Work and Project solution management

The SkyLine configuration language is a language that was not only designed to modify and work with the internal environment of the SkyLine programming language but also works with the external environment and directories in which a project exists. Currently, the engine is at a state where it can do the bare minimum but this document describes the ideas for what will go into the engine before release. Upon release will be the release of the Caster IoT manipulation interactive and open-sourced framework developed by SkyPenguin Solutions. Below you will find a general example of the workflow and then a section that will deeply explain this workflow.

A base design

Project in the SkyLine programming language may get big, especially for the developers who use their language to do and execute specific and much larger tasks. In order to make sure that the project remains consistent and that the engine itself can be more than just an engine, the base idea of the engine is also to help when working with the external environments of projects. The engine will work in multiple ways, allowing the user to load a file using the load keyword. This file is a Requires.json file that contains some data like the following.

```
{
   "Requirements": {
       "Libraries": [
           "IoT/Roku/ECP",
           "IoT/Apple/Database",
           "http",
           "http/server",
           "io",
           "math",
           "forensics"
       ],
       "Operating-System": "Linux",
       "SLC-Version":"0.0.1",
       "SL-Version":"0.0.5"
   }
}
```

The idea of this file is to specify and check the results and the system requirements as well as the SkyLine requirements in order to run the project. When working with SL and then running the SLC internal to an SL script, this file will be checked and if something goes wrong it will exit.

Technical details

This engine has multiple inner workings so let's first explain the syntax of the engine. Consider the following modifier code or modify_sl code.

```
// Modify the environment
ENGINE(true){
   INIT true {
       load("Requirements.json");
       constant DEFINE_CODE_MISSING_SEMICOLON = 12;
       constant DEFINE_CODE_MISSING_LEFT_BRCE = 109;
       set depth_var = 0;
       set basic_var = true;
       set verbosity = true;
       set debuglev  = true;
       system  |"errors"| -> modify[basic(true)];
       system  |"output"| -> modify[debug(debuglev)];
       system  |"import"| -> modify[expect("directories")];
       system  |"import"| -> modify[allow("*.modifier", "*.csc")];
       library |"IoT"|    -> modify["use:ROKU_ECP"];
       library |"IoT"|    -> modify["use:APPLE_ECP"];
       library |"OPN"|    -> modify["use:GOLANG_PLUGINS"];
       library |"SRV"|    -> modify["use:HTML,CSS"];
   };
};
```

This modifier script allows a person to load the requirements.json file which looks like the same exact code block shown above. In the case that you can not scroll up for whatever reason here is the code to that json file again.

```
{
   "Requirements": {
       "Libraries": [
           "IoT/Roku/ECP",
           "IoT/Apple/Database",
           "http",
           "http/server",
           "io",
           "math",
           "forensics"
       ],
       "Operating-System": "Linux",
       "SLC-Version":"0.0.1",
       "SL-Version":"0.0.5"
   }
}
```

Note: When working with SLC the developers have made it so that the engine itself will only open files that have the same structure. Files can be Requirements.json as that is preferred but if another name is found then the file's structure will be checked and compared to a basic template within the engine that can guide the engine to verify the file's integrity.

When the engine is done checking for the operating system, SLC version, SL version and is done verifying these libraries exist within the language it will then check for a specific projects-based directory. This directory name is named `**ProjectData**`, if it exists then it will look for a file known as `**Project.json**` where the JSON file will hold the following information.

