PLT Fall 2018

Shoo
Language Reference Manual

Claire Adams (cba2126)
Samurdha Jayasinghe (sj2564)
Rebekah Kim (rmk2160)
Cindy Le (xl2738)
Crystal Ren (cr2833)

October 14, 2018

Contents

1 Comments **2**
- 1.1 Single-line Comments . . . 2
- 1.2 Multi-line Comments . . . 2

2 Reserved Words **2**

3 Data Types **2**
- 3.1 Primitive Data Types . . . 2
- 3.2 Types Exclusively for Functions . . . 3

4 Statements **3**

5 Variables **3**
- 5.1 Variable Naming . . . 3

6 Arrays **3**
- 6.1 Declaring Arrays . . . 4
- 6.2 Defining Arrays . . . 4
- 6.3 Arrays in Arrays . . . 5

7 Structs **5**
- 7.1 Declaring Structs . . . 5
- 7.2 Variables of Type Struct . . . 6
- 7.3 Defining Variables of Type Struct . . . 6
- 7.4 Dot Operator . . . 7
- 7.5 "Destructing" Structs . . . 7

8 Operators and Arithmetic **8**
- 8.1 Plus Operator . . . 8
- 8.2 Minus Operator . . . 8
- 8.3 Multiplication Operator . . . 8
- 8.4 Division Operator . . . 8
- 8.5 Modulo Operator . . . 9
- 8.6 Boolean Operators . . . 9
- 8.7 Logical Operators . . . 9

9 Control Flow **9**
- 9.1 For Loops . . . 9
  - 9.1.1 Traditional For Loops . . . 9
  - 9.1.2 Enhanced For Loops . . . 10
- 9.2 Conditional Statements . . . 10

10 Functions **11**

11 Sample Code **11**

1 Comments

1.1 Single-line Comments

Single-line comments are denoted with two forward-slashes.

```
// single-line comment
```

1.2 Multi-line Comments

Multi-line comments are written in between /∗ and ∗/. Multi-line comments can be nested as long as the opening /∗ and closing ∗/ are all matched.

```
/* multi-line comment */

/* nested /* multi-line */ comment */
```

2 Reserved Words

boolean:	true	false				
control flow:	if	elif	else	for		
functions:	function	func	void	return		
primitive data types:	int	float	char	string	bool	any
data types:	array	struct				
built-in functions:	print	scan				

3 Data Types

3.1 Primitive Data Types

int is an architecture-dependent signed integer type.
float is an architecture-dependent signed floating-point type.
char is a character in ASCII.
string is a sequence of characters in ASCII.
bool is an expression with the value true or false.

− is used to denote negative numbers. It is a right-associative operator.

3.2 Types Exclusively for Functions

The type `void` has no associated value and can only be used as the return type for functions. This is useful for functions which are intended to perform "side-effect" operations only.

The type `any` is a function specific type that is a sort of catch-all, to allow for writing one function that can perform some equivalent operation on multiple types of data. For example, an add function that takes and returns type `any` will sum integers and floats as expected, but will concatenate strings when the parameters passed in are strings.

4 Statements

Blocks are defined for functions and statements using curly braces ({}). The curly braces must be balanced. Statements are usually terminated with semicolons (;). An empty statement is a semicolon by itself.

5 Variables

Variables must be declared with a type and a name. If the declaration is part of a larger definition, the variable name should be followed by an equal sign and a value corresponding to the type. If the variable is simply being declared, its name must be followed by a semi-colon.

5.1 Variable Naming

All variable names must follow [a-zA-Z][a-zA-Z0-9]* and can't be a reserved words (shown in the Reserved Words section).

Each variable has a type specified at the time of declaration. The type must be one of the primitive data types specified above or a composite types, which include arrays, structs, functions (func).

See the sections on Structs and Arrays below for information about declaring and defining variables of these types.

6 Arrays

An array is a container of primitive types, structs, functions (func), or other arrays (with caveats, see below).

6.1 Declaring Arrays

The declaration and definition can happen in one statement or two. See the Defining Arrays section below about how to do declare and define arrays in one statement.

The type of the elements for an array is specified at the time of definition. The size of the array is specified at the time of declaration. Arrays are defined using the keyword array, followed by arrow brackets (<>), the array name, and a semi-colon. The arrow brackets need to contain the type of the elements in the array. The semi-colon can be omitted if the array is being declared and defined in one line.

```
/* Declare an array called x that can hold integers. It has no size so element
   cannot be stored in it until its size is defined using "new." */
array<int> z;
```

6.2 Defining Arrays

You must define the size of the array when you declare an array. Arrays are fixed in size. To define an array, you have two options.

