Solace Running Documentation

Noé Garcia

March, 2023

1 Introduction

This document is a running documentation for the Solace language. This documentation contains information regarding value types, and general syntax along with simple examples.

2 Types

Solace is a statically typed language. All values are also immutable. The following table contains all available atomic types in Solace:

Solace Type	Description
int	general integer
float	general floating point number
char	general character
string	general string value
bool	general boolean value
:sym	symbol value
func	function type

There are a small number of composite types as well as the standard types listed above. The following table contains all available composite types in Solace:

Solace Type	Description
tpl	tupple type utilized for grouping values of different types
array	array types utilized for grouping values of the same type
struct	structure types for more complex data and value groupings

2.1 Integers and Floats

Built in number values are represented within Solace as integers or floats. Much like other languages, integers represent whole values, and floats represent floating point real values. integer values can be defined in the following manner:

```
pack:Main

func main int()
{
        // Declare a to be an integer and b to be a float
        a int = 1;
        b float = 0.1;

        // values of the same type can be defined with the use of commas:
        c, d int = 2, 3;

        0;
}
```

There are a number of arithmetic and comparison operations that are available for both integers and floats. Like other languages, number values are able to be added, subtracted, multiplied, and divided. Numbers are also able to be compared between one another. The following table contains operators for number types in Solace:

Operators	Description
+	addition
−	subtraction
∗	multiplication
/	division
%	modulo
=	value assignment
==	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
!=	not equal to

There are a small number of special operators.

The concatenation operation takes two values on the left and right of the concatenation operator and returns a new single value with the left and right value concatenated together. Only values of the same type may be concatenated. The new value returned will be of the same type as the original values. There is a single case in which the concatenate returns a different value than that of the values given, and that is with the concatenation of characters. The concatenation of two character values together will result in a string.

The ignore character operator functions exactly as its name implies. Values that are stored using the ignore chracter operator will be forgotten. This operator is utilized when a value is no longer necessary and can be safely forgotten.

The action delegation arrow operator is a main tool utilized in the control flow of any Solace program.

Special Operators	Description
|	concatenation of left and right
_	value ignore character
−>	action delegation

Arithmetic and comparison operators are performed between two given values. The following is an example of each operator usage:

```
pack:Main

fun main int ()
{
        // assume a and b are integers:
        a, b int = 5, 10;

        // arithmetic can be performed
        addition int = a + b;
        subtraction int = b - a;
        division int = a / b;
        multiplication int = a * b;

        // numbers can also be compared:
        a > b;  // false
        a < b;  // true
        a >= b; // false
        a <= b; // true
        a == b; // false
        a != b; // true

        // The above can also be done with float type variables, or between integers
```

```
        // and float variables.

        0;
}
```

2.2 Strings and Characters

Solace has two different types for handling raw text values: characters and strings. Characters represent a single character value, while strings represent a longer collection of character values in succession. Character variables and values can be compared between one another with the equality (==) operator. Strings can also be compared to one another in a similar way. Character values can be concatenated together to result in a new string value containing both character values. Strings can be concatenated together to result in a new string value containing both original strings.

```
pack:Main

func main int ()
{
        // define character and string variables
        a char = 'a';
        b char = 'b';
        c string = "hello,";
        d string = " world!";

        // comparisons between character values
        a == a; // true
        a == b; // false
        // comparisons between string values
        c == c; // true
        c == d; //false

        // concatenate values together
        ab string = a | b; // "ab"
        cd string = c | d; // "hello, world!"

        0;
}
```

2.3 Booleans and Symbols

Boolean values can represent two different states: true and false. boolean values are able to be compared through the use of comparison operations. Symbols are a unique type as they represent defined names rather than raw values. There are two predefined symbols: $:ok$ and $:err$.

```
pack:Main

func main int ()
{
        // function call will return two values: a symbol and the original integer.
        // the following will return (:ok, 4)

        evenResult :sym, val int = isEven(4);

        // function call will return a single boolean value: true or false.
```

```
        // the following will return false

        oddResult = isOdd(6);

        0;
}

func isEven (:sym, int) (n int)
{
        { (n%2 != 0) -> (:err, n);}
        (:ok, n);
}

func isOdd bool (n int)
{
        { (n%2==0) -> false;}
        true;
}
```

Symbols are special values within Solace. Symbols aside from :ok and :err can be defined for furhter use. Symbols hold special value within programs as their values are directly tied to their name, and no other value. variables can contain symbol values. The following is a simple example of symbol definition and use.

```
pack:Main

// define two new symbols
:sym {:apple, :orange}

func main int ()
{
        // define a symbol variable
        apple :sym = :apple;
        orange :sym = :orange;

        // symbols can be compared.
        :orange == :apple; // false
        :apple == :apple;  // true

        // symbol variables can also be compared
        apple == orange;   // false
        orange != apple;   // true

        0;
}
```

