The Solace Programming Language

Noé Garcia

January, 2024

1 Introduction to Solace

Solace is a small functional programming language developed as a hobby project starting in 2023. Solace is built to be a bridge between imperative and declarative functional programming language domains. Built to be simple and easy to read and write with, Solace is a statically typed, semi-pure functional language with basic support for simple tooling and functionality. While in its infant stages, the language will grow to support more common aspects found in other languages.

Functional programming can be intimidating for those unfamiliar to many who have never worked with it before. For this reason, Solace is a case study for a programming langauge which aims to make the introduction to the functional paradigm more familiar and easily approachable.

Solace is a general programming langauge as constructed, but domain specific library extensions will be built to extend the capabilities of Solace into specific use cases. As it stands, there are plans to epand on Solace by adding the ability to write logic centric programs.

This text is an introdution to the Solace language as it is, and should be used by anyone who is looking to learn how to read and write in the language.

2 Functional Programming

What is functional programming? In essence, functional programming is the practice of building computer programs using functions. While this sounds similar to other languages if you have worked with languages such as C or Python, there are a few rules that the functions in functional programming languages follow that are not followed in other imperative langauges.

The first is that functions should all accept at least one variable. And, each function may only have one output. For every input of a function there is a designated output; No matter how many times the function is run with that input, it will always result in the same output.

The Second is that functions do not access or manipulate variables or values outside of themselves. This means that functions are not able to work with global variables in their computation. Only the variables passed into the functions and variables defined within the function are able to be used in any computation within the function. Building on top of this, functions do not create side effects. This means functions do not alter variable values or states outside of its own scope. All functions are effectively stateless and perform the exact same every time they are called with the same input (as mentioned in the above point).

Lastly, all data in functional programs are immutable. When a variable is declared, its value is not able to be changed. This makes values much more safe to use in complex operations that may share the same values. Building on top of this, functional programming languages do not support loops that are found in imperative domains, such as for and $while$. Because values are immutable, the condition value in a $while$ loop would never change, likewise with the for loops, the variable defined in its scope would never be able to increase or decrease in value, making the loop impossible in the functional domain. Rather than using loops, however, recursion is used when looping is necessary in some computation.

Functional programming is a shift in how programs should be written. This shift makes for code that is more safe and easy to understand. In the following sections, this document will go into depth of the Solace programming language and provide information on the data types and structures of the language as well as how the language is written, used, and understood.

3 Domain Specific Language

Solace is a general purpose functional programming language built without a specific domain in mind. There are a number of general purpose programming languages that exist today, so there are a number of languages to choose from in that regard, Solace aims to be a hobby/playground/research language. The functional domain, while gaining traction in use, is still regarded as a fringe domain to build production level code with. There is plenty to be done in terms of functional language development, and Solace is a testing ground for that.

Beyond functional programming, logic programming is another domain of programming development that has not been brought out of the academic sphere much. Solace is the perfect language to begin to test

aspects of logic programming being brought into a toy language to play around with and to understand its respective use cases better.

4 Getting Started

Solace is written in C and compiled with the *Clang* compiler. Apart from this, the lexer for Solace is written in *Flex* (also known as fast lexical analysis generator), and the parser is written with GNU Bison. Solace is hosted on github here and can be cloned to a local machine using git. After Solace is installed, it can be built by typing the following into the console while within the *Solace/src* project directory:

```
$ make
```

This will run the make the executable compiler file in the source directory and build the language from source. Once this completes, the executable *solx* file will be available. This is the Solace compiler program. To compile a Solace file run the following with the Solace source file to compile:

```
$ ./solx solaceFileToCompile.solc
```

All Solace source files should have the *.solc* file extension to signal to the compiler that what its looking at is a Solace source code file. The compiler is also able to handle Solace source code files that do not have file extionsions. If a source code file named *Fibonacci* (with no file extension) is given to the compiler, it is taken and the *.solc* extension is appended to the end of the file name.

4.1 Writting Solace for the First time

Solace is easy to get started with. Similar to the C language, Solace requires a few things in order to compile and run. The first is a *module space name* definition, and then a main function (although main functions are not required when writting library functions). The following is an example of writting the most simple algorithm we can in Solace: A program that exits.

```
space::Main;

func main int() {
        0;
}
```

Disecting the source code above, we can see the first line of the program explicitly defining the module space to be *Main*. Solace picks the *Main* module space as the space to run when executing the project; it is where the compiler looks for the main function. The next line is the *main function* definition. We denote that we are defining a function with the *func* keyword followed by the name of the function and then its return value. All functions must have a return value. Main functions should return an integer to signal the exit value of the program. Within the main function there is a single expression: 0; This tells the main funciton to return 0 and exit.

While this is a simple program we can see how Solace files are structured. After saving the source file as *justExit.solc* we can then compile the program with *./solx justExit.solc* .

In the following sections this document will break down the Solace language and introduce new concepts and how to use them in the Solace language.

