Splat

By Samson Danziger & Oliver Steptoe

1 Language Reference

1.1 Types

`number`	Stored as float. Can be positive or negative.
`boolean`	Can be true or false.
`string`	Alphanumeric strings. Cannot yet contain spaces. Surrounded by “”.
`list`	Lists can contain about 12000 items.
`function`	Functions can accept only one argument. For multi-argument functions, the function should just return another function.

1.2 Type Definitions

Defining numbers, booleans and strings is done by following the type by the name of the variable. For example: `` `number n` ``.

While defining lists you must also define the content type of the list, for example: `` `number list l` `` is a list `` `l` ``, containing numbers. This is required but doing so makes no difference to the execution of the program whatsoever.

For functions, the function definition is of the form:

```
function [return-type] [name] ([parameter-type] [parameter]) {
	[function_body]
}
```

For example:

```
function number t1 (number n) {
	n
}
```

defines a function called `` `t1` `` which takes a number as its argument, and returns a number which in this case is the number passed as the parameter.

1.3 Main language

In Splat, the last thing to be evaluated is returned.

1.3.1 Comments

In Splat comments are surrounded by `` `<|` `` and `` `|>` ``. You cannot run files which contain only comments.

```
<| This is a comment |>
```

1.3.2 Functions

There are no `for` or `while` loops in Splat, instead functions can be used recursively.
Here is an example of a function that adds 5 to its argument.

```
function number add5 (number n) {
	n + 5
}
```

Here is an example of a function counts down from the given argument, using recursion.

```
function number countdown (number n) {
	if n < 0 {
		0
	} else {
		let [p = showln n] {
			countdown #(n - 1)
		}
	}
}
```

Functions can also return other functions. For example, the following function returns a function which returns a function which returns the result of multiplying the arguments all the arguments together.

```
function number multiplier (number a) {
    function number i (number b) {
        function number j (number c) {
            a * b * c
        }
    }
}
```

Once a function has been defined, you can use it by separating the arguments with `#`.

```
multiplier #2 #3 #4        => 24
```

1.3.3 If-Else Statements

An If-Else statement has 3 arguments, the first argument should evaluate to a boolean, if it evaluates to `true`, the second argument will be returned, otherwise the third argument will be returned.

```
if (2 > 3) {
	"Cats"
} else {
	"Dogs"
}
```

In this situation, the string “Dogs” would be returned.

1.3.4 Let Statements

A Let statement consists of 2 sections, the first is where a variable is defined, and the second where it can be used.

```
let [a = 3] {
	showln a
}
```

Here, we let `a` be set to 3, and then print `a`, which of course prints (and returns) 3. The result of this would be the equivalent of `showln 3`.
The variable definition is evaluated immediately, so the following statement prints "3 3", but returns only 3.

```
let [a = show 3] {
       showln a
}
```

First `show 3` is evaluated, which prints "3 " to stdout, and returns 3. `a` is set to 3, and prints "3\n" to stdout, which results in "3 3" and a newline, but returns 3.

You can even define a function in the variable definition section. In the following program, we define the f as a function that adds 5 to its argument, and then `add5` as `f`, where `add5` is accessible in the body of the let statement. This program returns 8.

```
let [add5 = function number f (number a) {
    a + 5
}] {
    add5 #3
}
```

When using recursion in a function in a let block, remember to use the initial function name in recursive calls, not the variable that is being set. For example:

```
let [add5 = function number f (number a) {
    if a > 10 {
        f #(a % 8)
    } else {
        a + 5
    }
}] {
    add5 #30
}
```

1.4 Predefined functions

1.4.1 Casting

Strings can be cast as numbers using the `num` function.

```
num "5" => 5
```

1.4.2 Streams

In Splat, stream functionality is wrapped as a list. Using the `stdin` keyword loads the stream where it can be used as a string list.

1.4.3 Generic Operations

In Splat, strings can be split using the `split` keyword. When given a string as argument, a list of strings is returned, having split the argument on the spaces.

```
split "1 2"   => [1 2]
```

`show` and `showln` can be used to print to stdout. Both functions return the argument they are given. `show` prints the argument, followed by a space, whereas `showln` prints the argument followed by a newline.

```
show 3                      => prints "3 " to stdout
show "splat"          => prints "splat " to stdout
showln 4              => prints "4\n" to stdout
showln "splat"        => prints "splat\n" to stdout
showln (2 < 3)        => prints "true" to stdout
```

1.4.4 List Operations

In Splat, an empty list can be created using `[]`. One can add elements to a list using the `::` operator.

```
1 :: []             => [1]
1 :: 2 :: 3:: []    => [1 2 3]
```

There is no intention in the language to be able to have lists of lists, so any code towards this result is untested.

There also exists a function `lstend` which checks if a list is empty.

```
lstend []           => true
lstend 5::4::[]     => false
```

Head and tail can be used to operate on lists. `head` returns the first element of a list. `tail` returns the rest of the list (excluding the `head`).

```
head (1 :: [])              => 1
head []                     => []
tail (1 :: [])              => []
tail []                     => Error
```

2 Appendix

2.1 Language Quirks

There are a few language quirks in Splat, most of which can easily be overcome. These are not bugs, they are features.

2.1.1 Environments in Recursive Calls

Sometimes the environments in recursive calls are not passed through correctly, which results in each subsequent function call having the incorrect arguments. The solution for this is to define the argument to the parameters or the recursive function in let statements before use.

2.1.2 Semicolon separated statements

Splat allows for statements separated by semicolons (`;`), in the body of function definitions, if-else statements and let statements. Ideally all statements would evaluate, but only the last one would be returned. This is mostly the case, but sometimes statements to not fully evaluate. To solve this, you can save the statement to a variable in a let statement.

