Fall 2017 COMS4115
Programming Languages & Translators

Strux Language Reference Manual

Josh Bartlett (jcb2254), Fredrick Tam (fkt2105),
Sophie Stadler (srs2231), Millie Yang (my2440)

Introduction 9

Lexical Elements 9

- Identifiers 9
- Keywords 9
- Whitespace 9
- Comments 10

Operators and Expressions 10

- Assignment Operator 10
- Arithmetic Operators 11
- Comparison Operators 11
- Logical Operators 12
- String Concatenation 12
- Operator Precedence 12
- Order of Evaluation 13

Statements 13

- Expression Statements 13
- Declaration Statements 13
- Control Flow Statements 13
  - Loops 13
    - For Loops 14
    - Enhanced For (forEach) Loops 14

While Loops 15
Conditionals 15
Break, Continue 16
Data Types **16**
Primitives 16
num 16
string 16
bool 16
Built-In Data Structures 16
Stack 16
Initializing an instance of a Stack 17
Library Functions 17
Peek 17
Pop 17
Push 18
isEmpty 18
Size 18
Queue 19
Initializing an instance of a Queue 19
Library Functions 19
Peek 19
Enqueue 20
Dequeue 20
isEmpty 20
Size 21
LinkedList 21
Initializing an instance of a LinkedList 21

Library Functions 21
Add 21
Remove 22
Check Empty 22
Get Size 22
Array 22
Array Declaration 22
Library Functions 24
Length 24
Find 24
QuickSort 24
BSTree 26
BSTree Declaration 26
Library Functions 27
Add 27
Remove 27
Contains 28
Functions **28**
Built-In 28
main() 28
show() 28
User-Defined 30
Style Guide **31**

Introduction

Lexical Elements

Identifiers

An identifier is a unique sequence of characters that are used to identify variables and functions. Identifiers can contain letters, numbers, and the underscore character. Additionally, identifiers are case-sensitive. A valid identifier adheres to the following rules:

1. At least 1 character long
2. Begins with a letter
3. Isn't equal to one of the reserved keywords

Keywords

Keywords are reserved words that each have some unique meaning when compiling. Keywords can not be used as identifiers or reassigned.

`num`	`string`	`bool`	`for`
`while`	`in`	`break`	`continue`
`true`	`false`	`void`	`if`
`elif`	`else`	`LinkedList`	`ListNode`
`Stack`	`Queue`	`Array`	`return`
`and`	`or`	`forEach`	`main`
`show`	`TreeNode`	`null`	`BSTree`
`not`	`new`		

Whitespace

Whitespace is largely ignored in Strux. Other than within string literals, whitespace is only used to separate different tokens. Therefore, these two statements are actually produce the same result after being compiled:

```
num addTwo(num a, num b) {
    return a + b;
}
```

```
num       addTwo    (num a  ,num b) {     return a+  b;}
```

A space *is* required after:

- The `return` keyword, before the value that is returned (if any).
- The `new` keyword, after an instance of an object is initialized.
- The return type of a variable when defined in an expression.
- The return type of a function in a function signature.

Do *not* put a space between:

- The type of values in an array and the brackets (`[]`) used to instantiate it
    - Example: `num[] arr = {1, 2, 3, 4};`

Comments

Anything in a comment will be completely ignored by the compiler. Strux does ***not*** have a special syntax for single-line comments, all comments are contained within `:(` and `):`.

```
:( This is a comment ):
```

Operators and Expressions

Assignment Operator:

Operator	Type	Associativity
=	Assignment	Right -> Left

Strux uses the standard assignment operator (=), to store the value of the right operand to the variable of the left operand of the same type. The left operand cannot be a literal (`string` or `num` literal) value and variables on the left cannot be named starting with numbers.

