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 while 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. There is currently **NOT** a build tool available, so all library requirements should be installed independentally. Solace Relies on Clang, Bison, and Flex. 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 Writing 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 writing library functions). The following is an example of writing the most simple program we can in Solace: A program that exits.

```
space::Main;

fn 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 *fn* 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 complex 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

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 types, as 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: functions, list/arrays, tupples, and structures. These types are more complex than atomic types. Functions are collection blocks of algorithmic logic that we use to group the functionality of our program. 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 different named type values.

5.2 Composite Types

type name	description
fn	defines blocks of algorithmic logic
list	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

Variables are essential to storing and retrieving data in programs. Within Solace users are able to define their own variable names and their types as well as the data information they want to store within the variable. Variables are defined using simple syntax rules:

```
space::Main;

fn main int() {
        n int = 32;
        0;
```

```
}
```

Looking at the code given, the initial setup has been seen before, what's different is the line *n int = 32;* . This tells the Solace compiler that we want to define a variable n that stores an integer value of 32, and we end the line with the semicolon character (;) to signal to the compiler that we are ending our line. This syntax definition is standard for defining any variable of any type.

```
space::Main;

fn main int() {
        letter char = 'c';
        passage string = "hello, world!";
        example bool = true;
        apple :sym: = :apple:;
        pie float = 3.14;

        0;
}
```

In the code segment above, there are examples of different variable types being defined. As previously discussed the *char* type is able to hold a single character value within single quotation marks, in this case it is given the value $'c'$. While character types are able to hold any aphabet symbol, numbers can also be represented ($'4'$) or special characters ($'\$'$). Strings are able to hold entire segments of characters, all incased within double quotation marks. In the above case the string *passage* contains the passage *"hello, world!"*. Boolean values are a set of values, true or fasle. Above, the variable *example* holds the boolean value *true*, but could also hold the value for $false$. The *apple* is a symbol type variable. Symbols are defined by encasing a name within colons on each side. the *apple* variable contains the : *apple* : symbol value. Floats are real number values. The variable *pie* contains the value 3.14 but can also contain different types of floats like 1.0 and 3.000007. The last type to mention is the function type. Functions are a little different than the rest of the types. Typically, to define a function the fn keyword is placed before the name of the function, followed by the return type of the function. Functions are the only type to have such syntactic behaviours.

6 Syntax Rules

Like in many languages, Solace has rules as to what constitutes legal syntax, as shown previously with variable declarations, function definitions, and naming criteria. Solace also utilizes different keywords to construct and enforce these syntax rules. Keywords are spercial words within a programming language that are not able to be used in variable, function, structure, or module space name defintions. They are standalone words that define specific meaning within the language. Solace defines a number of reserved words that are used in defining types, describing function definitons, and establishing data structures. The reserved words chosen for Solace are designed to be similar to other languages, such as C. 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 a list of the reserved words Solace utilizes.

6.1 Reserved Words

Reserved word	description
space	declare the module space for the following program
main	declare the main function of the program, used in the Main space
use	include external packages/libraries
return	keyword used to return value(s) from function
self	reference to the parent function
int	integer type keyword
float	floating point type keyword
char	character type keyword
string	string type keyword
:sym:	symbol type keyword
bool	boolean type keyword
fn	function type keyword
struct	structure definition type keyword

The names above the double line within the table are special words used to define specific parts of the program, while the names bellow the double line are type keywords that are used in defining or referencing the specific types of Solace. These names cannot be overriden by the user when building programs. You may not name a variable *int int = 4;* since the *int* keyword is reserved for describing the type of the variable in this case.

7 Operators

Operators are special characters that perform some operation within Solace. There are three different kinds of operations within Solace, numerical operations, logical operations, and string operations. Numerical operations take two numerical values and apply a specified mathematical operation to obtain a numerical result. An example of this is addition: $2 + 2$ would result in the value 4. The operation $3.0/2.0$ results in a value of 1.5. Logical operations are operations that determine a logical result from one or two boolean values.Some logical operations only take boolean values. We can use the and operator to test two different boolean values and return a resulting value as such: $true \&\& false$ which will result in $false$. Other logical operators can take numerical, character, or string data and compare them. For example $5 > 10$ (five is greater than 10) would result in the $false$ boolean value. Another example is comparing two characters $'c' == 'c'$ which results in $true$.

Operator character(s)	description
+	addition operation
−	subtraction operation
*	multiplication operation
/	division operation
%	modulo operation, find remainder of a division
&&	And logical operation
||	Or logical operation
!	not operation, take the inverse of a boolean value
>	greater than comparison
<	less than comparison
>=	greater than or equal to comparison
<=	less than or equal to comparison
==	equal to comparison, test equivalence
!=	not equal to comparison, test non-equivalence
|	concatenation operation, join two values together

8 Performing Operations in Solace

This section outlines how to operate on values and expressions in Solace. If you are familiar with C or any other high level programming language, performing operations in Solace will be easy to understand and pick up. Solace allows the ability to perform mathematical and logical operations on data. the following is a simple showcase of how to perform the basic mathematical operations on integer and float values.

