DART Edge AI

Table of Contents

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 BUILING 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:

Note:
Lexer - Formulates Rule Syntax.
Parser - Constructs Rule Syntax. Such as using the new formulated syntax language "dartedgeai.edge" as a build and run application file.

Lexer Circuit Rules for low level code language patterns:

Rule 0:
Brackets ‘[]’ will contextualize any syntax, statement, variable and function contained within. The preferred method of coding with brackets is to use them in the opening or header of the DART Edge AI code file(s). Brackets may as well contextualize imported files and libraries that are abiding by this bracket syntax rule.

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.

End of Lexer Circuit Rules.

Parser Circuit Rules:
Emulate The Parser for the Lexer against a common Lowlevel language. (exp. C++) within an Opensource Embedable IDE.

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. A math and physic combination Example function.
[(UnitizationRadian.txt):: | (UnitizationRadian)]

(UnitizationRadian):-: {

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

{

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

when var (a) =  u+u & var (b) = u*0.005
}
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.

// Section 1 Emulated FPGA Hardware for Html Editor

//Embedded DART Edge AI FPGA EMULATOR
//https://raw.githubusercontent.com/radicaldeepscale/DARTEdgeAI/main/EmbeddedDARTEdgeAI.js

```
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DARTLexer {
    public List<String> lex(String code) {
        List<String> tokens = new ArrayList<>();

        // Rule 0: Opening and Closing Brackets
        applyRule(code, "\\(.*\\)", "BRACKETS BLOCK", tokens);

        // Rule 1: Denoting characters for Memory Allocation
        applyRule(code, "'.*", "DENOTION BLOCK", tokens);

        // Rule 2: Accessing memory
        applyRule(code, "::.*", "ACCESS BLOCK", tokens);

        // Rule 3: Combining DATA
        applyRule(code, "\\|.*", "COMBINING BLOCK", tokens);

        // Rule 4: Peripheral access
        applyRule(code, "\\|::\\|.*", "PERIPHERAL BLOCK", tokens);

        // Rule 5: Function Begin
        applyRule(code, "\\(.*\\):-: \\{.*", "FUNCTION BEGIN BLOCK", tokens);

        // Rule 6: Language pattern "Is the same as / and, or, comparison"
        applyRule(code, "==.*", "COMPARISON BLOCK", tokens);

        // Rule 7: Language pattern "is not the same as"
        applyRule(code, "-=.*", "NOT BLOCK", tokens);

        // Rule 8: Combine strings with numbers or objects
        applyRule(code, "\\+=.*", "STRINGS OR OBJECTS BLOCK", tokens);

        // Rule 9: Function End
        applyRule(code, "}.*", "FUNCTION END BLOCK", tokens);

        // Rule 10: Comment out commands
        applyRule(code, "//.*", "COMMENT BLOCK 1", tokens);
        applyRule(code, "/\\*.*\\*/", "COMMENT BLOCK 2", tokens);

        // Rule 11: Define variables
        applyRule(code, "var \\(.*\\).*", "VARIABLE BLOCK", tokens);

        // Rule 12: Write code to text file
        applyRule(code, "\\(file\\)::.*", "WRITE TO FILE BLOCK", tokens);
```

```
    // Rule 13: Compile Code Language Patterns
    applyRule(code, "}:::.*", "FINISH CODE FOR BUILD AND EXECUTE BLOCK", tokens);

    // Rule 14: Variables must be declared immediately after the open curly brace
    applyRule(code, "\\{.*var \\(.*\\).*", "VARIABLE DECLARATION BLOCK", tokens);