Example:

`num myAge = 21;` ✓ `num "myAge" = 21;` □

`string myName = "Kennedy";` ✓ `bool 1true = true;` □

Arithmetic Operators:

Assuming num x = 100 and num y = 20

Operator	Type	Associativity	Example
+	Addition	Left -> Right	`x + y = 120`
-	Subtraction	Left -> Right	`x - y = 80`
*	Multiplication	Left -> Right	`x * y = 2000`
/	Division	Left -> Right	`x / y = 50`
%	Modulo	Left -> Right	`x % y = 0`

Comparison Operators:

Assuming `num x = 50` and `num y = 20`

Operator	Type	Associativity	Example
==	Equal To	Left -> Right	`(x == y)` returns false
!=	Not Equal To	Left -> Right	`(x != y)` returns true
>	Greater Than	Left -> Right	`(x > y)` returns true
>=	Greater Than Or Equal To	Left -> Right	`(x >= y)` returns true
<	Less Than	Left -> Right	`(x < y)` returns false
<=	Less Than or Equal To	Left -> Right	`(x <= y)` returns false

Logical Operators:

Assuming `bool x = true` and `bool y = false`

Operator	Type	Associativity	Example
`and`	Logical AND	Left -> Right	`(x and y)` returns false
`or`	Logical OR	Left -> Right	`(x or y)` returns true
`not`	Logical NOT	Right -> Left	`not x` returns false

String Concatenation:

Strings can be concatenated with the use of the + operator to create a new string value, and both left and right operands must be of strings as well.

```
string word = "Strux";

string sentence = word + " is awesome!";

:( sentence equal to "Strux is awesome!" ):
```

Operator Precedence:

Expressions can have multiple operators, for example `(x - y) * (x % y)`.
In these situations, operators are executed based on their level of precedence. List below arranges operators in order of precedence; from highest precedence to lowest.

1. Multiplication and Division expressions
2. Addition and Subtraction expressions
3. Greater Than, Less Than, Greater Than or Equal, and Less Than or Equal To expressions
4. Equal To and Not Equal To expressions
5. Logical NOT expressions
6. Logical AND expressions
7. Logical OR expressions
8. Assignment expressions

Order of Evaluation:

If we have a complex expression, it will be evaluated by starting with the leftmost subexpression. For example, in:

```
(( C() % D() ) * ( E() + Z() )
```

where C, D, E and Z are functions, C() will be called first, followed by D(), E() and Z(). Operator precedence will be ignored in this case.

Statements

Expression Statements

An expression statement is one that can be executed by Strux. Expressions are terminated with a semicolon, and include method invocations, value assignments, and creation of data structures. Some examples:

```
LinkedList myList = new LinkedList();
show(myList.isEmpty());
string greeting = "hello world";
```

Declaration Statements

Declaration statements are used to declare a new variable. They are comprised of its type, its name, and, optionally, its value. A value is assigned with the equals operator (=). One can declare multiple variables of the same type in one declaration. Declaration statements are terminated with a semicolon.

```
num five = 5;
num wordCount;
string missionStatement = "Strux rocks!";
bool isTired, isHungry, isThirsty;
```

Control Flow Statements

Control flow statements disrupt the linear evaluation of code. Conditionals, loops, and keywords are used by Strux to introduce specific flow.

Loops

Loops are used to execute a section of code multiple times. Strux includes three types of loops: for loops, enhanced for loops, and while loops.

For Loops

For loops are used to execute a block of code until a condition is satisfied. The format is as such:

```
for (initialization; termination; increment/decrement) {
    :( Code goes here ):
}
```

The termination expression above must evaluate to a boolean. When the loop is entered, the initialization is called and checked against the termination condition. Then, the code inside the loop is executed and the initialization value incremented on each iteration. The loop finishes when the termination expression returns false.

