Data Types

- int (64 bit, signed)
- bool (boolean)
- char (unicode character in range 0x0 to 0x10FFFF)
- T[] (array of type T (heap allocated))
    - String (real name char[], heap allocated, mutable structure)
- Abstract types (structs/records)

Encapsulation

- Module-based system. A file `module.evo` implicitly has an interface `module.evi`. A module is required to implement all definitions within its interface. If an interface is not found, it is assumed that all top level elements are accessible by other modules.
- A module is represented by its interface and in the case that an interface does not exist, the module is its own interface. Note there is a distinction between the terms “interface” and “interface file” (which refers to .evi files). Cyclic dependency between modules is prohibited. An example of cyclic dependency is when interface A imports interface B and interface B also imports interface A.
- A type can be defined without an implementation in an interface file. These abstract types expose 0 information about the declared types. A type can also be defined with a subset of the fields exposed. These exposed fields can then be accessed through object dot notation in other modules when the interface is imported.

Top Level Declarations

- Imports must precede all other definitions in any given file.
- Function and Type definitions (as well as global variables) are allowed to be given in any order.
- Global variables can only be assigned with constants; In the case of an object, no value other than null can be assigned. Objects are generally mutable and can lead to errors in the user’s program when such objects are shared across modules. A safer approach would be providing getter/setters where the getter only initializes the object once.

Default values for Declarations

- Objects are assigned default value of null
- Integers (also characters) and booleans are assigned 0, the null character, and false respectively.

Function definitions

- Function Overloading is allowed under the condition that all overloaded functions share the same return type(s). Overloading only occurs within a single module. Suppose module M defines a function `f` with return type `T1` and module N also defines a function `f` but with return type `T2`. In this case, `f` is not overloaded (even if `T1` and `T2` are equivalent).
- Functions can return any number of values. In the case that no values are returned, the function is a procedure and it can be optionally modified by the keyword `void`.
- Functions with multiple return values must return the same number of values in every return path. These values are returned as an N item list. The function specifies its return types in an N-tuple.
- When a function `f` is called (overloaded functions are treated as unique functions), the compiler first searches for its implementation in the given source file. In the case that `f` is not a defined local function, the compiler searches for `f` in the imported interface context.
    - If there are definitions of `f` across multiple interfaces, an ambiguity error is raised.
    - If there are no definitions of `f`, an undefined error is raised.
    - Lastly, there is one definition of `f` from some module M, then we simply use M.f()
- To resolve function name ambiguities, the programmer can attach the source module of a given function `f`. For example, if we have two modules M and N, the programmer can use `M.f()` and `N.f()` to be more precise.

Custom Types and Object Instantiations

- When a module defines a type T with a set of fields F, an instance of T can be created by invoking the default instantiation function T(...args). Arguments are passed into the function according to the following rule: each argument is to be of the form FIELDNAME=VALUE
- The fields defined in a given type T must all share unique names. No two fields can share the same name. Fields of different types can share the same name, however.
- A module M that defines a type T implicitly has the constructor T. In the event there is no interface file for M (i.e. module M is its own interface), the full definition of T and its associated constructor will be available to any module importing M. In the event an interface file is present, the interface may choose to expose a subset of the fields in T defined in M by using the syntax type T : {fields}. This will not expose the constructor. The full type definition for T and its associated constructor can be exposed by writing the full type definition in the interface file using the syntax type T = {fields}.

Namespaces

- In a given scope, a variable can only be declared once. Upon entry to a different scope, the same variable name can be used, although this is generally not recommended for good practices. Assignments to a local variable only take effect within the scope of the variable, meanwhile assignments to global variables will always modify the variable.
- Functions and variables have distinct namespaces (i.e. there can be a function named `f` and also a global variable named `f`).

Reserved Keywords

The following is an exhaustive table of *case-sensitive* keywords:

int	bool	char	type
while	for	if	else
continue	break	return	import
const	false	true	null
void			

Order of operations

The following is the table indicating the precedences of operators from highest to lowest, as well as the associativity of the operator.

Precedence	Operator	Description (?)	Associativity
1	( ) [ ] .	function call array access module access and field access	left to right
2	(int) (char) ! ~ -	cast to integer cast to character logical negation bitwise negation unary minus	right to left
3	* / %	multiplication integer division modulus	left to right
4	+ -	binary addition binary subtraction	left to right
5	>< < >= <=	greater than less than greater than or equal to less than or equal to	left to right
6	==	equality test	left to right

	!=	nonequality test	
7	& |	bitwise and bitwise or	left to right
8	&& ||	logical and logical or	left to right