```
space::Main;

fn main int ()
{
        2 + 2; // returns 4
        5 - 2; // returns 3
        0 * 7; // returns 0
        6 / 3; // returns 2
        8 % 4; // returns 2

        2.0 + 1.0; // returns 3.0
        0.0 - 1.0; // returns -1.0
        2.5 * 2.0; // returns 5.0
        7.0 / 3.0; // returns 2.5

        // Solace preserves the type of the highest precision value in the operation
        2.0 + 4;  // returns 6.0
        10.0 / 2; // returns 5.0
}
```

Solace allows the concatenation of a number of different types, including characters, strings, and lists. To concatenate two values together, the bar operator is used between the two values you want to concatenate. The concatenation operation is ordinal, so in whatever order you place the values to concatenate, the result will reflect that order. Concatenation always results in a larger value, and for that reason, when characters are concatenated together the result is a string rather than a character.

```
space::Main;

fn main int ()
{
        character1 char = 'h';
        character2 char = 'i';

        // result is a string
        concat1 string = character1 | character2;

        unfinishedString string = "ello";

        // result is a string: "hello"
        concat2 string = character1 | unfinishedString;

        unfinishedString2 string = ", world!";

        // result is a string: "hello, world!"
        concat3 string = concat2 | unfinishedString2;

        0;
}
```

Mathematical operations consist of two elements with the operation separating them in the middle.

Solace is able to handle addition, subtraction, multiplication division, and remainder operations as showcased above. Mathematical operations consist of either integer values or floating point values. When performing an operation on an integer and a floating point number Solace will return a floating point number to ensure the highest precision of the operation is maintained.

Comparison and logical operations are performed similar to that of mathematical operations. When comparing two values, the comparison operation resides between the two values being compared.

```
space::Main;

fn main_int()
{
        8 > 7;  // returns true
        1 < 0;  // returns false
        5 >= 5; // returns true
        6 <= 9; // returns true

        // some logical operations work with all types
        1 == 0;       // returns false
        'a' == 'a';  // returns true
        "hello" == "hello" // returns true

        1 != 0;       // returns true
        'a' != 'a';  // returns false
        "hello" != "hello" // returns false

        // Some logical operations only take boolean values in their operation
        true && true;  // returns true
        true && false; // returns false
        true || true;  // returns true
        false || true; // returns true
        !true;              // returns false

        0;
}
```

Comparison logical operations have a broad set of input capabilities. Unlike mathematical operations, which are only able to take numerical input, comparison operations can take numerical values, cahracter values, string values, boolean values, or symbolic values. When comparing if some value is greater than or less than another values, the comparison can only handle numberical values, however, as showcased above, when comparing whether two values are equal to each other or not, Solace is able to handle any value type that is thrown at it. All operations return boolean values.

Logical operations can only take boolean values as input and only return boolean values. There are three logical operations available: logical and, logical or, and logical not. Logical and only returns true when both values are true. Logical or only returns false when both values are false. Logical not returns the negation of the value given.

9 Working with Functions

Solace is a functional programming language and as such functions are treated the same way as any other type in the language. Functions are defined through the use of the function keyword followed by the name of the function, the return type, the parameter list, and then the function body. Seen above are a few examples of defining functions (we defined a main function to use). We can easily define and call functions within our programs.