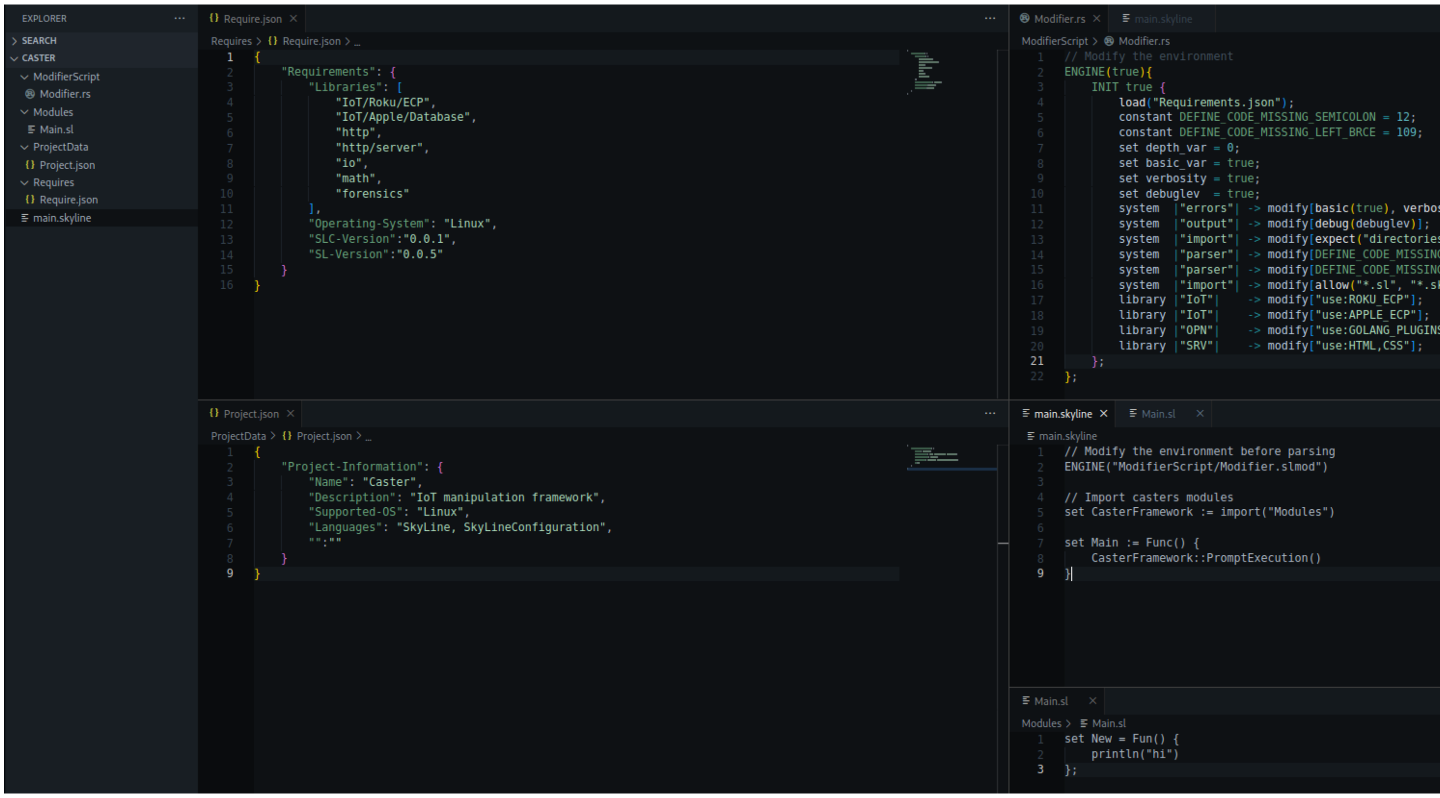
```
{
    "Project-Information": {
        "Name": "Caster",
        "Description": "IoT manipulation framework",
        "Supported-OS": "Linux",
        "Languages": "SkyLine, SkyLineConfiguration",
        "":""
    }
}
```

When the engine comes across this structure and starts to parse data from here, the engine will then tell SkyLine's interpreter to add these as constant variables to your project which will start out with ENGINE_ followed by the name. For example, if you wanted to output the name of the framework you can do the following.

```
println(ENGINE_Name)
```

Because the engine has told SkyLine to pre-register these as constant values in the environment, they can be called during development and during execution. It is also important to note the importance of the `Libraries` field in the Requirements.json file. This field tells the

SkyLine language to import or register all of those libraries within the language. When working with SkyLine individually you will need to use register() to register libraries or standard libraries into the environment, however, when you push it to prod or even beta or hell even testing you can use the Requirements.json file to tell SLC to communicate with SL to tell it to register those values. Now that you understand how SLC's workflow will work, here is a base example of the file paths that will or would be used in a project like a caster (the rewrite).



