <type-specifier> <identifier> = <expression>

```
int a = 5;
```

<identifier> must be unique to its scope, and all non-observable variables must be initialized at the time of declaration.

3.4.2 Immutability

All non-observables are immutable after they are created. Also, all variables once they are initialized cannot be reassigned with a different value.

```
int a = 0;
```

```
a = 4;   /* This line yields a compile-time error */
```

3.4.3 Observable re-assignment

Observables can be reassigned using following syntax:

<observable-identifier> = <expression>

4 Functions

4.1 Declaration

A Seaflow function is similar to a C function. Each function takes a list of fixed-type arguments and returns a fixed-type value. The declaration consists of two parts: a function declarator and a function body. All function names must start with a lowercase letter.

Function declarator:

<return-type> declarator ([type-list parameter-list])

Function body:

{declaration-list statement-list}

Simple example:

```
int add(int a, int b) {
    int c = a + b;
    return c;
}
```

4.2 Scope rules

Identifiers declared outside of any function are visible throughout the program after their declaration. The scope of identifiers declared in the declaration of a block is limited to within that block. It is an error to redefine an identifier that already exists in the current scope. Observables are all visible at the global level.

4.3 Higher Order Functions

Seaflow supports higher-order functions. Higher order functions can be defined either in-place or from declaration. Passed-in functions must be typed on both the upstream and downstream values.

Higher order functions example:

```
int function(int x, (int)->(int) func) {
    return func(x);
}

function(10, (int x)->{ return x + 10; });
```

4.3.1 Anonymous Function

One can declare an anonymous function by following this format:

(input) -> {statements}

and pass it into higher order functions as a parameter.

Anonymous function example:

```
function(10, (int x)->{ return x + 10; });
```

The type signature of anonymous functions will be inferred at the compile time.

5 Observables

Observables in Seaflow are special types of objects that can internally hold another object. When the state of the observable changes, the observable sends notification to all of its “observers” that are subscribed to it. Observables can be defined only in the global scope.

5.1 Declaration

One can declare an observable by using a observable identifier that starts with a dollar sign, $, followed by a normal identifier

<declaration>:

<type-specifier> <observable-identifer>

```
int $a;
```

An observable can have an initial value at the declaration time. This can be expressed in following form:

<type-specifier> <observable-identifer> = <non-obsevable-expression>;

```
int $a = 0;
```

We define <observable-expression> as an expression that is evaluated to an observable, and <non-observable-expression> as an expression that is evaluated to a normal non-observable.

5.2 Subscription

An <observer> can receive notifications from an <observable> by subscribing to the observable. Observer can be a function with the same input type, or an observer of the same type.

<observer>:
<function>
<observable>

<subscription>:
subscribe(<observable>, <observer>)

```
int $a;

void observer(int num) {
    print(num);
}

subscribe($a, observer);
```

5.3 Methods

5.3.1 Map

An <observable> of type T can call the map function as follows:

map($T obs, (T)->X func))

Where the higher-order function passed to the map function takes an argument of the same type T, with the return value being of any type X. The map function returns a new observable with type X. The function `func` is called for each upstream value and the returned value will be passed to the downstream.

In addition, initializing an observable to some function of another observable is equivalent to calling the map function.

```
int $c = map($b, (int x)->{ return x + 10; });

/* or equivalently */

$c = $b + 10;
```

5.3.2 Combine

An <observable> of type T can call the combine function as follows:

combine($T obs, $S obs, (T, S)->X func)

Where the first argument to the combine function is another observable of type S. The second argument is a function which takes two observables of types T and S, with the return value being any type X. The combine function returns a new observable with type X. The function `func` is called for each upstream value and the returned value will be passed to the downstream.

In addition, initializing an observable to some expression containing two observables is equivalent to calling the combine function.

```
int $d = combine($b, $c, (int x, int y)->{return x + y;});

/* or equivalently */

$d = $b + $c /* type must match when shorthand version is used */

$e = $f + $g * $h - $m / $k /* can chain combine calls using shorthand */
```

5.3.3 Complete

An <observable> can call the complete function as follows:

complete($T obs)

Any subscriptions that the observable had will be removed.

```
complete($d);
```

5.4 Operator overloading and Observable expressions

An <observable-expression> is an expression that contains one or more observables. An <observable-expression> is evaluated to an observable. Any other expression--all expressions that we discussed so far--is a <non-observable-expression>.

<observable-expression>
    <observable-expression> <operator> <non-observable-expression>
    <non-observable-expression> <operator> <observable-expression>
    <observable-expression> <operator> <observable-expression>

The base type of the observable-expression, the operator, and the non-observable-expression must be compatible.

The “=” operator with an observable on the left hand side has different behavior depending whether the right hand side of the “=” operator is an <observable-expression> or <non-observable-expression>

<observable> = <non-observable-expression>

This is an assignment statement that will set the internal object of the <observable> to a newly evaluated value of <non-observable-expression>.

<observable> = <observable-expression>

This is an assignment statement that will replace the reference of the observable identifier with the observable that is returned by the <observable-expression>.

6 Conversions

Relational operators cause implicit conversion between types. When operands of two types are compared, the one with lesser precision is converted to the type of the one with greater precision. For example:

```
int i = 5;
char c = 'a';
float f = 5.5;

/* c is converted to integer */
c == i

/* i is converted to float */
i == f

/* c is converted to float */
c == f
```

The three numeric data types can be implicitly cast to each other. Integers and chars can be cast without loss of precision to floats, whereas conversion of a float to an integer is done by truncating after the decimal point. Conversion of a float or integer with a value greater than 127 to a char will cause incorrect results.

```
int a = 5;
char b = 'a';
float c = 1005.5;

/* No loss of precision */
int d = c;
float e = a;
float f = b;

/* No error but value is truncated after decimal point */
int g = c;

/* Valid syntax but character will contain incorrect value */
char h = c;
```