Scorpion Syntax

This document describes the syntax of our language. Some basic syntax rules are that for line termination you are supposed to put *semicolon(;),* and *brackets : "{}"* are to be used for *if* conditions, *while* loops, *function* definitions.

Reserved Keywords

The reserved keywords for our language are as follows:

```
var char int bool string const arr tuple if elseif else void
func cfunc try throw catch print continue break type head tail slice cons Exception or and
not len for while true false format substr return
main
```

Basic Data Types

Following are the data types used in our language and their small description:

```
int : used to define integers variable size 4 bytes, range -2,147,483,648 to 2,147,483,647

bool : used to define boolean values can have only true or false as values, 1 bit data type

char : used to define character data type. Variable size 1 byte range -128 to 127.

string: used to define strings.
```

We chose C style variable declaration where the data type has to be specified because it makes the code more readable and easy to understand. Also the development of our compiler would be easy if the data type is specified explicitly.

Variable Declaration

This section contains the description as to how to define variables in our language.

```
var - This keyword has to be used to define variable data types. For example to define a
variable of type int the use will be:

var int myInteger;

you can also assign value to the variable at the time of variable defination

var int myInteger = 10;

const - This keyword has to be used to define constant datatypes, its values can't be
changed in the program. example:

const int myInteger = 10;

the basic structure is as follows :

<var/const> <datatype> <name of datatype> = <value> ;
```

We chose the keyword var for data types whose values can be changed and const whose values cannot be changed because we wanted to be as explicit as possible in our language so the code generated is readable, and also it may be easy to make parser this way.
The variable naming convention is the same as in most other languages. It should start with an alphabet or underscore, followed by finite-length combination of alphabets, numbers, and underscores.

Compound Data Types

This section contains the declaration of arrays and tuples, the compound data types.

Array declaration

To declare an array *arr* keyword has to be used at the starting of the declaration.
There are three ways in which you can define an array:

```
1. arr <datatype> <name of variable> = [data1, data2, ...] ;

2. arr <datatype> <name of variable> : <size of array> ;

3. arr <datatype> <name of variable> : <size of array> : <default value of array>;
```

Tuple Declaration

To declare a tuple, *tuple* keyword has to be used at the starting of the declaration.
A tuple can be defined in the following way:

```
tuple <datatype> <name of tuple> = [data1, data2, ...] ;
```

The user does not need to specify the *var, const keyword before array or tuple* declaration because we wanted to avoid redundancy, if a user wants a const array they can use tuple, similarly if the user wants a variable array they should use arr. Also as tuples are immutable we allowed only single type of declaration.

Mutable Variables

Do define mutable variables we defined the *keyword var* for basic data types, while for non mutable datatypes we defined the *keyword const*. Similarly mutable compound variables are arrays and non mutable data types are tuples.

Conditionals

Conditional statements in this language, like in any other, allow us to *control the flow of execution* based on different conditions, providing a structured decision-making mechanism.

We plan to use *if* keyword for the first conditional statement, *elseif* (note- without space) for subsequent cases, and *else* keyword for last declaration.

Curly braces **{}** is to be used after every declaration of a conditional statement and to mark the start and close of that block of code.

Note: We will put brackets even if only one statement is under a condition (unlike in C) to make it uniform and simpler.

Syntax:

```
if (<condition>) {
    <statements> ; (semicolon after each statement)
}
elseif (<condition>) {
    <statements> ; (semicolon after each statement)
}
else {
    <statements> ; (semicolon after each statement)
}
```

Example:

```
var int myInteger = 12;
if (myInteger < 5) {
    print: "number less than 5" ;
}
elseif (myInteger < 10) {
    print: "number less than 10" ;
}
elseif (myInteger <= 15) {
    print: "number less than 16" ;
}
else {
    print: "number greater than 15" ;
}
```

Loops:

The language supports both *while* and *for* loops, enabling execution of repetitive code.

Syntax:

```
while (<conditions>) {
    <statements> ; (semicolon after each statement)
}

for ( <initialisation>; <condition> ; <iteration update>){
    <statements>
}
```

We've maintained a syntax closely resembling that of C/C++ because we found their syntax to be effective and saw no need for significant alterations.

Inside the for loop we have chosen *semicolon* so that we may use multiple iteration variables by separating them with commas.

Like conditionals, we here too follow a *strict opening and closing* {} rule for both types of loop.

