GIL 1.0 Specification

This document outlines GIL 1.0, in an effort to make compiler development simpler. This is a living document and will be updated in future versions. I will outline the basic syntax and keywords of a GIL program, talk about several features of GIL, and provide examples of proper syntax throughout.

Contents

---

1. What is GIL and what should it be used for?

GIL (Genetic Intermediate Language) is a programming language for synthetic biology. Its main goal is to simplify synthetic biology constructs to make them easier for biologists to design. The core feature of GIL is reusability. GIL allows users to write libraries of reusable genetic code to make the development process simpler.

2. Syntax

2.1. Preprocessor commands

GIL defines a preprocessor command as a '#' followed by a single word, followed by an arbitrary number of arguments. Below is a list of preprocessor commands with their syntax. Any arguments surrounded by square brackets are optional

Region preprocessor commands

1. `#Region Name`
2. `#region Name`
3. `#EndRegion [Name]`
4. `#endRegion [Name]`

Target organism preprocessor commands

1. `#Target "Organism Name"`
2. `#target "Organism Name"`

`#SetAttr` and `#setAttr`

These preprocessor commands can be used to alter one of the compiler's behaviours. While the following attributes are required to be implemented, any usage of `#SetAttr` should set the attribute specified.

Syntax: `#SetAttr "KeyName" "Value"`

Misc preprocessor commands

1. `#Var Type Name [value]` - creates a variable with type Type and name Name. Type can be either `str` or `num`
2. `#Inc VarName` - increments the variable of type `num` and name `VarName`
3. `#If Condition(expression)` - Executes a block of code if the condition is true. Conditions are either built-in or added by modules. The condition `IsEqual(Var1 Var2)` must be built-in
4. `#Else` - Executes a block of code if the previous condition wasn't true
5. `#EndIf` - ends the block of code to be executed.

2.2. Comments

GIL defines two types of C-style comments. A single-line comment can be specified using two forward slashes. The compiler will ignore everything from the slashes until the next newline character, though it may include comments in CGIL files.

A multiline comment may be specified using `/* ... */`. Anything in between the stars will be ignored by the compiler, although it may be included in CGIL files.

2.3. Literals

2.3.1. DNA literals

GIL defines a DNA literal as a sequence containing only [A, C, G, T, a, c, g, t]. DNA literals are added to the compiler's output as-is.

2.3.2. Amino acid literals

GIL defines an amino acid literal as at least one amino acid single-letter code in between two `@` symbols. Because amino acid literals are designed to represent a protein, the compiler may modify the DNA sequence generated to increase translation efficiency in the target organism, but the DNA sequence generated must be translated to the original amino acid sequence in the target organism.

For example, the compiler may add additional elements to the sequence, but these elements must be either (1) untranslated, (2) removed by splicing, or (3) removed by post-translational modification.

2.4. Target organism

For most applications, the GIL compiler requires information about the target organism in order to optimize the DNA generated. The user can specify which organism they wish to target using the preprocessor commands `#Target` and `#target`. This command loads an organism definition file with the same name as the organism specified.

2.4.1. Organism definition file

All GIL compilers should support organism definition files ending with the file extensions ".organism" or ".Organism", though they may accept other extensions that use the same format. To maintain interoperability, GIL compilers may not accept other file formats for setting the target organism. Instead, they may extend the native organism definition file format.

An organism definition file is divided into sections. Sections are titled using `#SectionName`. If the compiler reaches a section with a name that it does not recognize, it should skip the section and continue with compilation, though it may output a warning to the console.

There are two sections of an organism definition file that are required and defined here. They are `OrganismInfo` and `Codons`. A compiler may define additional sections with additional information not included here.

Finally, if the first line of an organism definition file is `goto organism name`, the compiler should load the organism definition file associated with the other organism name. In this way, multiple aliases may be given for an organism that all point to the same definition file.

OrganismInfo

This section contains information about the species of the organism being targeted. Below is a list of fields and their possible values:

1. `Domain:[P or E or A]`
2. `SupportsPlasmids:[Y or N]`

These fields will be expanded as the compiler develops further.

Codons

This section contains the codon(s) that code for an amino acid, along with their availabilities. Each codon is specified using `Code:Codon1.Availability Codon2.Availability ...`. While the upper-case IUPAC amino acid codes should be used for canonical amino acids, non-canonical amino acids are also supported. In theory, any single-letter code may be used.

2.5. Identifiers

GIL defines an identifier as a sequence of characters beginning with one of the (uppercase or lowercase) characters a-z or _ and continuing to the next space, newline, carriage return, tab, colon, or right or left parentheses. A regex expression for this would be `[^ ^\n^\r^\t^:^\(^\)]*`.