```
space::Main;
```

```
fn main int ()
{
        IO::out(double(5)); // prints 10;
        0;
}

fn double int (n int)
{
        n * 2;
}
```

Here we define two functions: the main function that runs when we execute the program, and the double function that takes an integer n and returns double its value back. Within the main funciton, we call the double function and print its output to the console, and then we return 0. With Solace every function must return a value, and the type of the return value is specified in the heading of the function definition. To return a value from a function, Solace automatically takes the last expression in the function and returns it. We are also able to use the *return* keyword to explicitly state to the compiler that you want to return a value.

Since functions are like any other type, we are able to define and bind full functions to variable names to use, we can even return functions if we want! To do so we use lambda functions. These special functions are essentially nameless functions that can be bound to some name or used as they are with no reference bounding them.

```
space::Main;

fn main int ()
{
        // we can bind a function to a local variable by defining it
        // as the assignment expression
        double1 fn = (int n) -> { n * 2; };

        // we can bind a function from a return value of another function
        double2 fn = returnAFunction();

        // we can call the functions like any other function
        double1(5); // returns 10
        double2(2); // returns 4

        0;
}

fn returnAFunction fn ()
{
        (n int) int -> { n * 2; };
}
```

Above we can see some examples of how Solace is able to handle functions as types. The *returnAFunction* function returns a lambda function that can then be stored within another variable to use later. Lkewise, we are able to define functions within other functions by binding them to another variable of the funciton type. Finally we are able to call the lambda functions like any other function, by referencing where its bound and providing any necessary paramters.

Lets take a look at more complex use cases for functions. In Solace, there are not any loops, and as such we have to make use of recursion. Let us assume we are building a summation function for values from 1 to the given limit. We can build it as follows:

```
space::Main;

fn main int ()
{
        summation(5); // returns 15 (5 + 4 + 3 + 2 + 1)
}

fn summation int (n int)
{
        { (n <= 1) ->
                return n;
        }
        n + summation(n-1);
}
```

We begin by creating the funciton signature for summation as taking a single input parameter n that is of type integer and its return type is also the integer type. We then have to create our base case; the condition for which we stop calling the summation function. In this case, our base case is if the value of n is equal to or falls below 1. If this happens we simply return n. If this is not the case, we want to return the current value of n added to the return value of calling the summation function again with the parameter $n-1$. This ensures we work our way down from the original n value that was passed into the function when it was first called. Using this technique of recursion, we are able to simulate any looping we would need to do in the program.

This creates a big problem, however. When working with a lambda function that does not have a name to reference, how can we use recursion? To deal with this problem Solace provides the *self* keyword. To reference the parent function the program is currently in, we can use *self* followed by the necessary parameter list in order to recurse.

```
space::Main;

fn main int ()
{
        // define a lambda function that requires looping to complete
        raiseToPower fn = (n int, r int) int -> {
                { r <= 1 ->
                        return n;
                }
                n * self(n, r-1);
        };

        0;
}
```

In the Example given above, we create a lambda function that we store in the variable *raiseToPower*. This function takes two parameters: n the base integer value that we want to raise to a power, and r the power we want to raise n to. We first define our base case to check if r is less than or equal to 1. If it is, we simply return n. If the base case is not met we want to multiply n by itself once more and recurse. We do this with n * $self(n, r-1)$ where we call the lambda function again with the parameters n and $r-1$.

10 Pattern Blocks and Control Flow

In any programming language control flow is essential in dictating how a program executes given specific conditions. Solace is no different, and offers what are called pattern blocks to assist in managing the control flow of programs. Patterns blocks are shown and used in the previous section within functions. Pattern

blocks are defined through the use of brackets to denote a new block, followed by an expression, and arrow operator, and expressions to be executed if the pattern expression is found to be true.

