References!!

Appendix C – Directives:

Directives are part of the language, no pre-processor as in C is needed.

#add_context `see Context:` adds a declaration to a context; Your main program, and any modules that you use, can add things to the Context.

#align used to align struct member fields relative to the start of the struct.

#as indicates that a struct can implicitly cast to one of its members. It is similar to `using`, except #as does not also import the names. `#as` works on non-struct-typed members. For example, you can make a struct with a float member, mark that #as, and pass that struct implicitly to any procedure taking a float argument.

#asm specifies that the next statements in a block are inline assembly.

#assert does a compile-time assert. This is useful for debugging compile-time meta-programming bugs.

#bake_constants generate a compiled procedure with predefined values for type variables

#bake_arguments provide specific values to a procedure at compile time

does a compile-time currying of a procedure/parameterized struct.

#c_call `follows C ABI conventions:` makes the function to use the C calling convention. Used for interacting with libraries written in C.

#caller_location `it gives the line number from where a procedure is called.`

#caller_code fills out its Code's parent scope. #caller_code to help create macros that make it easier to implement the kind of thing you might do from an external metaprogram. See how_to/497_caller_code.jai for details.

#char `see § 12.1 /` see 2.3 Fundamental types: makes the next one character string after it into a single ASCII character (e.g. #char "A").

#code specifies that the next statement/block is a code type.

#complete `see if-case:` Ensure an if-case statement checks all values of the enum

#compiler is a function that interfaces with the compiler as a library. The proc is internal to the compiler.

#compile_time evaluate to `true` if execution is occuring during compile time / is a boolean value that evaluates to `true` during compile time and `false` during runtime.

#cpp_method allows one to specify a C++ calling convention.

#cpp_return_type_is_non_pod allows one to specify that the return type of a function is a C++ class, for calling convention purposes `(pod = plain old data)`

#deprecated marks a function as deprecated. Calling a deprecated function leads to a compiler warning

#dump dumps out the bytecode and basic blocks used to construct the function. This is useful for viewing the disassembly of the bytecode.

#expand marks the proc as a macro.

#file evaluate to the name of the current source file / path+filename of running executable

`#filepath`	path of the currently running executable gets the current filepath of the program as a string
`#foreign`	instruct compiler to link against a foreign library / specifies a foreign procedure
`#if`	`Compiling conditionally with #if:, #else does not exist, use else`
`#import`	bring a library file into scope / takes foreign modules located in the Jai `modules` directory and compile the library into your program.
`#insert`	inserts a piece of compile-time generated code into a function or a struct.
`#insert, scope()`	similar to `#insert`, except it also allows `code` to access variables in the local scope.
`#intrinsic`	marks a function that is handled specifically by the compiler, like memcpy, memcmp, memset (see Preload.jai)
`#library`	provide a library for the compiler to link against for procedures marked with `#foreign` directive. specifies file for foreign functions
`#line`	evaluate to the line number of the current statement
`#load`	bring a source code file into scope / Load source code, as if it were placed right here; takes `Jai` code files written by the programmer and adds the files to your project.
`#location`	`gives the location of a piece of Code`
`#modify`	`used in polymorphic procedures:` filter polymorphic parameter type

lets one put a block of code that is executed at compile-time each time a call to that procedure is resolved. One can inspect parameter types at compile-time; Goal: to filter or check on polymorphic parameter type / provide a function to manipulate a type variable prior to it being used in a polymorphic procedure

`#module_parameters`	specifies the variable as a module parameter.
`#must`	requires the caller to assign / use the particular return value of the called function. Used primarily for malloc or opening file handles.
`#no_abc`	Turn off bounds checking for the scope of a particular array/string access: in this function, do not do array bounds checking
`#no_context`	tells the compiler that the function does not use the context.
`#no_alias`	
`#no_padding`	tells the compiler to do no padding for this struct.
`#no_reset`	lets one store data in the executable's global data, without having to write it out as text.
`#place`	a way of forming a union data type with a struct; Set location in struct of following members
`#placeholder`	specifies to the compiler that a particular symbol will be defined/generated by the compile-time metaprogram.
`#program_export`	`see Program entry point`
`#procedure_name`	gives you the statically-known-at-compile-time name of a procedure.

`#procedure_of_call` `see § 22.2.1`

`#program_export` ?? used in modules Runtime_Support.jai and Program_Print

`#run` execute `<code>` at compile time (not run time) / takes the function in question and runs that function at compile time (e.g. `PI :: #run compute_pi();`).

`#runtime_support` Proc comes from runtime support.

`#scope_export` makes the function accessible to the entire program

`#scope_file` makes the function only callable within the current file.

`#scope_module` makes the function only callable within the current module.

`#specified` requires values of an enum to explicitly be initialized to a specific value. An enum marked specified will not auto-increment, and every value of the enum must be declared explicitly.; Declare intention of maintaining enum values compatibility over time.

`#string` `see § 12.1 / #string<token>` Parse the next lines as a string up to the next occurrence of the token: used to specify a multi-line string.

`#symmetric` `operator overloading (commutativity):` allows to swap the 1st and 2nd parameters in a two parameter function. Useful in the case of operator overloading.

`#system_library` specifies system file for foreign functions

`#this` tells the compiler that the next following syntax is a type. Useful for resolving ambiguous type grammar; it returns the procedure, struct type, or data scope that contains it *`as a compile-time constant. (see 050)`*

`#through` `see if-case:` case fall-through

`#type` tells the compiler that the next following syntax is a type. Useful for resolving ambiguous type grammar. `(see § 26.13)`

`TYPE VARIANTS:`

`#type,` `distinct`

`#type,` `isa`

`#type_info_none` marks a struct such that the struct will not generate the type information; Struct does not keep runtime type info.

`#type_info_procedures_are_void_pointers`

makes all the member procedures of a struct void pointers when generating type information. See Type_Info_Struct_Member.Flags.PROCEDURE_WITH_VOID_POINTER_TYPE_INFO.

`#type_info_no_size_compliant` prevents the compiler from complaining about the size of the type information generated by a struct