Finally, an identifier must contain at least one character that is not (uppercase or lowercase) A, C, G, or T. This is to distinguish between DNA literals and identifiers.

2.6. Regions

GIL defines a region as a named section of DNA. Multiple regions may overlap, and the output of a GIL program that has been compiled to DNA should include a list of regions associated with that DNA.

GIL defines several preprocessor commands to begin and end regions. `#Region` and `#region` begin a region with the name given. `#EndRegion` and `#endRegion` allow the user to specify the name of the region to end, but will end the last region to be created if no name is specified.

2.7. Sequences

Sequences are the core construct used by a GIL program. While it is possible to write a GIL program using only regions (or without using regions), this is not advisable because it removes the ability to reuse this code. Instead, users should use sequences.

A sequence is defined using the keyword `sequence`, followed by an identifier, followed by a list of arguments enclosed in parentheses. For example, `sequence MySequence(Param1, Param2)`. A sequence may optionally have a type. If it has a type, the type name should be specified before the sequence keyword. If the sequence has no parameters, the parentheses are not required.

The sequence's parameters may also optionally have a type. If they have a type, the type will be specified before the parameter name just like with sequences. Sequences may also accept other sequences with parameters as parameters. To be clear, all parameters of a sequence **must** be other sequences, but they may or may not accept parameters themselves.

The sequence's definition begins with a curly bracket after the closing parentheses, separated only by whitespace characters and/or comments. One of the sequence's parameters may be used with the syntax `$ParamName`. If the parameter is a sequence with arguments, the sequence may be "called" just like a regular sequence, only with a dollar sign before the name of the sequence.

If a sequence's first parameter is `$InnerCode`, it is treated as an operation.

Sequences may be "called" in two ways. First, you can just add the name of the sequence to part of the code. This will call the sequence with any parameters it has set to their defaults. In most cases, this is a sequence containing no DNA.

Sequences may also be called with parameters. This is done by adding the name of the sequence, followed by a list of arguments enclosed in parentheses. If you have too few arguments, a warning will be output to the console and the rest of the arguments will be set to their defaults. If you have too many arguments, a warning will be output to the console and the rest of the arguments will be discarded.

The exception to this is when a sequence is passed as an argument of another sequence and called from within the sequence. In this case, no warning will be printed, since the sequence was added further upstream.

Both of these behaviours can be changed by the `#SetAttr` or `#setAttr` preprocessor commands. Setting the attribute `ErrorOnSequenceParamMismatch` to `true` (or anything other than `false` or an empty string) will make it so that GIL throws an error instead of printing a warning when the wrong number of parameters are given.

Setting the attributes `WarnSequenceAsParameterMismatch` or `ErrorOnSequenceAsParameterMismatch` will send either errors or warnings when the wrong number of parameters are given to a sequence passed as a parameter.

Hoisting

In GIL, the order of definition of any element does not matter. Every element's dependencies **must** be representable as a directed acyclic graph, and an element's dependencies will be compiled/defined before the element.

2.8. Operations

An operation is an extended sequence. Operations are just like sequences, only using the `operation` keyword. However, operations have an implicit `InnerCode` argument as their first argument. Operations are called using `.OperationName(Parameters)`, followed by a block of GIL code enclosed in curly brackets.

Like with sequence and operation definitions, the opening curly bracket must be separated by only whitespace or comments. Everything within this block is compiled and passed as the `InnerCode` parameter. Operations may only have one `InnerCode` parameter.

2.9. Forwards

When writing libraries of GIL code, it is useful to have multiple aliases for a sequence or operation. This can be done using the syntax `[operation or sequence] NewName => OldName`. Forwards must be included in CGIL files and must be treated exactly the same as normal sequences and operations syntactically.

GIL compilers must support forwards to sequences or operations in other namespaces.

2.10. Types

GIL allows for optional types to help limit what data can be passed to sequences and operations. Anything without the specified type has the `any` type. The `any` type can be reinterpreted as any other type. Below is a list of built-in types:

1. `any`
2. `cds` - a bit of DNA intended to be transcribed
    1. `protein inherits cds` - a bit of DNA intended to be translated
3. `ncds` - a bit of DNA that isn't transcribed
    1. `promoter inherits ncds`
    2. `bool inherits ncds` - special type, see booleans
4. `data` - for internal use, a wrapper around a pointer or reference

GIL also allows the definition of additional types using the `typedef` keyword. The compiler must also make the function `GetTypeID` available to modules. This function should accept the type name as a parameter and return a unique type ID that can be used by the module.

