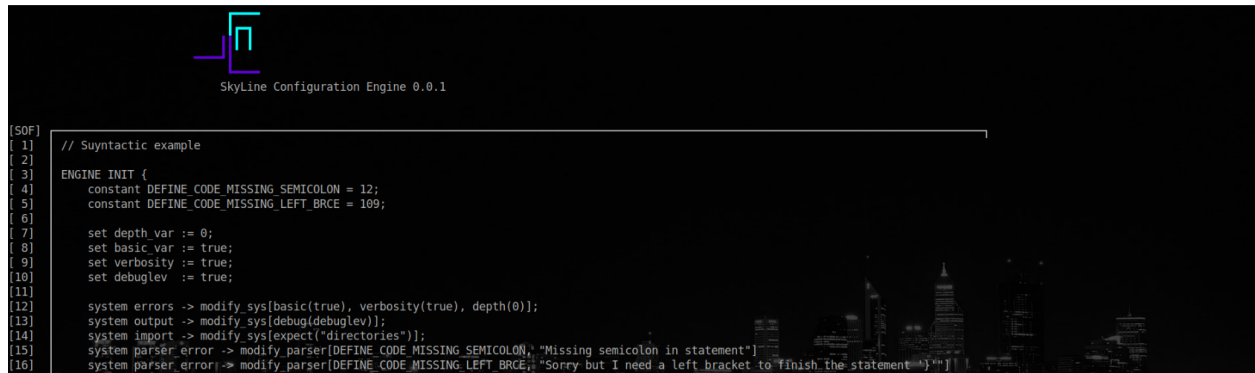


# SkyLine Configuration Syntax Example ( Passers )

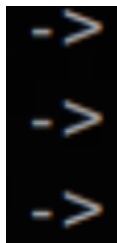


```
[SOF]
[ 1] // Syntactic example
[ 2]
[ 3] ENGINE INIT {
[ 4]     constant DEFINE_CODE_MISSING_SEMICOLON = 12;
[ 5]     constant DEFINE_CODE_MISSING_LEFT_BRCE = 109;
[ 6]
[ 7]     set depth_var := 0;
[ 8]     set basic_var := true;
[ 9]     set verbosity := true;
[10]     set debuglev := true;
[11]
[12]     system errors -> modify_sys[basic(true), verbosity(true), depth(0)];
[13]     system output -> modify_sys[debug(debuglev)];
[14]     system import -> modify_sys[expect("directories")];
[15]     system parser error -> modify_parser[DEFINE_CODE_MISSING_SEMICOLON, "Missing semicolon in statement"];
[16]     system parser_error -> modify_parser[DEFINE_CODE_MISSING_LEFT_BRCE, "Sorry but I need a left bracket to finish the statement ->"];
[17]
```

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.

```
1  modify("errors:{", format("basic"))
2  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

S