Appendix C – Directives:

Directives are part of the language, they are not part of a pre-processor as in C.

In the see § references below, the bold reference defines the directive, the other references mostly contains examples of use.

(Count: Jan 2023 - 60)

#add_context adds a declaration to Context. The main program, and any modules that you use, can add things to the Context*: see §* **25.1**, 25.9

#align used to align struct member fields relative to the start of the struct*: see §* ***12.11****, 18.3*

#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*: see §* ***12.8****, 16.2, 16.3.6, 17.13.2, 23.9, 23B*

#asm specifies that the next statements in a block are inline assembly: *see § 28*

#assert does a compile-time assert. This is useful for debugging compile-time meta-programming bugs: *see §* ***20.2.1****, 20.1.2, 26.3, 51.2*

#bake_constants generate a compiled procedure with predefined values for type variables: see § 23.7, 23.8

#bake_arguments provide specific values to a procedure at compile time; does a compile-time currying of a procedure/parameterized struct: see § 22.6, 23.3, 23.7

#bytes inline binary data or machine code

#c_call follows C ABI conventions: makes the function to use the C calling convention. Used for interacting with libraries written in C: *see §* 4.11, 25.1, 23.13, 29.7, 30.14

#caller_location it gives the line number from where a procedure is called: *see §* 25.5, 30.2

#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: *see* § 26.14

#char makes the next one character string after it into a single ASCII character (e.g. #char "A"): *see §* ***5.1.2****, 5.3.1, 6.2, 19.1, 19.2*

#code specifies that the next statement/block has type Code : *see §* ***26.4.1****, 26.5.6,26.12, 34.3.3,*

#complete Ensures an if-case statement checks all values of the enum: *see §* 13.2, **14.3, 14.3.1**

#compiler is a function that interfaces with the compiler as a library; the proc is internal to the compiler: *see § 26.12, intro § 30*

#compile_time evaluates to true if execution is occurring during compile time / `false` during runtime: *see §* 26.2.1

#cpp_method allows one to specify a C++ calling convention: *see §* /

#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): *see § /*

#deprecated marks a function as deprecated. Calling a deprecated function leads to a compiler warning: *see* § 17.15

#dump dumps out the bytecode and basic blocks used to construct the function. This is useful for viewing the disassembly of the bytecode: *see § 20.2.3*

#expand marks the proc as a macro: *see §* 25.9, 26.4.1, **26.5** – 26.8, 28.2, 30.13, 31.4, 33.6, 34.3.3, 51

#file evaluate to the name of the current source file / path+filename of running executable: *see § 30.2*

#filepath path of the currently running executable: *see §* 30.2, 30.2, 30.6, 30.12

#foreign instruct compiler to link against a foreign library / specifies a foreign procedure: *see §* 19.8, **29.2**, 29.3, 29.6.2, 29.8, 30.14, 50.2,

#if Compiling conditionally with #if:, #else does not exist, use else: *see § 4E, 20.2.1, 22.6, **26.3,** 29.8, 31.2.3,*

#import brings a library file into scope / takes foreign modules located in the Jai `modules` directory and compile the library into your program: *see § **8.1**, 8.3, 8.4*

- **#import, file** see § 8.4
- **#import, dir** see § 8.4
- **#import, string** see § 8.4

#insert inserts a piece of compile-time generated code into a function or a struct: *see § **26.4**, 26.5.1, 26.10.2*

34.3.3

- **#insert,scope()** similar to `#insert`, except it also allows `code` to access variables in the local scope: *see § 26.5.4*

#intrinsic marks a function that is handled specifically by the compiler, like memcpy, memcmp, memset (see Preload.jai): *see § 4C*

#library provide a library for the compiler to link against for procedures marked with #foreign directive / specifies file for foreign functions: see § **29.3**, 29.6.2, 30.14

#line evaluate to the line number of the current statement: see *§ 30.2*

#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: *see §* 8.1, **8.2**, 26.3, 30.3.1

#location gives the location of a piece of Code: *see § 30.2*

#modify used in polymorphic procedures to filter polymorphic parameter type: *see § **26.9***

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: *see § **8.7***

#must requires the caller to assign / use the particular return values of the called function. Used primarily for malloc or opening file handles: *see § **17.6.2,** 19.6.2, 22.2.3*

#no_abc Turn off bounds checking for the scope of a particular array/string access: in this function, do not do array bounds checking: *see § **18.3.2**, 22.1*

#no_alias *see §*

#no_context tells the compiler that the function does not use the context: *see § **25.4***

#no_debug tag a macro / macro-call with #no_debug to prevent the compiler from generating any debug line info for that macro: *see §*

#no_padding tells the compiler to do no padding for this struct: *see §* ***12.11***

#no_reset lets one store data in the executable's global data, without having to write it out as text: *see §* ***26.2.2***

#place a way of forming a union data type with a struct; Set location in struct of following members: *see §* ***13.1***

#placeholder specifies to the compiler that a particular symbol will be defined/generated by the compile-time metaprogram: *see §* **30.3.3**

#program_export see Program entry point / used in modules Runtime_Support.jai and Program_Print: *see § 4E, 30.14*

#procedure_name gives you the statically-known-at-compile-time name of a procedure: *see §* **17.14.1**

#procedure_of_call shows what procedure would be called in a particular case of a polymorphic proc *see §* **22.2.1**

#run execute <code> at compile time (not run time) / takes the function in question and runs that function at compile time: *see § 2B,* ***3.3.4****, 4.2, 4B, 5.2.2, 8.7.2, 10.3, 16.3.5, 20.2.2, 26.2.3, 26.4, 30.1, 30.3.3, 30.4.9*

#runtime_support Proc comes from module Runtime_Support: *see §* 4E, **25.1**

#scope_export makes the function accessible to the entire program: *see §* **8B.1**

#scope_file makes the function only callable within the current file: *see §* **8B.1**

#scope_module makes the function only callable within the current module: *see §* **8B.2**

#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: *see §* **13.4,** 16.3

#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: *see §* **19.1***, 26.4, 27.1, 30.3.2, 34.4*

#symmetric operator overloading (commutativity): allows to swap the 1st and 2nd parameters in a two parameter function. Useful in the case of operator overloading: *see §* **24.2**

#system_library specifies system file for foreign functions: *see § 2.2.6, 19.8,* ***29.2****, 29.8, 50.2*

#this it returns the procedure, struct type, or data scope that contains it as a compile-time constant: *see § 17.9,* ***17.9.1****, 17.9.2, 22.5,23.7, 23.9,*

#through see if-case fall-through: *see §* ***14.3***

#type tells the compiler that the next statement is a type. Useful for resolving ambiguous type grammar: *see §* 21.2, **26.13**, 29.7, 31.2.1, 51

TYPE VARIANTS:

#type,distinct *see §* **26.13**

#type,isa *see §* **26.13**

#type_info_none marks a struct such that the struct will not generate the type information; Struct does not keep runtime type info: *see §* **16.3.5**

#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: *see §* **16.3.5**

#type_info_no_size_compliant prevents the compiler from complaining about the size of the type information generated by a struct: *see §*