Type inheritance can be specified using the `inherits` keyword after the `typedef` keyword. If a type inherits from another type, data with this type can be used for parameters that require the type it inherits from. The compiler must also make the function `SetInheritance` available to modules.

If the attribute `AllowDownstreamTypeConversion` is set to `true`, this type conversion can work in the other direction. In this case, type `ncds` could be implicitly converted to `promoter`. This attribute should default to `false`.

2.10.1. Data type

Certain sequences returned by modules can be type `data`. These are references to data controlled by the compiler or modules and should only be used to pass data to other sequences added by modules. Attempting to call a sequence of type `data` will throw an error and terminate compilation. `data` is also the only type that can't be passed to a parameter of type `any`. If a sequence has a parameter of type `data`, this must be explicitly written.

While the `data` type is intended to be a single reference to an object in memory, implementations using programming languages without garbage collection may include an additional pointer to a function for freeing the memory used by the `data`.

2.11. Operators

GIL allows users to define almost any operator. GIL only comes with a couple of built-in operators (listed below), but almost any other operator can be created.

Operators may be defined just like sequences using the `operator` keyword. However, an operator's name is less restricted than a sequence. An operator's name may contain any of the characters found in an identifier, but it can also start with any of the following symbols: ~, `, !, @, $, %, ^, &, *, :, ;, ', <, >, ?, -, _, +, and =.

If an operator starts with the symbols @ or :, it must be at least two characters long. If an operator starts with a :, the second character must not be another :.

Operators must have one or two parameters. If the operator has one parameter, it will be given the sequence immediately after it as a parameter if it exists. If there isn't a sequence immediately after it, the operator will be given the sequence immediately behind it as a parameter. For example, if the user defines an operator `n#` that accepts one parameter, `(a n# b)` is equivalent to `(a (n# b))` and `(b n#)` is equivalent to `(n# b)`.

If an operator has two parameters, it should be placed in between its operands.

Operators may also be created using forwards. In this case, you'd use the syntax `operator x* => SomeSequence`. Operators may also have types. Like with sequences, this is specified before the `operator` keyword.

Operators may also be overloaded by the user. If this is the case, the special parameter `$_prev` will point to the last version of this operator. This way, operator overloads may be defined that work under certain conditions but point to the previous operator if these conditions are not met.

List of built-in operators

1. Forward operator `=>` (not overloadable)
2. Namespace accession operator `::` (not overloadable)
3. AND operator `&` (implemented by modules)
4. OR operator `|` (implemented by modules)
5. NOT operator `!` (implemented by modules)

2.12. Booleans

GIL allows the construction of genetic circuits using an interface similar to C-style languages. Boolean inputs can be bound to any `ncds` type operation (keep in mind that this includes types that inherit from `ncds`). An input is bound using a forward with the syntax `bool BoolName => SomePromoter`. The operation should be structured in a way that regulates the `InnerCode` parameter. For example, you could define an operation to represent the Tet-On promoter and define a boolean input that's bound to this promoter using the code `bool TetIsPresent => Tet-On`.

Once a boolean has been defined, it can be applied to any `cds` type sequence(s) using an `if` statement. An `if` statement consists of the `if` keyword followed by an expression and a block of code. The expression may be a single boolean or an expression contained within parentheses. For example,

```
bool TetIsPresent => Tet-On


if TetIsPresent
```

```
{
        SomeSequence
        AnotherSequence
}
```

The above code is equivalent to

```
.Tet-On
{
        SomeSequence
}

.Tet-On
{
        AnotherSequence
}
```

GIL also supports genetic logic gates. To allow for this, boolean variables need to be created. The difference between a boolean variable and a bound boolean input is that inputs cannot be changed and are just an alias for an operation.

New boolean variable types can be defined using the `boolean` keyword. This is different from the `bool` type. The `boolean` keyword defines a new type of bool, while the `bool` type is used when using this new boolean type.