An example:

```
for (num i = 1; i <= 10; i++) {
    show(i);                          :( Prints the numbers 1-10 ):
}
```

Enhanced For (forEach) Loops

Enhanced for loops are used to iterate over items in a data structure, including arrays, linked lists, stacks, and queues. They are useful because they eliminate the need for a counter and terminator as in a standard for loop, and instead provide direct access to the structure's data. The syntax follows this pattern:

```
forEach item in iterable {
    :( execute this code }:
}
```

An example:

```
forEach node in myLinkedList {
    show(node);              :( Prints all nodes in a LinkedList ):
}
```

While Loops

While loops are used to iterate over a block of code until a condition is being evaluated as false. The syntax is such:

```
while (expression) {
    :( execute this code ):
}
```

The expression above must evaluate to a boolean value. The code contained within the braces will execute until the expression returns false. An example:

```
num i = 1;
while (i <= 10) {
    show(i);                          :( Prints the numbers 1-10 ):
}
```

Conditionals

Strux uses `if-else` and `if-elif-else` expressions to introduce conditional evaluation. In each of these statements, code within the required braces (`{}`) will evaluate only if the given expression is true. Conditional statements must be enclosed in parentheses. Below, an if-else statement:

```
bool october = true;
if (october == true) {
    show("It's October!");
} else {
    show("It isn't October.");
}
```

An `if-elif-else` statement presents the opportunity to introduce more (infinite, in fact) conditional statements.

```
num temp = 65;
if (temp > 80) {
    show("It's hot!");
} elif (temp < 45) {
    show("It's cold!");
} elif (temp < 10) {
    show("It's freezing!");
} else {
    show("It's nice out.");
}
```

Break, Continue

The `break` keyword stops iterating code immediately and exits the looping condition. Using `break` outside of a loop will throw an error.

The `continue` keyword skips the present iteration of a looping condition, and enters the next iteration. Using `continue` outside of a loop will throw an error.

Data Types

Strux is a typed language. Type must be specified when a variable is declared, and is immutable.

Primitives

num

Strux represents all digits, whether integers or decimal values, using `num`. A `num` is a 64-bit value.

string

A `string` is a sequence of ASCII characters enclosed by double quotes ("). Calling `.length` on a `string` returns the number of characters in the string. Characters in a string can be accessed much like array elements, with `str[0]` returning the first character in string `str` as a `string`.

bool

A variable of type `bool` represents the logical value `true` or `false`.

Built-In Data Structures

Stack

Stack is a data structure that represents LIFO (Last-in-first-out) operations on stack of objects.

Initializing an instance of a Stack

Stacks can be initialized using one of two constructors. The type is specified after two colons (`::`); this pattern is adopted by Queues and LinkedLists as well.

```
Stack::type emptyStack = new Stack();
```

The second initializes a stack filled with the values of an array. This array must be composed of `num` or `string` values, but not both.

```
Stack::num stack = new Stack({1, 2, 3});
```

Library Functions

There are several builtin functions for manipulating a stack.

Peek

To look at the top element of the stack, use peek(). This method retrieves, but does not remove the top of the element in the stack. If the stack is empty, this function returns null.

```
stack.peek();                                   :( returns 1 ):

stack.peek();                                   :( returns 2 ):

stack.peek();                                   : ( returns 3 ):

stack.peek();                                   :( returns null ):
```

Pop

To look at the top element of the stack and remove it from the stack, use `pop()`. This function retrieves value of top most element of stack and removes it from stack. If the stack is empty, this mefunctionthod returns null.

```
Stack::num stack = new Stack({1, 2, 3});
stack.pop();                                          :( returns 1 ):
stack.pop();                                          :( returns 2 ):
stack.pop();                                          :( returns 3 ):
stack.pop();                                       :( returns null ):
```

Push

To add items to the top of the stack, use `push(num` or `string)`. A new element is created and added to the top of the stack. This new element has value that was passed in as the parameter. Method does not return anything.

```
stack.push(5);
```

isEmpty

To check whether there are any elements left in our stack, we call `isEmpty()`. Method returns true when stack is empty, and false when stack is not.

```
Stack::num stack = new Stack({1, 2, 3});
stack.isEmpty();                                   :( returns false ):
Stack::num stackTwo = new Stack();
stackTwo.isEmpty();                                 :( returns true ):
```

Size

Calling `size()` returns the number of elements in the stack.

```
Stack::num stack = new Stack();
stack.size();                                          :( returns 0 ):
Stack::num stackTwo = new Stack({1, 2, 3});
stack.size();                                          :( returns 3 ):
```

Queue

Queue is a data structure that represents FIFO (first-in-first-out) operations on a list of objects.