Example

```
var int i = 5;
while (i){
    print: i ;
    i--;
```

```
}

for (i=0 ; i<5; i++){
    print: i;
}
```

Branch Control

- **Break**: It is used to *exit* a loop prematurely, stopping further iterations even if the loop condition is still true. It's often used to terminate a loop early based on some condition.
- **Continue**: It is used to *skip* the rest of the current iteration of a loop and continue with the next iteration. It's often used to skip certain values or conditions within a loop.

```
var int i = 5;
while (i){
    print: i ;
    i++;
    if(i == 10){
          break;
    }
}

for (i=0 ; i<5; i++){
        if(i < 2){
          continue;
        }
    print: i;
}
```

Functions:

All function definitions start with the keyword *func*. After that we mention the function name which is then followed by *parenthesis* **()** that has pairs of parameter datatype and parameter name, all comma separated. After the closing bracket we have a colon and the datatype of the return value. Note that all datatypes must be specified. And here too opening and closing **{}** are mandatory.

Syntax:

```
func <func_name>( <datatype> <parameter_name>, …) : <return datatype> {
        // lines of code
        return <value>
}
```

We devised such a syntax because the *func* keyword would make it easier to identify a block of code to be a function definition and the necessary datatype specification would account for better documentation (like in Python) and **type safety** (like in C)

Example:

```
func first( int num1, int num2) : int {
        var int num_sum = num1 + num2;
        return num_sum;
}
```

Closure:

A closure is a function that captures and retains the environment (scope) in which it was defined, including local variables, and can access and manipulate those variables even after the enclosing function has finished executing. In this, a function is defined inside a function such that the local variables of the parent function act as global variables for the child function. We can reuse this child function within the parent function, or even return it as a first-class object for execution in some later part of our code.

We account for this '*child function*' with the keyword *cfunc*. Other than this, it follows the same rules and syntax of a function in our langauge.

Syntax:

```
func <func_name> ( <datatype> <parameter_name>, ...) : func(datatype,datatype,...) : <return
datatype> {
    // lines of code
        cfunc <func_name> ( <datatype> <parameter_name>, ...) : <return datatype> {
                // lines of code
                return
        }
}
```

```
func <func_name> ( <datatype> <parameter_name>, ...) : func(datatype,datatype,...) : <return
datatype> {
    // lines of code
        return cfunc <func_name> ( <datatype> <parameter_name>, ...) : <return datatype> {
                // lines of code
                return
        }
}
```

Example:

```
func outer(int num, int a, int b) : int {

  cfunc inner(int x) : int{
    return num + x;
  }
  return inner(a) * inner(b);
}
```

To declare a variable that can store the closure function as output we can use **type** to first define a function datatype and then use it to declare such variables.

```
func substring(string s) : func(int, int) : string {

    return cfunc slicing(int start_index, int sub_str_size) : string {

        return slice(s, start_index, start_index + sub_str_size);
    }
}

type MyFunc = func(int, int) : string;
MyFunc x = substring("abcdef");
var string sub = x(1,3);
```

Entry Point

Our compiler is going to start reading from the main function, like what happens with languages like C and C++. This is to ensure that the compiler always has a non ambiguous line of code from which it needs to start compiling the code. The syntax is like any function declaration. Eg:

```
func main() : void {
        shrey_joshi Enter your code here
}
```

Operators

- *Basic arithmetic* operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulo)
- *Basic logical* operators: `or`, `and`, `not`
- *Bitwise* operators: `|` (or) , `&` (and), `~` (not)
- *Bit Shift* operators: `<<` (left shift), `>>` (right shift)
- The following table summaries the *usage of the above operators*

`<int>`	`<string>`	`<bool>`
`<int> + <int>`	`<string> + <string>`	`<bool> and <bool>`
`<int> - <int>`	`<string> + <char>`	`<bool> or <bool>`
`<int> * <int>`	`<char> + <string>`	`not <bool>`
`<int> / <int>`		
`<int>` | `<int>`		
`<int> & <int>`		
`~ <int>`		
`<int> >> <int>`		
`<int> << <int>`		

- Also, the **return type** of the operation will remain the same as the 2 operands. Hence, to add 2 chars, the user will need to initialise an empty string and the 2 add chars to them. Example: `<int> / <int>;` -> *returns int*
- Multiple operators with integers are supported, and they will be evaluated using the *BODMAS* rule.
- *Multiple string concatenation* using multiple **+** operators is also supported.
- Available *Shorthand notations*: `<int> += <int>;` , `<string> += <string>;` . In similar fashion, `-=, /=, *=, %=, >>=, <<=` are also supported for `int`.
  Example: `a += b;` is same as

Unary operators:

`<int>++;` *Increments* the value of the integer by 1. The return type is void.

`<int>--;` *Decrements* the value of the integer by 1. The return type is void.

`not bool;` *Logical negation* operator

`~ <int>;` *Bitwise negation* operator

Comparison operators:

Return type is *boolean*.

`a < b` Checks if `a` *Lesser than* `b`.

`a > b` Checks if `a` *Greater than* `b`.

`a <= b` Checks if `a` *Lesser than or equal to* `b`.

`a >= b` Checks if `a` *Greater than or equal to* `b`.

`a == b` Checks if `a` *equal to* `b`.

`a != b` Checks if `a` *not equal* `b`.