A boolean definition is more similar to a C++ class definition than a sequence definition. A boolean definition follows the following syntax:

```
boolean BoolType : [DNA or RNA]
{
        _new => SequenceToCreateBool    //data type
        _set => SequenceToSetBoolTrue    //cds type
        _use => SequenceToRegulateUsingBool    //bool type
}
```

The colon after the bool type tells the compiler whether the boolean regulates at the DNA or RNA level. If the boolean regulates the transcript at the RNA level, the compiler will wrap the contents in the default promoter. For example, the code

```
bool B1 = someBool    //At RNA level, ExampleType
if B1
{
        SomeSequence
}
```

is equivalent to

```
bool B1 = someBool    //At RNA level, ExampleType
.DefaultPromoter
{
        if B1
        {
                SomeSequence
        }
}
```

which is equivalent to

```
data sequence B1 => ExampleType::_new    //At RNA level, ExampleType
.DefaultPromoter
{
        ExampleType:: ._use(B1)
        {
                SomeSequence
        }
}
```

A boolean definition may contain more forwards than this, but it must contain forwards for `_new`, `_set`, and `_use`. Forwards present in the definition may be accessed using the syntax `BoolType::Forward`.

Each of these forwards is used under different circumstances. `_new` is called internally by the compiler before the boolean is first used. This sequence should be type `data` and return both some sort of ID (this will depend on the implementation) that can be used to differentiate the instances of this boolean definition and a boolean value (in the programming language of the compiler) to tell the compiler if the bool was successfully created.

`_set` is a sequence of type `cds` with a single parameter of type `data`. This sequence returns a sequence that will activate any sequences controlled by the bool when transcribed. As such, the code

```
bool TetIsPresent => Tet-On
bool SomeBool = TetIsPresent    //Type ExampleType
```

is equivalent to

```
bool TetIsPresent => Tet-On
data sequence SomeBool => ExampleType::_new

if TetIsPresent
{
        ExampleType::_set(SomeBool)
}
```

which is equivalent to

```
data sequence SomeBool => ExampleType::_new

.Tet-On
{
        ExampleType::_set(SomeBool)
}
```

Finally, `_use` is the operation called when an instance of this boolean is used in an if statement. So, the code

```
bool TetIsPresent => Tet-On
bool SomeBool = TetIsPresent    //Type ExampleType

if SomeBool
{
        SomeSequence
}
```

is equivalent to

```
bool TetIsPresent => Tet-On
data sequence SomeBool => ExampleType::_new

//Setting SomeBool to TetIsPresent omitted

if ExampleType:: ._use(SomeBool)
{
        SomeSequence
}
```

which is equivalent to

```
bool TetIsPresent => Tet-On
data sequence SomeBool => ExampleType::_new

//Setting SomeBool to TetIsPresent omitted

.ExampleType:: ._use(SomeBool)
{
        SomeSequence
}
```

To be clear, the sequences pointed to by `_new`, `_set`, and `_use` should be sequences added by modules.

The GIL compiler must support multiple user-defined boolean types. These definitions are added to a stack. The most recently defined definition will be used until `_new` returns a failure. When this happens, this definition is popped from the stack and the next most recent definition is used. If a bool ever fails to be created due to all the implementations failing, the compiler throws an error and terminates compilation. The compiler may allow the user to specify the specific bool type by adding the type name before the `bool` keyword, but this is not required. If the compiler does not support this, it should print a warning **or** throw an error when it encounters it.

Finally, there are genetic logic gates. This will depend on the bool implementation used, but all boolean definitions should also define logic gates to work with these bool types. If two bools used in a logic gate are different types and no logic gate exists that supports both of them, the second bool will be converted to the type of the first bool. If this fails, the first bool will be converted to the type of the second bool. If this fails, both bools will be converted to a new type. For example, the code

```
bool b1 = SomeCondition    //ExampleType
bool b2 = AnotherCondition    //ExampleType2

if (SomeCondition & AnotherCondition)
{
        SomeSequence
}
```

is equivalent to

```
bool b1 = SomeCondition    //ExampleType
bool b2 = AnotherCondition    //ExampleType2

ExampleType bool Temp = AnotherCondition

if (SomeCondition & Temp)
{
        SomeSequence
}
```

If the type of bools that must be converted was not specified, the compiler may change their types to minimize conversions for logic gates.

If a logic gate's type is not specified, it is assumed to be the type of the first argument.

2.13. Namespaces

Just like in other programming languages, GIL allows the user to create namespaces to isolate certain operations/sequences. A namespace may be created using the `namespace` keyword, followed by the namespace name and a block of GIL code. A namespace may be accessed using the namespace name, followed by the `::` operator. If files and/or libraries are imported into a namespace, the library will be stored within this namespace. As such, if the file `SomeFile.gil` contains the sequence `MySequence`, the following code would be incorrect:

```
namespace Names
{
        import SomeFile.gil
}

SomeFile::MySequence
```

The correct code would be

```
namespace Names
{
        import SomeFile.gil
}

Names::SomeFile::MySequence
```

When a sequence is called from within a namespace without specifying which namespace it is in, the GIL compiler should first look for a matching sequence in the namespace the sequence was called from. If it fails to find one, it should check parent namespaces until it finds a match or reaches the global namespace. As such, the namespace with the location `Global::Space1::Space2::Space3` will be searched in the order `Space3`, `Space2`, `Space1`, `Global`.