Initializing an instance of a Queue

Queues can be initialized using one of two constructors.

```
Queue::num emptyQueue = new Queue();
```

The second initializes a queue filled with the values of an array. This array must be composed of `num` or `string` values, but not both.

```
Queue::num queue = new Queue({1, 2, 3});
```

Library Functions

There are several builtin functions for manipulating a queue.

Peek

To look at the head of the queue, use peek(). This function retrieves, but does not remove the element in the head of the queue. If queue is empty, this function returns null.

```
Queue::num queue = new Queue({1, 2, 3});
queue.peek();                                          :( returns 1 ):
queue.peek();                                          :( returns 1 ):
```

Enqueue

To add items to the tail of the queue, use `enqueue(num or string)`. A new element is created and added to the tail of the queue. This new element contains value that was passed into the parameter. Function does not return anything.

```
Queue::num queue = new Queue({1, 2, 3});
queue.enqueue(4);
```

At this moment, the queue contains 4 elements: 1,2,3,4. 1 is the head of the queue, and 4 is the tail of the queue.

Dequeue

To remove items from the head of the queue, use `dequeue()`. This function looks at the head of the queue. If the queue is empty, function returns null. Otherwise, function returns the value of the head of the queue.

```
Queue::num queue = new Queue({1, 2, 3});
queue.dequeue();
```

At this moment, the queue contains 2 elements: 2,3. Element with value 1 was removed from the queue since it was the head of the queue.

isEmpty

To check whether there are any elements left in our queue, we call `isEmpty()`. Function returns true when queue is empty, and false when queue is not.

```
Queue::num stack = new Queue({1, 2, 3});
queue.isEmpty();                                    :( returns false ):
Queue::num queueTwo = new Queue();
queueTwo.isEmpty();                                  :( returns true ):
```

Size

Calling `size()` returns the number of elements in the queue.

```
Queue::num queue = new Queue();
queue.size();                                          :( returns 0 ):
Stack::num queueTwo = new Queue({1, 2, 3});
queueTwo.size();                                       :( returns 3 ):
```

LinkedList

A LinkedList is comprised of `ListNode` objects, which contain data (either a `num` or `string`), and a reference to the next `ListNode`.

Initializing an instance of a LinkedList

Linked lists can be initialized using one of two constructors. The first produces an empty `LinkedList` object:

```
LinkedList::type emptyList = new LinkedList();
```

The second initializes a `LinkedList` filled with the values of an array. This array must be composed of `num` or `string` values, but not both.

```
LinkedList::num numList = new LinkedList({1, 2, 3, 4});
```

Library Functions

There are several builtin functions for manipulating a Linked List.

Add

To append items to the tail of the Linked List, use `.add(num or string)`. A node is created from the value passed into `add`, and is appended to the end of the list. Returns `true` if the item is appended successfully.

```
numList.add(5);                                    :( returns true ):
emptyList.add("not empty anymore");                :( returns true ):
```

Remove

To remove an item from the list, call `.remove(num or string)`. The first node containing this value is removed. If there are multiple nodes with this value, all but the first remain. Returns `true` if this list contained the specified element, `false` otherwise.

```
numList.remove(3);                                 :( returns true ):
```

Check Empty

To check if a list is empty, call .isEmpty(). This function returns `true` if the list contains 0 nodes, and `false` if it has 1 or more.

```
numList.isEmpty();                                :( returns false ):
```

Get Size

Calling `.size()` returns the number of elements in the list.

```
numList.size();                                     :( returns 4 ):
```

Array

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

Array Declaration

Array declarations are made by specifying the type and name of the array. The naming conventions for the array are consistent with Strux's variable naming

conventions. Array types are shown before the brackets during declaration. For example:

```
num[] myArray;
```

