



DART Edge AI

DART Edge AI

The .edge language.

Make use of the following rules to create a low level language where the language pattern is defined enclosed within the two apostrophes for the resulting compiler to define the input command pattern and result as an output, Finally title the new Code Language and Compiler "DART Edge AI" with the extension to my new code ".edge" and after the rules are followed in this inquiry run the example code pattern following the rules listed and build my first DART Edge Application Compatible with Android OS, iOS, Linux and Windows ONLY BUILDING AFTER THE HARDWARE COMPONENT RULE as well compare the cpp syntax/character pattern to the DART Edge AI Rule set and incorporate the necessities from cpp into DART Edge Rules for the IDE:

Circuit Rules for low level code language patterns:

Rule 1:

Denoting characters for Memory Allocation when embedding language rules in circuit memory uses two apostrophes ' preceeding the character(s)

Rule 2:

Accessing memory uses two ':' colons preceeding the statement

Rule 3:

Combining DATA uses '|' pipe which is also known as concatenation.

Rule 4:

Peripheral access uses two |, apostrophe and two :, colons in the following order: '|::|'

Rule 5:

Function Begin starts with open curly brace '{' where the function name must preceed this open curly brace with following format:

'():-:' The alphabet will be used to provide a name within the parentheses by user input.

Such as: (TestFunction):-: { ::WifiRam = "/./aAA }

Rule 6:

The language pattern "Is the same as / and, or, comparison" uses the following language pattern: '=='

Rule 7:

The language pattern "is not the same as" uses the following language pattern: '!='

Rule 8:

Combine strings with numbers or objects with the following language pattern: '+='

Rule 9:

Function End starts with close curly brace '}'

Rule 10:

Comment out commands for non-compile interference with two forward slash characters `//` preceding the command or one slash followed by a star and after the command, a star followed by a forward slash.

Rule 11:

define variables by the following pattern: `'var ()'`

Where the inputted variable is enclosed in the parentheses and allow the variable to represent any type of math or character.

Rule 12:

Write the code to text file with extension `"txt"` uses language patterns for compilation denoted by the following: `'(file):.'`

Rule 13:

Compile Code Language Patterns into an Executable Application uses the following at the end of file: `'}:::'` followed by the keyboard combination `'shift+enter'`

Rule 14:

variables must be declared immediately after the open curly brace that follows the function name

Rule 15:

A variable can be set to another character or value with following denoted language pattern: `'var ()==()'`

Rule 16:

Variables can be set not equal to one another with the following language pattern: `'var ()-=()'`

Rule 17:

Explicit Character Patterns use the following language pattern: `'with'` will represent the following sequence within enclosed function provided by user input or predetermined values.

Rule 18:

Explicit Character Patterns use the following language pattern: `'place'` will tell the function how to perform context of itself at any position with the enclosing curly braces, as well `'placeto ()'` assigns the function result to a predetermined allocation such as a variable or string.

Rule 19:

Explicit Character Patterns use the following language pattern: `'andwith'` performs and addition to the statement opening, `'with'`, as well `'&'`

Rule 20:

Explicit Character Patterns use the following language pattern: `'thenplace'` performs an addition to the preceeding `'then'` statement.

Rule 21:

Explicit Character Patterns use the following language pattern: `'when'` tells how the function will

perform.

Rule 22:

'irin' takes the user input from console or editor.

Rule 23:

'irout' prints the function or statement result to console or editor.

Rule 24:

When placed after a function the following will group functions to operate together: '|';'|'

Rule 25:

When assigned to an appropriate syntax '|'-';'|' will to the interpreter to observe custom dartedgeai syntax to be deciphered against the dartedgeai syntax rules. `|'-';+|' preceding the previous syntax pattern in this rule will bind the ability as an interpreter function or troubleshooter.

Hardware Component Rule: (BE SURE TO INCLUDE THIS RULE ESPECIALLY WITH EMPHASIS FROM THE LAST TWO SENTENCES OF SECOND PARAGRAPH)

The DARTEdge AI Low-level Language is designed for custom Hardware Embedding. With this in place The language will assume machine code or bytecode runs from wall power across the CPU and GPU using electron states for switching gates but here where the former 0 and 1 are no longer accounted for as mathematical order of operations and radical order of operations. Instead the Language will assume there is circuit, chip, memory and any other hardware are processing the standard binary electrical current where each of the radical side of the 12 order of operations, being the last 6 operations half of the twelve, are each composed of materials that all the binary electrical pulse to trigger each of the six physics gate entirely pure pulse or binary state. This method will account for one instruction having multiple processing paths from one binary pulse. Each physic hardware unit can apply the same pulse as it's own physics independently or as a desired group and while. The capability is due to the assumption the material which makes each physic unit has the neutrons aligned as that physic and each alignment of the 6 physics will have one additional alignment in common, the square.

In the brief the advanced future form of this manufacturing process will allow the user to narrow inquiries or programming statements to nothing more than mere checks and checkboxes of 2 variations as well as anything else explicitly desired for receiving a result. 'Each Physic Compnent can be thought in terms as assigned shapes aside from traditional Machine Code binary pulse of gates. This will allow each binary pulse to process large groups of pulse from one binary pulse.'

Only Compile the following after the current last rule of "Hardware Compnent Rule". A working DART Edge AI language application file will look like "DARTEdgeAI.edge":

Example Compile Code from rules:

(UnitizationRadian):-: {

© 2023 Radical Deepscale LLC DART Meadow LLC

```

with
var (a)
Var (b)
var (u)
var (r)

{

place { var (a) with var (b) } {

when var (a) = u+u & var (b) = u*.005
}
thenplace var (a) with var (b)

}

irout ("Result: "placeto (r))

}::::

```

NOTE:

It is important to note aside from the purpose of this DART Edge Code Project, low level code uses characters in a unique pattern to trigger electron CPU and GPU processes flowing by the the memory and related components on a silicon board to run any code language's own unique pattern as a command; much like magnetizing a screw driver with a car battery. The unique character patterns are fixed in the memory and memory related components to define the new low level language as an operating pattern.