The first option is to use the keyword 'new.' Variables of types struct can be initialized with the keyword 'new' followed by parentheses that enclose the word array, the type of the array in the arrow brackets, and the size of the array in square brackets.

```
/* Define the array using keyword "new." You must provide a size in this step. The
   size goes in the [] brackets. */
array<int> z;
z = new(array<int>[5]);

// Declare and define an array in one line.
array<int> xy = new(array<int>[5]);

/* Declare and define an array of type struct. Assume BankAccount has already been
   declared as a struct type. See the section "Variables of Type Struct" below for
   information about declaring variables of type struct. */
array<BankAccount> ba = new(array<BankAccount>[100]);
```

The second option is to define the elements of the array explicitly. The elements are put in square brackets and are separated by commas. The last element in the list of elements is not followed by a comma. The whole statement is terminated by a semi-colon. The number of elements you list is the size of the array.

```
/* Declare an array and specify the elements in the array explicitly during
   definition all in one statement. */
```

```
array<int> x = [5,4,3,2,1];

// Defining the elements in the array explicitly after declaring the array.
array<int> b;
b = [1,2,3];
```

6.3 Arrays in Arrays

Currently, you cannot declare an array inside an array without also specifying the values when defining the array. For example, the following is okay:

```
array<array<int>> foo = [
      [5,2,3,4],
      [11,12,13,14]
];

array<array<int>> emptyArr = [[],[]];
```

But you currently cannot do something such as:

```
array<array<int>> foo = new(array<array<int>>[2][4]);
```

Additionally, you can have a struct with an array as a member so you can achieve 2D arrays where you don't have to specify all the values explicitly that way.

7 Structs

7.1 Declaring Structs

A struct is a list of grouped variables, whose types can be any primitive types, arrays, functions (func), or other structs. Structs are defined using the keyword `struct`, then the name, which must start with a capital letter and then can contain any combination of letters and numbers, all followed by {}, which contain the body of the struct. The body of the struct can only contain variables terminated by a semicolon. The variables are optionally initialized to values, which act as their default values. The body of the struct can also be empty, but the {} are still mandatory. Example:

```
 struct BankAccount { // Declare the struct.
    float balance = 0.0; // Initializing values is optional.
    int ownerId;
}

// See the function section for details about writing functions.
function myFunction(int y) int {
```

```
        return y+5;
}
 struct Baz {
    func field1 = myFunction; // This memeber is of type func and is initialized to
        myFunction.
    int field2;
}

struct SomeStuff {} // Empty struct
```

7.2 Variables of Type Struct

If the variable is of type struct, the struct type must be declared before the variable of that type is defined. The struct name, without the keyword `struct`, will be used to identify the type of the variable.

```
 struct BankAccount { // Declare the struct.
                      // This syntax is described above in the Struct section.
    int balance;
    int ownerId;
}
BankAccount myAccount; // Declare a variable of type BankAccount.
```

7.3 Defining Variables of Type Struct

Variables of types struct can be initialized with the keyword `new` followed by parentheses that enclose the name of the struct. The declaration and definition can happen in one statement or two.

```
BankAccount myAccount; // Declare a variable of struct type BankAccount.
myAccount = new(BankAccount); // Define the variable of struct type BankAccount.

// You can also declare and define the variable in one line.
BankAccount yourAccount = new(BankAccount);
```

Additionally, variables of type struct can be defined by assigning the variable to a block contain the names of the fields in the struct, all initialized to values. This block must have a semi-colon following the closing curly brace.

```
BankAccount myAccount;
myAccount = {balance = 0; ownerId = 12345;};
```

7.4 Dot Operator

You can access the element of a struct using the dot operator (.). You need to specify the name of the variable that is of the struct type (see the variable section below about how to declare variables of struct type) and the fields within the struct that you would like to access or change.

The dot operator is left associative.

```
BankAccount yourAccount = new(BankAccount);
yourAccount.balance = 5.27; // Change the value of the balance in yourAccount.
print(yourAccount.balance); // Read the value of the balance in yourAccount using
    the dot operator.
```

7.5 "Destructing" Structs

In the following code, a struct of some type `bar` named `foo` is created and then "destructed".

```
// define a struct Bar
struct Bar {
    int field1;
    int field2;
    int field3;
}

// create an instance of Bar named foo
Bar foo = new(Bar);
foo.field1 = 2;
foo.field2 = 3;
foo.field3 = 4;

int x;
int y;
int z;

// now destruct foo
{x; y; z;} = foo;

print(x); // will print 2
print(y); // will print 3
print(z); // will print 4
```

The line `{x; y; z;} = foo;` contains the destructing syntax. It is a shortcut for copying the fields of a struct instance into variables correspond to each field in one line. For instance, in the example above, the variable `x` now holds a copy of the value of `field1` in `foo` and similarly with `y` for `field2` and `z` for `field3`.

8 Operators and Arithmetic

For order of evaluation, our arithmetic operators follow PEMDAS. The modulo operator has the same precedence as multiplication and division. All operators, expect the NOT (!) described in 8.7 Logical Operators, are left associative. NOT (!) is right associative.

8.1 Plus Operator

+ is used for addition in the traditional mathematical sense for variables of int and float type. If you add two numbers and one is of type int and one is of type float, the variable of type int will be promoted to a float and the resulting value will also be a float.

```
int a = 5;
float b = 6.5;
float ab = a + b;
print(result); // this prints: 11.5
```

The addition operator can also be used to concatenate strings.

```
string one = "hello";
string two = " world";
string result = one + two;
print(result); // this prints: hello world
```

8.2 Minus Operator

The minus operator (-) is used for subtraction in the traditional mathematical sense for ints or floats only.