Initializing an Array

Array sizes are indicated at time of array creation and should be specified for the array to be created. Once created, array sizes are immutable. You can create an array by using the new operator. The example below illustrates the creation of arrays in Strux.

```
num[] intArray = new num[5]; ✓:( creates integer array of size 5 ):

string[] name = new string[8];✓:( creates string array of size 8 ):

num[] numArray = new num[];      :( creates empty integer array
):
```

If we know the elements we want to put into an array, we can create one using this alternative syntax, without specifying the size of the array.

```
string[] struxers = {"Josh", "Sophie", "Millie", "Fred"};

num[] ages = {21, 20, 19, 20};
```

Accessing an Array

Array elements are accessed by their numerical index.

```
num[] numArray = {2, 4, 6, 8, 10};

show(numArray[2]);                              :( prints out 6 ):
```

Array values can also be assigned/modified by doing the following:

```
num[] numArray = {2, 4, 6, 8, 10};

numArray[1] = 3;

show(numArray);                    :( prints out [2, 3, 6, 8, 10] ):
```

Library Functions

There are a few built-in functions for manipulating arrays

Length

To find the number of items in an array, use the `.length` method.

```
num[] numArray = {2, 4, 6, 8, 10};

show(numArray.length);                          :( prints out 5 ):
```

Find

The `.find(x)` function returns the smallest index i, where i is the first occurrence of element x in an array. This function returns `-1` if element does not exist in array.

```
num[] numArray = {2, 4, 6, 8, 8, 22, 10, 30};

show(numArray.find(8));                          :( prints out 3 ):

show(numArray.find(11));                        :( prints out -1 ):
```

QuickSort

QuickSort is sorting algorithm we use to sort arrays in strux. QuickSort is a Divide and Conquer algorithm. We first consider the first, last, and middle element of the array. Between these three elements, we will pick the pivot, which is the median of the three. To sort an array using quicksort, call the function `.quickSort()`. To visualize quicksort, call the function `.showQuickSort()`. An example is shown below:

```
:( using .quickSort ):

num[] arr = {10, 100, 30, 90, 40, 50, 70};

arr.quickSort();                    :( calls quicksort on array ):

:( prints out sorted arr: [10, 30, 40, 50, 70, 90, 100] ):


:( using .showQuickSort ):

num[] arr = {10, 100, 30, 90, 40, 50, 70};

:( shows this: low = 0, high = 6, pivot = median(10,90,70) =
arr[high] = 70 ):

Initialize index of smaller element, i = -1

:( Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], myArray[j]
i = 0
j = 1 : Since arr[j] > pivot, do nothing
):

:(prints out step 1: [10, 100, 30, 90, 40, 50, 70] ):

:( j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1 ):

:(prints out step 2: [10, 30, 100, 90, 40, 50, 70] // We swap 100
and 30 ):

:( j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[] ):

:(prints out step 3: [10, 30, 100, 90, 40, 50, 70]):

:( j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2 ):

:(prints out step 4: [10, 30, 40, 90, 100, 50, 70] //100 and 40
Swapped):


:( j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3 ):

:(prints out step 5: [10, 30, 40, 50, 100, 90, 70] // 90 and 50
Swapped ):
```

```
:(We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping
arr[i+1] and arr[high] (or pivot)


:( prints out step 6: [10, 30, 40, 50, 70, 90, 100] // 100 and 70
Swapped ):

:( prints out sorted arr: [10, 30, 40, 50, 70, 90, 100] ):
```

BSTree

A tree is a data structure comprised of `BSTreeNode` objects, each of which has references to its children. In Strux, the tree is a binary search tree, meaning that it adheres to the following rules:

1. Each node has at most two children
2. All children in the left subtree of a node are less than the value of the parent node
3. All children in the right subtree of a node is greater than or equal to value of the parent node
4. BSTree only supports nums

BSTree Declaration

Initializing a binary search tree in Strux is as easy as:

```
BSTree tree = new BSTree();            :( Creates new empty tree ):
```

Additionally, a new binary search tree can be created with the following syntax:

```
BSTree tree = new BSTree({5,2,6,2,9});
```