```
space::Main;

fn main int ()
{
        { true != false ->
                IO::out("true does not equal false");
        }

        0;
}
```

In the pattern block, the pattern expression, $true\ != false$, dictates whether or not the proceeding expressions within the pattern block are executed. If the pattern results in *true* then the block is executed, if the pattern results in *false* then it does not, and is skipped over.

There are some cases in which we need more pattern expressions within a pattern block to dictate complex control. In this case, we can use the bar operator to separate different expression patterns without having to completely define a new pattern block. If there is a situation in which we want to perform a specific task within a pattern block if no other pattern branch was executed, we are able to leave the pattern expression empty and just apply the arrow operator followed by the execution expressions.

```
space::Main;

fn main int ()
{
        value char = "c";

        { value == "a" ->
                IO::out("the value is a");
        | value == "b" ->
                IO::out("the value is b");
        | value == "c" ->
                IO::out("the value is c");
        | ->
                IO::out("unsure value");
        }

        0;
}
```

11 Lists and Tuples for Collections of Data

11.1 Lists

When we want to group a number of values together, we can use a list. For example, let us assume we are storing the average temperature for each month in our program. We could define 12 separate variables, each coresponding to a different month in the year, but then if we wanted to pass those values to a function we would have to pass each of those 12 variables as parameters. Instead, we can use lists. In Solace, lists enable us to define a list of values and store them in a single variable. Later, when we want to access a specific value in the list, we use bracket notation along with a integer value to tell the compiler what value we want to retrieve.

```
space::Main;
```

```
fn main int ()
{
        // define an array to contain temperatures
        tempuratures [float] = {32.7, 46.9, 53.1, 68.4, 73.1};

        IO::out(temperatures[2]); // print the temp for march

        0;
}
```

In the example given above we can see that we create an array to store the average temperatures of the first five months of some year in a single variable named *temperatures*. We then print the third entry in the list, the entry for march (denoted in the index 2 as we begin our indexing from 0), to the console. In this way, we are able to define new lists of any type we want, and access the values of the array using the simple bracket notation.

Similar to characters and strings, we are able to concatenate lists together, and add values to lists through concatenation. Using the bar operator, we can add single values to the beginning or ends of lists so long as the value we are concatenating and the types of the values in the list are the same. We are also able to take two different lists and concatenate them together so long as the types of their values matches as well.

```
space::Main;

fn main int ()
{
        n int = 0;
        nums [int] = {1,2,3};
        nums2 [int] = {4,5,6,7};

        // combine lists with concat
        combinedNums [int] = n | nums; // returns {0,1,2,3}

        finalCombinedNums [int] = combinedNums; // returns {0,1,2,3,4,5,6,7}

        0;
}
```

Looking above, we can see that we can add a single integer to the beginning of a list of integers through the use of the bar operator. We can also concatenate two different lists together as we combine both integer lists and store it in another variable.

11.2 Tuples

Tuples are another collection type. Like lists, tuples are able to store data in an organized fashion. However, unlike lists, tuples are able to store data values that are not of the same type. For example, if we want to ensure that our program functions and exits correctly, we can return a tuple of the desired output value and a symbol designating if the function performed as it should (: *ok* : or : *err* :). In this case, we are able to wrap our return values into a list like structure that we can pass back to our caller. In the following example, we have a game character position represented as an integer. We call a function that tries to move the character one space more, but if the character is at step 15 or more, we don't want to move the character. To represent that this computation fails in this case, we wrap the returning position integer in a tuple with a symbol value representing if the computation moved the character or not.