8.3 Multiplication Operator

Asterisk is used for multiplication in the traditional, mathematical sense for ints and floats only.

8.4 Division Operator

Forward slash is used for integer division. That is, when you divide two ints, you get the quotient and the remainder is discarded without any rounding. If you divide two floats or an int and a float, the result will be to some decimal point unit. This operator can be applied to ints and floats only.

8.5 Modulo Operator

The percent sign (%) is used for the modulo operation and can be used for ints and floats only.

8.6 Boolean Operators

Boolean operators in this language are: == , !=, <, >, >=, <=. They operate on ints and floats only. There is automatic type promotion when you compare ints and floats so they can be compared without any problems.

== and != can also be used for booleans and boolean expressions that evaluate to true or false.

8.7 Logical Operators

! is used for NOT in boolean expressions.
&& is used for AND in boolean expressions.
|| is used for OR in boolean expressions.

9 Control Flow

9.1 For Loops

`for` loops can be used to specify iteration and looping behavior. (Note: there is no while loop in Shoo.)

9.1.1 Traditional For Loops

A for loop statement has a header and a body. The header consists of three expressions separated by semi-colons. Both parts (the header and the body) are mandatory. The header has an initialization expression, a testing condition, an increment/decrement expression.

The initialization expression is evaluated one time only. It is evaluated before you test the test condition then possibly enter the for loop for the first time. This initialization expression is optional. If you choose to omit the initialization expression, you still must have one semi-colon denoting where it would have ended. Note, you cannot declare a variable (or have any other statement for that matter) in a for loop header.

The test condition is tested before you enter the loop the first time and then every time after the increment/decrement expression is evaluated until the test condition is false.

Finally, the increment/decrement expression is evaluated after each loop iteration, before the test condition is tested again. A loop iteration consists of evaluating the statements in the loop body. This part is optional. If you choose to omit the increment/decrement expression, you still must have one semi-colon denoting where it would have ended.

The loop body can contain any number of statements, include zero statements. The brackets are required even if the loop body is empty.

Example:

```
int sum = 0;
for (int i = 0; i < 10; i = i + 1) {
   sum = sum + i;
}
```

9.1.2 Enhanced For Loops

There is also the enhanced for loop, which operates on arrays only. The enhanced for loop has only an iterator variable and an array name in the header. The iterator variable is one declaration (without definition) of a variable that is of the same type as the items in the array it is iterating over. The iterator variable and the array name are separated by the keyword `in`. Example:

```
int result = 0;
array<int> quantities = [0,1,2,3,4];
for (int amount in quantities) {
   result = result + amount;
}
```

9.2 Conditional Statements

The if statement supports conditional execution. The body of the if statement can be a single statement or a block. It can then be followed by any number of optional elif statement and finally by a single, optional else statement. The headers for the if, elif, and else statements consist of boolean expressions. If the boolean expression for that condition is true, you evaluate that condition and then skip to the next statement without evaluating any following elif/else statements for that block. The parentheses and braces are required.
Example:

```
if (x > 0) {
```

```
   print("x is positive\n"); // print is a built-in function; see built-in function
       section below
} elif (x == 0) {
   print("x is zero\n");
} else {
   print("x is negative\n");
}
```

10 Functions

Shoo has first-class functions, which means that functions are treated as variables and can be passed as parameters, stored in variables, and returned from other functions.

The syntax of defining a function is to have the keyword 'function,' the function name, parentheses containing zero or more arguments, each with a type and a name, followed by the return type and curly brackets {} that contain the function body.
As mentioned in the section "Types Exclusively for Functions," functions can also have return types of void or "any".

```
function myMethod() int {
   return 5;
}
```

The keyword `func` is used to delineate functions passed as parameters from function definitions which use the keyword `function`. The below is an example of a function with another function passed in as a parameter.

```
function baz(func f, int x) int {
   int result = f(x)+5;
   return result;
}
```

And because we have first-class functions, we can also define functions inside of other functions (and then return these functions and so on).

11 Sample Code

A sample program that demos the use of the `any` keyword, arrays, and higher-order functions in Shoo.

```
/* This program sums over each array in the 2d array. */
function sampleProgram1() void {
```

```
  array<array<int>> tasks = [
    [1,2,3,4,5,6,7,8,9,10],
    [11,12,13,14,15,16,17,18,19,20]
  ];
  function sum(int x, int y) int {
    return x + y;
  }
  function foldl(func f, any acc, array<any> items) array<any> {
    if (length(items) == 0) {
      return acc;
    } else {
      return foldl(f, f(acc, first(items)), rest(items));
    }
  }
  function map(func f, array<any> items) array<any> {
    if (length(items) == 0) {
      return [];
    } else {
      return concat(f(first(items)), map(f, rest(items)));
    }
  }
  array<int> results = map(function (array<int> task) array<int> { return
      foldl(sum, 0, task); }, tasks);
  print(stringOfInt(foldl(sum, 0, results)));
}
```