    // Rule 15: Variable assignment
    applyRule(code, "\\(\\).*==\\(\\).*", "VARIABLE ASSIGNMENT BLOCK", tokens);

    // Rule 16: Variables can be set not equal to one another
    applyRule(code, "\\(\\).*-=\\(\\).*", "VARIABLE SET TO NOT BLOCK", tokens);

    // Rule 17: Explicit Character Patterns - 'with'
    applyRule(code, "with.*", "WITH BLOCK", tokens);

    // Rule 18: Explicit Character Patterns - 'place'
    applyRule(code, "place.*", "PLACE BLOCK", tokens);

    // Rule 19: Explicit Character Patterns - 'andwith'
    applyRule(code, "andwith.*", "COMBINE STATEMENTS BLOCK 1", tokens);

    // Rule 20: Explicit Character Patterns - 'thenplace'
    applyRule(code, "thenplace.*", "ADD TO STATEMENT BLOCK", tokens);

    // Rule 21: Explicit Character Patterns - 'when'
    applyRule(code, "when.*", "STATEMENT OPERATION CONTEXT BLOCK", tokens);

    // Rule 22: 'irin' takes the user input from console or editor
    applyRule(code, "irin.*", "TAKE USER INPUT BLOCK", tokens);

    // Rule 23: 'irout' prints the function or statement result to console or editor
    applyRule(code, "irout.*", "DISPLAY OUTPUT BLOCK", tokens);

    // Rule 24: Grouping functions
    applyRule(code, "\\|';'\\|.*", "GROUP FUNCTION BLOCK", tokens);

    // Rule 25: Binding as interpreter function or troubleshooter
    applyRule(code, "\\|'-';\\|.*", "INTERPRETER SYNTAX ACCESS BLOCK 1", tokens);

    return tokens;
}

private void applyRule(String code, String regex, String tokenType, List<String> tokens) {
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(code);

    while (matcher.find()) {
```

```
            String token = matcher.group().trim();
            tokens.add(token + " (" + tokenType + ")");
        }
    }
}

// Section 2 Operand Check for loaded .edge in Html Editor

function lex(code) {
  let tokens = [];

  for (let i = 0; i < code.length; i++) {
    let currentChar = code.charAt(i);

    // Rule 0: Opening and Closing Brackets
    if (currentChar === "(") {
      let bracketsBlock = "";
      i++;
      while (i < code.length && code.charAt(i) !== ")") {
      bracketsBlock += code.charAt(i);
      i++;
     }
     tokens.push({ type: "BRACKETS BLOCK", value: bracketsBlock });
     continue;
   }

    // Rule 1: Denoting characters for Memory Allocation
    if (currentChar === """) {
      let denotionBlock = "";
      while (i < code.length && code.charAt(i) !== """) {
        denotionBlock += code.charAt(i);
        i++;
      }
      tokens.push({ type: "DENOTION BLOCK", value: denotionBlock });
      continue;
    }

    // Rule 2: Accessing memory
    if (currentChar === ":" && code.charAt(i + 1) === ":") {
      let accessBlock = "";
      i += 2;
      while (i < code.length && code.charAt(i) !== "::") {
        accessBlock += code.charAt(i);
        i++;
      }
      tokens.push({ type: "ACCESS BLOCK", value: accessBlock });
      continue;
    }
```

```
    // Rule 3: Combining DATA
    if (currentChar === "|") {
      let combiningBlock = "";
      while (i < code.length && code.charAt(i) !== "|") {
        combiningBlock += code.charAt(i);
        i++;
      }
      tokens.push({ type: "COMBINING BLOCK", value: combiningBlock });
      continue;
    }

    // Rule 4: Peripheral access
    if (currentChar === "|" && code.charAt(i + 1) === ":" && code.charAt(i + 2) === "|") {
      let peripheralBlock = "";
      i += 3;
      while (i < code.length && code.charAt(i) !== "|") {
        peripheralBlock += code.charAt(i);