```
space::Main;

fn main int ()
```

```
{
        characterPosition {int, :sym:} = moveCharacter(4);
        { characterPosition[1] == :ok: ->
                IO::out("character position moved");
        | characterPosition[1] == :err: ->
                IO::out("character position did not move");
        }

        0;
}

// move the character one space, if the character position is
// over 15, do nothing and return an error symbol.
fn moveCharacter {int, :sym:} (position int)
{
        { position >= 15 ->
                return {position + 1, :ok:};
        }
        {position, :err:};
}
```

Tuples can contain any combination of types, even other tuples. Because of this, however, we must explicitly define the structure of the tuple in its type definition. If we want to define a tuple named *rectangle* and we want it to contain all the information we would want including the length of its sides, and whether or not its a square, we would explicitly need to define it as follows: $rectangle\ \{float, float, :\ sym\ :\} = \{3.4, 5.1, :\ rectangle\ :\}$. The more complex the tuple structure, the more complex the definition required.

12 Using Structures to Store Data

Structures are composite data types that are defined by the user and used to group different typed values together. Structures are very useful when you have information that you want to stay together that can easily be referenced. Unlike other types, structures cannot be defined within a funciton body, and instead have to be defined outside all function declarations. After defining how a structure is built, we can build a structure reference by using the name of the structure as the type of the variable. We must define the value of the structure fields in the assignment of the variable. Each field of the structure is ordinal, so the order in which we provide the field values matters. To then access the values we must use dot notation to reference field names of the structure.

```
space::Main;

struct PlayerCharacter {
        name string;
        level int;
        exp float;
};

fn main int ()
{
        playerOne PlayerCharacter = {"Echo", 14, 186.9};

        // print the values to the console
        IO::out(playerOne.name);
        IO::out(playerOne.level);
        IO::out(playerOne.exp);
```

```
        0;
}
```

In the example given above, we define a structure named *PlayerCharacter* that has three fields: *name* as a string, *level* as an int, and *exp* as a float. Then within the main function we create an instance of the *PlayerCharacter* structure and store it within the *PlayerOne* variable. We set each of the fields of the structure in the assinment of the *PlayerOne* variable. Next we reference each field of the *PlayerOne* variable through dot notation to print the values stored in the structure to the console. We can easily create a new instance of the *PlayerCharacter* structure and name it *PlayerTwo* with different values.

13 Including Other Module Spaces

Solace enables the use of different module spaces in programs through the *use* keyword. When building our program, we can state what external module spaces we want to include in our compilation for the compiler to find and link with our current program. There are two different ways we are able to include external module spaces in the current space: the standard inclusion, and the specified inclusion.

```
space::Main;

use::IO;
use::Math::{pow};

fn main int ()
{
        IO::out(pow(10, 2)); // prints 100

        0;
}
```

The standard inclusion, shwocased with the third line *use* :: *IO*; is comprized of the *use* keyword followed by the module space name. This lets the compiler know to include the entire specified module space for use. However, when using a function or structure defined in the included module space, the module name must still be specified before using the desired functionality. The second inclusion possible is shown on the fourth line with *use* :: *Math* :: *pow*;. In this case, we include a small part of the *Math* module space, but only the specified named functionality within the brackets. Then, when we go to use the included functionality, unlike the first inclusion, we are able to use it without specifying what module space it belongs to.

14 Conclusion

Solace is a simple functional programming language built to act as a bridge between the imperive and declarative domains. Solace is built to be easy to read and write programs with. While in the early stages of its development, Solace is capable of handling common types and operations as well as complex data structures. Understandably, there are still some limitations with the language in terms of its capabilities and what it has to offer symantically, but there are plans to expand on the language in the future. Solace is written utilizing Yacc, Bison, and C, with the clang compiler. Solace will compile down to LLVM intermediate code to then be interpreted and compiled down further to machine code.