This syntax is equivalent to creating a new, empty tree and then calling add to the tree on each of the numbers in the array. Therefore, it is equivalent to:

```
BSTree tree = new BSTree();

tree.add(5);

tree.add(2);

tree.add(6);

tree.add(2);

tree.add(9);
```

Library Functions

Add

Adds a new element to the tree. Because this is a binary search tree, the element is added is added according to its value. If the value is less than the root, the value is then compared to the left child of the root, and if the value is greater than or equal to the root, the value is compared to the right child of the root. This process is done recursively until the child that must be compared is null, at which point, a new TreeNode is created with the value to be added, and the TreeNode is added to the tree. A boolean is returned indicating whether or not the add was successful.

```
BSTree tree = new BSTree();                    :( tree is empty ):

tree.add(5);                                  :( tree now has 5 ):

tree.add(6);                              :( tree now has 5 and 6 ):
```

Remove

Removes the first instance of a specified value from the tree. When the element is removed, its children and parent are updated to reflect the change while still maintaining the binary search tree properties. The function returns true if the element was successfully deleted, or false if the value wasn't found inside the tree.

```
BSTree tree = new BSTree();
tree.add(5);
tree.add(6);
tree.remove(6);                              :( tree now only has 5 ):
tree.remove(1);                   :( returns false, tree unchanged ):
```

Contains

Used to check if a certain value can be found within a tree. Simply returns true if the value is in the tree or false if it isn't.

```
BSTree tree = new BSTree();
tree.add(5);
tree.add(6);
tree.contains(5);                                    :( returns true ):
tree.contains(2);                                   :( returns false ):
```

Functions

Built-In

main()

A `main()` function is required for every program to run. The program will not execute without a main method. The main method looks like this

```
void main() {
    show("Hello World!");
}
```

The main method does not return anything. Note that in this main method we have introduced another built-in function, called `show()`.

show()

`show()` takes in a data structure in its parameter and visualizes it. In the next examples, we will illustrate how show is used for our different data structures/types.

String:

```
show("Hello World!");
```

Will print this to the console:

```
"Hello World!"
```

Num:

```
num x = 3;
show(x);
```

Will print this to the console:

```
3
```

Array:

```
show({0, 1, 2, 3, 4, 5});
```

Will print this to the console:

```
[0, 1, 2, 3, 4, 5]
```

LinkedList:

```
show(new LinkedList::num({0, 1, 2, 3, 4, 5}));
```

Will print this to the console:

```
Head                                  Tail
+---+  +---+  +---+  +---+  +---+  +---+  +------+
| 0 |->| 1 |->| 2 |->| 3 |->| 4 |->| 5 |->| null |
+---+  +---+  +---+  +---+  +---+  +---+  +------+
```

Stack:

```
show(new Stack::num({1,2,3}));
```

Will print this to the console:

```
+---+
| 3 | <- Top
+---+
| 2 |
+---+
| 1 |
+---+
```

Queue:

```
show(new Queue<num>({4,5,6,1}));
```

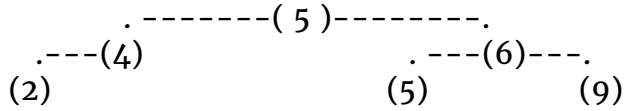
```
Head          Tail
+---+---+---+---+
| 4 | 5 | 6 | 1 |
+---+---+---+---+
```

BSTree:

```
BSTree tree = new BSTree({5,6,4,9,5,2});

show(tree);
```

Will print this to the console:



User-Defined

User-defined methods contain return types that are determined when writing the method signature, for example:

```
boolean isTrue() {
    return true;
}
```

returns boolean variable true. The return type is determined to be boolean, and the method signature is `isTrue()`. Note that if this method is defined to be

```
num isTrue() {
    return true;
}
```

This does not work.

Style Guide

The following statements are only suggestions for helping keep your Strux code clean and readable:

1. Use camelCase on variable and function names
    a. `addTwoNumbers()`
    b. `medianValue`
2. Use 4 spaces instead of the tab character. This ensures uniformity across all devices and text editors.