2.4 functions

Functions function like any other type in Solace. Unlike other types, functions are defined using the func keyword before the name of the function, not after. Functions can take multiple parameters but **must** return a sinle value. Functions cannot return no value, or more than one value. To return multiple values, tuples may be used to encapsulate all values to be returned. Functions, like any other type, can be passed into other functions as parameters. The following is an example of function definition and usage:

```
pack:Main
```

```
func main int ()
{
        // call a function and store the results
        result int = addTwo(2, 3);

        // call a function that returns another function and store it
        // wthin a variable
        f func = retFunc(); // prints: "Returning a function from a function"
        f();                // prints: "Function returned from another function"

        // define a function
        func double int (n int) -> n+n;

        // call the function that applies the passed function argument
        // on the given integer value
        doubleResult int = applyGivenFunc(double, 5); // 10

        0;
}

func addTwo int (a int, b int)
{
        a+b;
}

func retFunc func ()
{
        io:out("Returning a function from a function");

        () :sym -> {io:out("Function returned from another function"); :ok;}
}

func applyGivenFunc int (f func, n int)
{
        f(n);
}
```

3 Complex Data Types

Among the basic types available in Solace, there are a small number of complex types. The inclusion of complex types enable efficient grouping and defining of the more simple types of Solace. This enables the construction of more interesting data values and interactions between information.

Solace offers the use of structure types. Similar to structures seen in languages such as C, structures are defined using a special name and all included types in the structure as their signature. The following is a simple showcase of how to define and use a structure in a Solace program.

```
pack:Main

struct cake
{
        cakeType string;
        icingType string;
        candles int;
};
```

```
func main int ()
{
        // Define a new cake structure
        // The new chocolateCake structure instance is a cake with
        // the cakeType value as "chocolate", the icingType value as
        // "vanilla", and the candles value as 12
        chocolateCake cake = cake{"chocolate", "vanilla", 12};

        // Access the value fields of a structure instance
        chocolateCake:cakeType; // returns "chocolate"
        chocolateCake:candles;   // returns 12

        0;
}
```

4 Variables and Naming

Variables are essential to Solace, and are a core functionality of the language. Variables have a broad naming convention, but there are a few restrictions on variable names.

Variable and function names cannot begin with a number. Names such as "1*value*" are not allowed; however, numbers are viable anywhere else within a name, for example "*value*2" or "*here*2*there*" are viable names. Variable and function names can begin with and contain upper case letters. Names such as "*Dataval*" and "*dataVal*" are both viable. Standard naming convention for variable and function names is camel case. Names cannot start with or contain any special characters with the exception of being able to contain the underscore character: "_". Names are able to utilize the underscore character within a name such as "*some_var*". The underscore character holds a special meaning where any value bound to the variable whose name is soley the underscore character is dropped from memory.

5 Control Flow

Solace defines its own control flow for program development. Rather than having if-else statements, Solace utilizes pattern matching code blocks. Control flow is defined and contained through the use of brackets, and the arrow operator. The following is an example of simple control flow:

```
pack:Main

func main int ()
{
        // determine if an integer is even or odd
        evenOrOdd(5); // odd
        evenOrOdd(8); // even

        // function for a more complex control flow
        groupFunc(8); // Mod4
        groupFunc(9); // Mod2
        groupFunc(7); // NoMod

        // control flow with default functionality
        doubleIfEven(6); // 12
        doubleIfEven(5); // 5

        0;
```

```
}

// This function takes an integer and returns the string
// "even" if the integer is even and "odd" otherwise
func evenOrOdd string (n int)
{
        // pattern matching block
        // if n mod 2 is 0, do what is to right of arrow:
        // return even
        {(n%2 == 0) -> "even";}

        // otherwise return odd
        "odd";
}

// This function takes an integer and returns a string
// depending on if the integer is divisible by 4, 3, 2
// or none.
func groupFunc string (n int)
{
        // The control flow block. other patterns
        // begin on a new line with a bar-arrow character: |>
        {(n%4 == 0) -> "Mod4";
        |> (n%3 == 0) -> "Mod3";
        |> (n%2 == 0) -> "Mod2";}
        "NoMod";
}

func doubleIfEven int (n int)
{
        // The control flow block. The base case
        // is defined with no pattern, just an arrow
        {(n%2 == 0) -> n*2;
        |> -> n;}
}
```