2.1.3 List contents

List definitions were intended to strictly define the contents of lists, but this doesn't really happen.

2.1.4 Negative Space

If you intend to subtract one number from another, you must have a space to the right of the minus sign.

```
2-3           => Error
2 - 3         => -1
```

This is because `` `2-3` `` is parsed as `` `2` `` followed by `` `-3` ``, with no operation being applied, whereas `` `2 - 3` `` is parsed as `` `2` `` followed by `` `-` `` followed by `` `3` ``.

2.2 List of Operators

Operator	**Description**	**Operator**	**Description**
`true`	Boolean true	`showln`	Print with a new line
`false`	Boolean false	`show`	Print followed by a space
`stdin`	Interpreted as a list containing the input stream	`split`	Split a string on the spaces
`::`	Construct a list	`head`	Return the head of a list
`[]`	Empty list	`tail`	Return the tail of a list
`and`	Boolean and	`num`	Cast a string as a number

`or`	Boolean or	`not`	Boolean not
`+`	Addition	`<=`	Less than or equal
`-`	Subtraction	`>=`	Greater than or equal
`*`	Multiplication	`==`	Equal to
`/`	Division	`!=`	Not equal to
`^`	Power of	`<`	Less than
		`>`	Greater than

2.3 Examples

2.3.1 Hello World

Print and return "`HelloWorld`".

```
showln "HelloWorld"
```

Semicolon separated statements can only be used inside, let, if-else, and function statements, so to print "Hello World", you could do:

```
let [a = 0] {
	show "Hello";
	showln "World"
}
```

2.3.2 Adder

A function to add two numbers, a and b. You can see this function is being called with the arguments `2` and `3`, so this would return 5.

```
function number adder (number a) {
	function number a2 (number b) {
		a + b
	}
} #2 #3
```

2.3.3 Blastoff!

Blastoff is an extension to the `countdown` example listed under *functions.* This example defines a recursive function inside a let statement for later use. It also makes use of the `show` and `showln` functions to print text on the same line.

```
let [ f = function number countdown (number n) {
	if n == 0 {
		let [print = showln "Blastoff"] {
			0
		}
	} else {
		let [print = showln n] {
			countdown #(n - 1)
		}
```

```
        }
}] {
        let [s = 0] {
                show "Launching";
                show "in";
                show "T";
                showln "minus";
                f #5
        }
}
```

2.3.4 Fizzbuzz

Fizzbuzz is a function that counts from `c` to `max`, and prints "fizz" if divisible by 3, "buzz" if divisible by 5, and "fizzbuzz" if divisible by 3 and 5. If none of those requirements are met, the number if printed. If you gave the function the arguments 6 and 1, you would get: 1, 2, fizz, 4, buzz. In the following example, you can see that the function is given the arguments 100 and 1, so it would run fizzbuzz for the numbers 1 to 99 inclusive.

```
function number fizzbuzz (number max) {
    function number looper (number c) {
        if (c == max) {
            0
        } else {
            let [n = c + 1] {
                if ( c % 15 == 0 ) {
                    let [a = showln "fizzbuzz"] {
                        looper #n
                    }
                } else {
                    if ( c % 5 == 0 ) {
                        let [a = showln "buzz"] {
                            looper #n
                        }
                    } else {
                        if ( c % 3 == 0 ) {
                            let [a = showln "fizz"] {
                                looper #n
                            }
                        } else {
                            let [a = showln c] {
                                looper #n
                            }
                        }
                    }
                }
            }
        }
    }
} #100 #1
```

2.3.5 Fizzbuzz with Streams

The following example is a fizzbuzz program which outputs one of: the number, "fizz", "buzz", or "fizzbuzz", depending on input from a stream.

```
function number fizzbuzz (number list splat) {
    if (head splat) == [] {
        0
    } else {
        let [h = (num (head splat)) ] {
            if (h % 3 == 0) and (h % 5 == 0) {
                let [ s = (showln "fizzbuzz") ] {
                    fizzbuzz #(tail splat)
                }
            } else {
                if h % 3 == 0 {
                    let [ t = (showln "fizz") ] {
                        fizzbuzz #(tail splat)
                    }
                } else {
                    if h % 5 == 0 {
                        let [ u = (showln "buzz") ] {
                            fizzbuzz #(tail splat)
                        }
                    } else {
                        let [ v = (showln h) ] {
                            fizzbuzz #(tail splat)
                        }
                    }
                }
            }
        }
    }
} #stdin
```

Reversing a Finite Stream

```
function number reverse (number list splat) {
    function number r2 (number list ret) {
        if lstend splat {
            ret
        } else {
            let [h = head splat] {
                reverse #(tail splat) #(h::ret)
            }
        }
    }
} #stdin #[]
```

Mapping a function onto a list

Here you can see the `reverse`, `map`, and `add5` functions are defined in `let` statements, and the the map function is used in the body of the final `let` statement.

```
let [reverse =
    function number list r (number list lst) {
        function number list r2 (number list ret) {
            if lstend lst {
                ret
            } else {
                let [h = head lst] {
                    r #(tail lst) #(h::ret)
```

```
                }
            }
        }
    }
] {
    let [map =

        function number list mapf (number list l) {
            function number list m2 (number list l2) {
                function number list m3 (function number (number) f) {
                    if lstend l {
                        reverse #l2 #[]
                    } else {
                        let [nextval = f #(head l)] {
                            mapf #(tail l) #(nextval :: l2) #f
                        }
                    }
                }
            }
        }
    ] {

        <| actual body |>
        let [add5 =
            function number f (number n) {
                n + 5
            }
        ] {
            map #(1::2::3::4::5::[]) #[] #add5
        }
    }
}
```