Require.json — Requires > Require.json > ...

```
{
    "Requirements": {
        "Libraries": [
            "IoT/Roku/ECP",
            "IoT/Apple/Database",
            "http",
            "http/server",
            "io",
            "math",
            "forensics"
        ],
        "Operating-System": "Linux",
        "SLC-Version":"0.0.1",
        "SL-Version":"0.0.5"
    }
}
```

Modifier.rs — ModifierScript > Modifier.rs

```
// Modify the environment
ENGINE(true){
    INIT true {
        load("Requirements.json");
        constant DEFINE_CODE_MISSING_SEMICOLON = 12;
        constant DEFINE_CODE_MISSING_LEFT_BRCE = 109;
        set depth_var = 0;
        set basic_var = true;
        set verbosity = true;
        set debuglev  = true;
        system  |"errors"| -> modify[basic(true), verbos
        system  |"output"| -> modify[debug(debuglev)];
        system  |"import"| -> modify[expect("directories
        system  |"parser"| -> modify[DEFINE_CODE_MISSING
        system  |"parser"| -> modify[DEFINE_CODE_MISSING
        system  |"import"| -> modify[allow("*.sl", "*.sk
        library |"IoT"|    -> modify["use:ROKU_ECP"];
        library |"IoT"|    -> modify["use:APPLE_ECP"];
        library |"OPN"|    -> modify["use:GOLANG_PLUGINS
        library |"SRV"|    -> modify["use:HTML,CSS"];
    };
};
```

Project.json — ProjectData > Project.json > ...

```
{
    "Project-Information": {
        "Name": "Caster",
        "Description": "IoT manipulation framework",
        "Supported-OS": "Linux",
        "Languages": "SkyLine, SkyLineConfiguration",
        "":""
    }
}
```

main.skyline

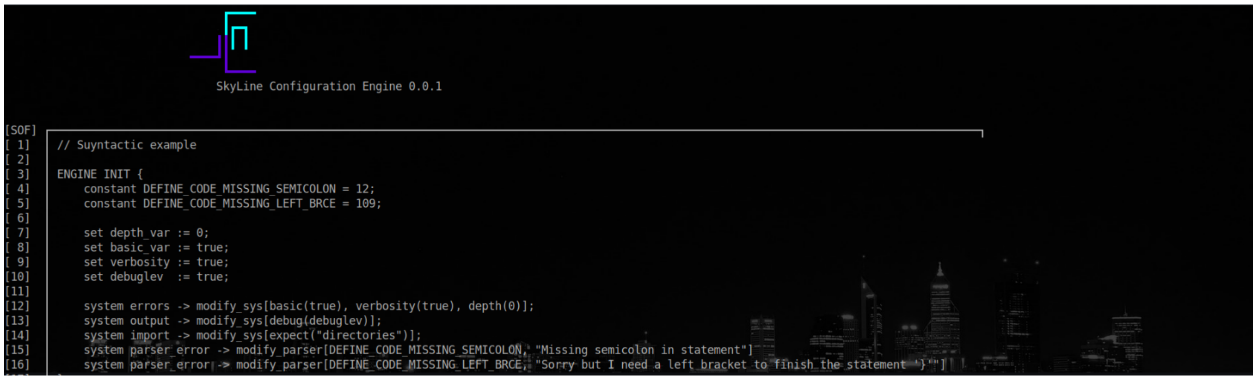
```
// Modify the environment before parsing
ENGINE("ModifierScript/Modifier.slmod")

// Import casters modules
set CasterFramework := import("Modules")

set Main := Func() {
    CasterFramework::PromptExecution()
}
```

Main.sl — Modules > Main.sl

```
set New = Fun() {
    println("hi")
};
```

This is a great example as to how a projects directory should look like and what the files should contain! This is a future idea for SLC in terms of the library modification but everything else is currently being worked on and is a good design plan.

SkyLine Configuration Syntax Example (Passers)

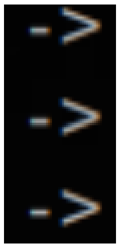


```
SkyLine Configuration Engine 0.0.1

// Suyntactic example

ENGINE INIT {
    constant DEFINE_CODE_MISSING_SEMICOLON = 12;
    constant DEFINE_CODE_MISSING_LEFT_BRCE = 109;

    set depth_var := 0;
    set basic_var := true;
    set verbosity := true;
    set debuglev  := true;

    system errors -> modify_sys[basic(true), verbosity(true), depth(0)];
    system output -> modify_sys[debug(debuglev)];
    system import -> modify_sys[expect("directories")];
    system parser_error -> modify_parser[DEFINE_CODE_MISSING_SEMICOLON, "Missing semicolon in statement"]
    system parser_error -> modify_parser[DEFINE_CODE_MISSING_LEFT_BRCE, "Sorry but I need a left bracket to finish the statement '}'"]
```