5 Types

Types are a foundational building block to all programming languages. Different types are represented to the computer using different sizes of bytes, and allow the computer to know what to expect when we define some value for it to store for later use. In languages like C, variables are statically typed. This means that the author has to explicitly define what type a variable is at its declaration. Solace is also statically typed. To define a variable in Solace, the name of the variable is first given followed by the type of the variable, and then the assignment of the variable.

Types can be broken down into two different classifications for Solace: atomic and composite. Atomic types are the basic building blocks of the typing system within the language, while composite types are types that are built on top of atomic types, used to handle the structure and storage of data.

Solace has general atomic types that are present in many other programming languages: integers, floating point numbers, characters, strings, booleans, functions, and symbols. In Solace (like in many other functional programming languages) functions are considered to be basic atomic types.

5.1 Atomic Types

reserved words	description
int	integer data type, defaults to int64
float	floating point number type, defaults to float 64
char	stores a single character
string	stores multiple conjoined character values between ""
bool	either a true or a false value
:sym:	symbolic literal values
func	defines funciton definitions and types

Integers are used to handle and store whole numbers. Numbers like 0 or -14 can be stored in a variable with the integer type. Float types are used to handle real numbers. Floats can store and handle values such as 0.1 or 0.0001. Characters handle single character values such as $'a'$, while strings can handles multiple characters strung together like "$hello, world$!". Booleans handles values like $true$ and $false$. Function types are used to store whole functions. In functional programming languages such as Solace, functions are treated like any other type. Symbols are an interesting type, as they store literal values. A symbol value : *apple* : is only equivalent to itself, and represents its literal value as defined by the author. Symbols can be any string of characters (excluding special characters and white space).

Composite functions, as was previously defined, are built on top of atomic types as a coalition enabling the structure and storage of information. Solace provides the following composite types: list/arrays, tupples, and structures. These types are more complex than atomic types. Arrays are able to store a collection of values within its definition, enabling storage of multiple values within a single variable. Tupples are similar but more flexible than arrays. Arrays are a collection of values that are of the same type, tupples are able to store multiple values of different types. Structures are the most flexible of the composite types, enabling the construction of a data store capable of storing multiple variable values.

5.2 Composite Types

type name	description
list/array	contains a number of same type values.
tupple	can contain number of different type values.
struct	designating a structure object definition for more complex data objects.

5.3 Defining and storing values

Things

6 Lexical Rules

Like in many languages, Solace has rules as to what constitutes legal syntax for things such as variable declarations, function definitions, and naming criteria.

Solace defines a number of reserved words that are used in defining types, function definitions, and data structures. The reserved words chosen for Solace are designed to be similar to other languages, such as C, while taking a different approach. A lot of the reserved words are defined for type declarations shown above, or the legal operators for the language that are defined below. The following are the reserved words for the language that do not fit in under the other sections.

6.1 Reserved Words

Reserved word	description
main	used to declare the main function of the program
use	used to include packages/libraries
module	used to define the module the file belongs to
ret	keyword used to return value(s) from function

7 Operators

The following showcase some of the legal oporators for Solace

Oporator character(s)	description
+	addition operation
−	subtraction operation
*	multiplication operation
/	division operation
%	modulo operation
>	greater than comparison
<	less than comparison
>=	greater than or equal to comparison
<=	less than or equal to comparison
==	equal to comparison
!=	not equal to comparison

8 Syntax

This section outlines an overview on the basic syntax of the Solace language. The main goal of the syntax is to remain simple but flexible enough to build interesting programs. This includes the ability for defining variables and functions, higher order functions, and establishing basic program workflow. The following showcase a simple program that defines a fibonacci function and the main function of the program. The main function is necessary for the solace to run, similar to C type languages.

Solace will take an approach to syntax similar to C like syntax. Solace is a statically typed langauge, so variable types must be defined upon their declaration. Code blocks are defined through the use of brackets. nested blocks can be defined within function definitions for pattern matching functionality.

The following is a showcase of a simple program:

```
module:Main

func main int ()
{

        out int = fibonacci(7); // returns 13

        // there is a ret and return keyword, but is not needed.
        // the last statement in a block will be returned.
        ret 0;
}

func fibonacci int (n int)
{
        // embedded blocks can be used when pattern matching
        // and grouping functionality:
        { (n <= 1) -> 1; }
```

```
        fibonacci(n-1) + fibonacci(n-2);
}

func factorial int (n int)
{
        // when matching a single parameter, it is possible to
        // omit the parameter for just the matched values.
        { (0) -> 1; }
        n * factorial(n-1);
}
```

In maintaining with the traditional functional language paradigm, all functions are designed to be pure. There are no plans as of this time to completely force pure functions.

9 Summary

Solace is a simple functional programming language built as a toy project language. This language is meant to be simplistic to read and write programs with. Solace is written utilizing Yacc, Bison, and C, with the clang compiler. The goal right now is getting the language up and off the ground by implementing the specifications defined above. From there, the language will be evaluated and this document will be updated on where the language will build from there.