Order of Precedence:

Note: Currently this is just *a proposal* regarding resolving precedence in operators, it will be easier for us to decide once we get a hold of the implementation specifics.

The table below outlines the *precedence and associativity* of inspired from C++ operators. The operators are arranged from top to bottom, with higher precedence listed first.

Associativity, establishes the sequence in which operators with the same level of precedence are processed within an expression.

Precedence	Operator	Operation Description	Associativity
1	`a++` `a--` `a[]`	Postfix Increment or decrement Indexing	Left to right
2	`~ a` `not b`	Bitwise NOT Logical NOT	Left to right
3	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	Left to right
4	`a+b` `a-b`	Addition and subtraction	Left to right
5	`<<` `>>`	Bitwise left shift and right shift	Left to right
6	`<` `<=` `>` `>=`	For relational operators	Left to right
7	`==` `!=`	For equality operators = and ≠ respectively	Left to right
8	`a&b`	Bitwise AND	Left to right
9	|	Bitwise OR	Left to right
10	and	Logical AND	Left to right
11	or	Logical OR	Left to right

Print Statements and output

- Basic syntax: `print: <expression>`
- *Python-like string formatting* features are also present.
  Example: `print: " The two variable are {} and {}".format(var1, var2);`
- A newline character is not inserted at the end of any print expression by default, and the user needs mention any of these escape characters.
- *Escape characters* supported: `\n`, `\t`, `\0` (null character), `\\` (using backslash), `\'` (single quote) , `\"` (double quote)
- Support for *direct variable printing* is also available.

Example : `var int a = 5; print : a`. This prints 5.

- *Multiple operands* in the print statement are **not supported**. The flexibility to print multiple variables at once is provided to the user using formatting operations.

Indexing and changing values

- *Arrays, strings, tuples* follow **zero-based** indexing.
- As of now the support is *not extended multi-dimensional arrays and nested tuples*, although we plan to incorporate this feature once we have some clarity with the implementation.
- We use the *square brackets notation* like most languages.This notation is used *access* and *alter values* in arrays and tuples. Changing values in *tuples* is not permitted, since it is an **immutable object**.
- Example:

```
arr int sample_arr = [1, 2, 3];
print: arr[0];
print: '\n';
sample_arr[0] = 5;
print: arr[0];
```

```
The above code return the following ouput:
0
5
```

Slicing of arrays and tuples

- `len(<tuple/array>)` returns the **length** of the *array or tuple*.
- Use `slice(<array/tuple>, start_index, end_index)` : The range is *[start, end]* - (**both inclusive**)
- To *slice till the end of the array/tuple* use: `slice(<array/tuple>, start_index, tail(<array/tuple>))`
- To *slice from start of the array/tuple* use: `slice(<array/tuple>, head(<array/tuple>), end_index)`
- To *insert an element at the start* of an *array/tuple,* use: `cons(<array/tuple>, <element>`. The element needs to be of the same type as the elements in the array/tuple. This is an **inplace operation.**
- This is can also be explicitly defined for arrays and tuples. To include support for both of these in a single function we plan to use function overloading, variadic functions or object oriented programming constructs like methods and attributes.
- For *strings* use `substr(<string>, start, end)` : The range is*[start, end]* - (**both inclusive**)
- Also, the return types for all of these functions is the same as the first operand, except the `cons` function, which is an **inplace function**.
- The `head` and `tail` functions *just return integers* corresponding to the start and end of their arrays/tuples respectively. We plan to add iterators once we a better idea of the implementation details.

Exception Handling

- Use `try-catch` blocks to handle exceptions
- The `catch` block can't be used alone, the `try` block *must precede* it.
- We can handle *specific exceptions*, using `catch(specific_exception spec_ex)`
- To handle *general exceptions*, using `catch(specific_exception spec_ex)`

```
try {
  // block of code to try which could produce errors
}
catch(specific_exception spec_ex) {
  // Block of code to handle specific_exception error
```

```
}
catch(Exception e) {
  // Block of code to handle other types of error
  // not handled by above catch blocks.
}
```

- Additionally, `throw` keyword can to throw *custom errors*.

```
throw Exception_type("error message");
```

Note: As per our current understanding, throwing errors would require us to *implement different classes of error types*, and we are still exploring different ways to provide the same functionality, as we can't yet foresee the *implementation of classes*. Based on future developments, we might make some changes to the above functioning.

Comments

- Structure for *single line comments*, `shrey_joshi <- comment line ->`
- It is **important to notice** that after using `shrey_joshi`, the keyword is followed by space.
- Structure for *multi-line comments*, which is similar to *multi-line comments in c++* `/*` followed by multi line comments or paragraph and end with `*/` to terminate the comment section.

```
/*
Example for multi-line comments
Hello, World!
*/

shrey_joshi This is an example for single line comment
```