        i++;
      }
      tokens.push({ type: "PERIPHERAL BLOCK", value: peripheralBlock });
      continue;
    }

    // Rule 5: Function Begin
    if (currentChar === "(" && code.charAt(i + 1) === ":" && code.charAt(i + 2) === "-" &&
code.charAt(i + 3) === ":" && code.charAt(i + 4) === " ") {
      let functionBeginBlock = "";
      i += 5;
      while (i < code.length && code.charAt(i) !== "{") {
        functionBeginBlock += code.charAt(i);
        i++;
      }
      tokens.push({ type: "FUNCTION BEGIN BLOCK", value: functionBeginBlock });
      continue;
    }

    // Rule 6: Language pattern "Is the same as / and, or, comparison"
    if (currentChar === "=" && code.charAt(i + 1) === "=") {
      let comparisonBlock = "";
      i += 2;
      while (i < code.length && code.charAt(i) !== "=") {
        comparisonBlock += code.charAt(i);
        i++;
      }
      tokens.push({ type: "COMPARISON BLOCK", value: comparisonBlock });
      continue;
    }
```

```
// Rule 7: Language pattern "is not the same as"
if (currentChar === "-" && code.charAt(i + 1) === "=") {
  let notBlock = "";
  i += 2;
  while (i < code.length && code.charAt(i) !== "=") {
    notBlock += code.charAt(i);
    i++;
  }
  tokens.push({ type: "NOT BLOCK", value: notBlock });
  continue;
}

// Rule 8: Combine strings with numbers or objects
    if (currentChar === "+" && code.charAt(i + 1) === "=") {
  let stringsOrObjectsBlock = "";
  i += 2;
  while (i < code.length && code.charAt(i) !== "=") {
    stringsOrObjectsBlock += code.charAt(i);
    i++;
  }
  tokens.push({ type: "STRINGS OR OBJECTS BLOCK", value: stringsOrObjectsBlock });
  continue;
}

// Rule 9: Function End
if (currentChar === "}") {
  tokens.push({ type: "FUNCTION END BLOCK" });
  continue;
}

// Rule 10: Comment out commands
if (currentChar === "/" && code.charAt(i + 1) === "/") {
  let commentBlock = "";
  i += 2;
  while (i < code.length && code.charAt(i) !== "\n") {
    commentBlock += code.charAt(i);
    i++;
  }
  tokens.push({ type: "COMMENT BLOCK 1", value: commentBlock });
  continue;
}

if (currentChar === "/" && code.charAt(i + 1) === "*" && code.charAt(i + 2) === "*") {
  let commentBlock = "";
  i += 3;
  while (i < code.length && code.charAt(i) !== "*" && code.charAt(i + 1) !== "/") {
    commentBlock += code.charAt(i);
```

```
      i++;
    }
    tokens.push({ type: "COMMENT BLOCK 2", value: commentBlock });
    continue;
  }

  // Rule 11: Define variables
  if (currentChar === "v" && code.substr(i, 4) === "var (") {
    let variableBlock = "";
    i += 4;
    while (i < code.length && code.charAt(i) !== ")") {
      variableBlock += code.charAt(i);
      i++;
    }
    tokens.push({ type: "VARIABLE BLOCK", value: variableBlock });
    continue;
  }

  // Rule 12: Write code to text file
  if (currentChar === "(" && code.substr(i, 7) === "(file)::") {
    let writeFileBlock = "";
    i += 7;
    while (i < code.length && code.charAt(i) !== ")") {
      writeFileBlock += code.charAt(i);
      i++;
    }
    tokens.push({ type: "WRITE TO FILE BLOCK", value: writeFileBlock });
    continue;
  }

  // Rule 13: Compile Code Language Patterns
  if (currentChar === "}" && code.substr(i, 4) === "}:::") {
    let finishCodeBlock = "";
    i += 4;
    while (i < code.length) {
      finishCodeBlock += code.charAt(i);
      i++;
    }
    tokens.push({ type: "FINISH CODE FOR BUILD AND EXECUTE BLOCK", value:
finishCodeBlock });
    continue;
  }

  // Rule 14: Variables must be declared immediately after the open curly brace