2.14. API

GIL requires that certain functions are exposed to GIL programs through modules. Below is a list of which functions are required:

1. `AddCodeStart` - Operation that adds `InnerCode` to the beginning of the GIL program
2. `AddCodeEnd` - Operation that adds `InnerCode` to the end of the GIL program
3. `CompileNewOutputFile` - Takes the name of the file as a parameter and compiles `InnerCode` and inserts the output into this file

2.15. Libraries

GIL supports either importing other GIL files as a library or linking to external code to call functions from within GIL.

2.15.1. GIL files

Other GIL files may be imported using the `import` keyword, followed by the name of the file (excluding file extension) in double-quotes. When a GIL or CGIL file is imported, it is put into a namespace with the same name as the name of the file. If the file name contains spaces, the spaces will be replaced with underscores in the namespace name.

The `include` keyword works slightly differently. `include` effectively pastes the contents of the file that was included in place of the `include` statement. As such, the file extension must be added, and include statements should not be used for CGIL files. Generally, the `import` keyword should be used, as this isolates the file by name and supports CGIL files.

When an imported file contains other imports, these files will also be imported into the imported file's namespace. As such, if the file `SomeFile.gil` had the following code and was imported into another file, the contents of `Another File.gil` could be accessed by the file doing the importing at the location `SomeFile::Another_File`.

```
import "Another File"

//Some more code
```

While circular imports may be supported by the GIL compiler, all sequences and/or operations **must** be able to have their dependencies be represented using a directed acyclic graph. This means that circular dependencies are not allowed, and the following code **must** throw an error.

```
sequence Seq1
{
        //Some code
        Seq2
}

sequence Seq2
{
        //Some code
        Seq1
}
```

From looking at the above code, it should be clear why circular dependencies can't be allowed. If the above code was compiled, it would cause infinite recursion and cause another error either by taking up all the program's available memory or by causing a stack overflow.

While circular imports are theoretically allowed, circular includes must be prohibited for the same reason.

CGIL

Because of the inefficiency of lexing and parsing source files multiple times, the compiler must provide the option to compile to an intermediate CGIL format. This file format is explained later in this document. The compiler may also allow files to be precompiled to CGIL files to cut down on compile time.

Search order

When importing or including files, the compiler must search for matching files first in the current working directory. If it fails to find a match, it should search for the matching file in a global directory for libraries shared between different programs.

2.15.2. Modules

GIL allows the user to link to and call functions supplied in external code. This can be done using the `using` keyword, followed by the name of the module surrounded in double-quotes without the file extension. The format of modules may differ between different compilers, but all modules should have a way to expose functions as sequences and operations to a GIL program. Like with imports, all module imports are added to their own namespace.

2.15.3. Organism-specific libraries

Certain libraries may need to behave differently or expose different sequences depending on the target organism. For example, certain genetic circuits can only work in prokaryotes or eukaryotes and shouldn't be used in incompatible organisms.

GIL allows a library to be specified for a specific organism, as well as variants or add-ons of a library. This can be done by adding `OrganismName@` before the name of the file. When GIL imports a library or links to a module, it first imports/links to variations of the library and then imports the library (if it exists). A library can also be specified for a specific domain by adding `Domain@` to the start of the file. Below is the order in which GIL imports files:

1. `Domain@Organism@File`
2. `Organism@File`
3. `Domain@File`
4. `File`

3. CGIL format

Compiled GIL (CGIL) files are a binary format that can be used to represent parsed GIL files. Below is the CGIL format for GIL 1.0

1. GIL major version (1 byte)
2. GIL minor version (1 byte)
3. GIL fix (2 bytes)
4. Target organism (String)
5. Number of operations (4 bytes)
6. Operations (Operation)
7. Number of sequences (4 bytes)
8. Sequences (Sequence)
9. Number of tokens in entry point (4 bytes)
10. Entry point (Token)
11. Number of imports (4 bytes)
12. Imports (String)
13. Number of includes (4 bytes)
14. Includes (String)
15. Number of modules (4 bytes)
16. Modules (String)
17. Number of namespaces (4 bytes)
18. Namespaces (embedded CGIL file)

String

1. Length (4 bytes)
2. Characters (1 byte * Length)

Operation

Will be defined further on in development

Sequence

Will be defined further on in development

Token

Will be defined further on in development