# Synthesizer Syntax

Synthesizer Syntax (also referred to as SynthSyntax and SynSyn) is a programming language that compiles into WebAssembly.

It is designed to be a runtime compiler for tools that need a “scripting language” to define creating procedurally generated geometries.

# Compilation Passes

To compile the SynthSyntax, there are several forms of the logic and transformations involved:

## Script

The SynthSyn script is a high-level programming language inspired off C++, C#, and applications that allow editing scripted values in a GUI via reflection.

## Tokenization

SynthSyn scripts are tokenized into low-level symbols. Comments and whitespace information is stripped, and the rest of the script is broken down into atomic symbols called tokens.

## Token Grouping

The syntax for SynthSyn requires grouping for brackets that have an opening and close character. These are parenthesis (), square brackets [] and curly braces {}. Based off this and certain high-priority keywords, tokens can be grouped into nesting scopes.

Some of the nesting scopes that are parsed also involve discovering structs, member variables, and function definitions. Until the entire file is processed, only certain information is reliable, mainly the string names of these things.

## Token Tree

The tokens are then turned into a tree form, where a data-structure hierarchy is formed.

Limited information of types is used to help with forming the hierarchies, but most of the rules for processing are grammar based.

## AST Tree

The token tree is then processed and turned into an AST form. During this process, the understanding of variables, types, and functions are formalized. This includes calculating the size of structs, and their memory alignments.

After memory alignments are known, it is possible to allocate and access values in memory.

## WASM Builder

After the AST phase, a binary WASM representation is built. The most common strategy to perform this is to crawl through the AST, emitting a certain pattern of WASM code for each specific AST node type.

## WASM Instance

To execute the code, the WASM must be loaded into a WASM interpreter.

It should be noted that SynthSyn is designed to be executed with a specific WASM interpreter, Precision WASM.

# Exceptions

Because of how dense and complicated debugging miscompilation issues can be, extensive error checking and diagnostic systems are favored.

When questions of if the use of exceptions in any part of the code are presented, usually the preference is that more testing and thrown errors are better than less.

There are currently 3 main types of assertions that should be used:

## Syntax Exceptions

This is an exception thrown during compiling that is thrown when invalid syntax is discovered.

## Impossible Exceptions

This is an exception thrown during compiling when an unexpected event occurs. When the compiler goes through the various compilation passes, there are a large set of rules and agreements on how the data will be transformed and organized that should make encountering certain situation theoretically impossible.

## Compiler Exceptions

Compile exceptions are breadcrumb exceptions that flag when the compiler is attempting the process code that is currently unsupported.

What makes this different than a syntax exception is that compile exceptions are not technically errors when compiled against a hypothetical finished SynthSyntax implementation. Instead, it is flagging an error to broadcast that something is attempting to compile that should be supported but is not yet [supported].

# Testing

Testing is performed with Unity’s NUnit package.

The tests are fully self-contained scripts that perform calculation and exercise scripting features. This process also involves logging specific values which are checked against an answer key.

## Answer Key

The keys are stored in header comments. Each unit testing script will have a comment with the format   
// >> Results : {\*}  
where the key features are to greater than symbols, the word Results, and a colon. The colon is followed by paired curly braces that contain a set of ordered integers, separated by a comma. The parser for the results should be whitespace insensitive.

These numbers are checked against the values logged when the script is executed. The quantity, order, and value of expected results need to match what was logged during execution for the unit test to pass.

## Test Name

The name of the individual tests are defined in the test comments using the format   
// >> Name : “\*”  
Where the asterisk inside the quotes would contain the name. There is currently no specific convention for how to name the tests.

## Test Comments and Header

All tests are expected to have a name and key section. Beyond that, there are no hard requirements for any additional comments or details, but it is preferred that tests have comments about what is being tested.