This document belongs to SkyPenguinSolutions and is a prime design plan for the SkyLine Configuration engine, an engine that is designed for system and environmental-based configuration for SkyLine. This document explains passers which is a concept that takes the output of one function and then further passes it to another function. The passer symbol or rather operator is defined as an arrow going from one resulting function to another.

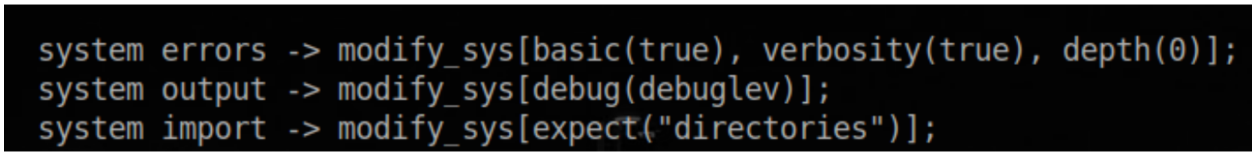
0x00: What are passers defined as?

A passer is defined in the context of SkyLine development as a “director” to a specific function. In the SkyLine configuration engine, the concept of a passer would be to take the left value or result of a function to be passed onto a function known as `modify` on the right-hand side of the symbol. A passer looks like the ones in the images below.



```
->
->
->
```

These are pretty obvious, shown below you will see a syntactic example of what a passer looks like fully kitted with an expression on the left and the right-hand side.

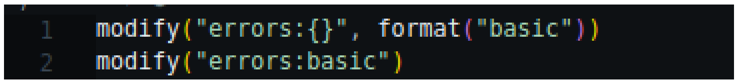


```
system errors -> modify_sys[basic(true), verbosity(true), depth(0)];
system output -> modify_sys[debug(debuglev)];
system import -> modify_sys[expect("directories")];
```

This is the idea of what a passer looks like filled with statements or results which can also be configured to work with strings and only strings. Below is the concept of a passer.

0x01: The concept of a passer and the left and right

A passer is not that hard to explain, first it is important to go over why the SkyLine configuration language and engine for the language exists. The engine is there for a few reasons, instead of making pragmas or constantly calling modify() in every single file that is imported to modify Skyline's environment which includes modifying the error system, parser, or AST for macros. When working with the modification you do not want to completely flood your project with constantly modify statements. A typical modify statement looks like the following.



```
modify("errors:{}", format("basic"))
modify("errors:basic")
```

Constantly typing this for each system and each keyword a developer may want to modify is really a pain to work with. So the idea of adding the configuration language just works.

0x1.5: The bare concept of passers

The idea of passers is to allow people to easily pass values from one function to another in the engine. The bare concept would be to make it easy to declare a system you want to modify. It is important to note that 90% of the engine runs off of just standard functions and the rest is keywords, data types, and base syntax since the engine's only purpose is to make it easier for people to configure their environments during a project. The passers work like so

- **Main operator:** the main operator to define a passer is an arrow '->' or minus and a greater than key. The main operator has two sides and one primary function. These are all shown below in their own bullet points.

- **Left side:** The left side of a passer has to be a data type of type string, this is because a passer only exists to pass values to the modifier (more on this in 0x2). The left side will be either the type of system in quotes like "errors" or will be the result of a system() function call. system() returns a data type of type STRING in the language. The point of the calling system is to allow you to verify the system you want to modify is actually a real system in SkyLine and if it is it will return a string if not then the system call will return a specialized error that will be made aware before parsing the modify statement.

- **Right Side:** The right side of the operator is a bit complex and confusing for some people, the original idea of the right side is to pass the output of the left side to the modified statement(). The modify function call is actually not a function, it works like a function but is actually an array of values that are based on the system. The design of modify and how much data can fit in the modify[] array are going to depend on the type of system you are modifying. In the example above the system of errors takes 3 values but the output system only allows one. This is actually quite easy to understand and develop