  if (currentChar === "{" && code.substr(i + 1, 4) === "var (") {
    let variableDeclarationBlock = "";
    i += 5;
    while (i < code.length && code.charAt(i) !== ")") {
```

```
      variableDeclarationBlock += code.charAt(i);
      i++;
    }
    tokens.push({ type: "VARIABLE DECLARATION BLOCK", value: variableDeclarationBlock
});
    continue;
  }

  // Rule 15: Variable assignment
        if (currentChar === "(" && code.charAt(i + 1) === ")" && code.charAt(i + 2) === "=" &&
code.charAt(i + 3) === "(" && code.charAt(i + 4) === ")") {
    let variableAssignmentBlock = "";
    i += 5;
    while (i < code.length && code.charAt(i) !== "=") {
      variableAssignmentBlock += code.charAt(i);
      i++;
    }
    tokens.push({ type: "VARIABLE ASSIGNMENT BLOCK", value: variableAssignmentBlock });
    continue;
  }

  // Rule 16: Variables can be set not equal to one another
  if (currentChar === "(" && code.charAt(i + 1) === ")" && code.charAt(i + 2) === "-" &&
code.charAt(i + 3) === "=" && code.charAt(i + 4) === "(" && code.charAt(i + 5) === ")") {
    let variableSetToNotBlock = "";
    i += 6;
    while (i < code.length && code.charAt(i) !== "=") {
      variableSetToNotBlock += code.charAt(i);
      i++;
    }
    tokens.push({ type: "VARIABLE SET TO NOT BLOCK", value: variableSetToNotBlock });
    continue;
  }

  // Rule 17: Explicit Character Patterns - 'with'
  if (currentChar === "w" && code.substr(i, 4) === "with") {
    let withBlock = "";
    i += 4;
    while (i < code.length && code.charAt(i) !== " ") {
      withBlock += code.charAt(i);
      i++;
    }
    tokens.push({ type: "WITH BLOCK", value: withBlock });
    continue;
  }

  // Rule 18: Explicit Character Patterns - 'place'
  if (currentChar === "p" && code.substr(i, 5) === "place") {
```

```
    let placeBlock = "";
    i += 5;
    while (i < code.length && code.charAt(i) !== " ") {
      placeBlock += code.charAt(i);
      i++;
    }
    tokens.push({ type: "PLACE BLOCK", value: placeBlock });
    continue;
  }

  // Rule 19: Explicit Character Patterns - 'andwith'
  if (currentChar === "a" && code.substr(i, 7) === "andwith") {
    let combineStatementsBlock1 = "";
    i += 7;
    while (i < code.length && code.charAt(i) !== " ") {
      combineStatementsBlock1 += code.charAt(i);
      i++;
    }
    tokens.push({ type: "COMBINE STATEMENTS BLOCK 1", value: combineStatementsBlock1
});
    continue;
  }

  // Rule 20: Explicit Character Patterns - 'thenplace'
  if (currentChar === "t" && code.substr(i, 10) === "thenplace") {
    let addToStatementBlock = "";
    i += 10;
    while (i < code.length && code.charAt(i) !== " ") {
      addToStatementBlock += code.charAt(i);
      i++;
    }
    tokens.push({ type: "ADD TO STATEMENT BLOCK", value: addToStatementBlock });
    continue;
  }

  // Rule 21: Explicit Character Patterns - 'when'
  if (currentChar === "w" && code.substr(i, 4) === "when") {
    let statementOperationContextBlock = "";
    i += 4;
    while (i < code.length && code.charAt(i) !== " ") {
      statementOperationContextBlock += code.charAt(i);
      i++;
    }
    tokens.push


  // Rule 22: 'irin' takes the user input from console or editor
  if (currentChar === "i" && code.substr(i, 4) === "irin") {
```

```
      let takeUserInputBlock = "";
      i += 4;
      while (i < code.length && code.charAt(i) !== " ") {
        takeUserInputBlock += code.charAt(i);
        i++;
      }
      tokens.push({ type: "TAKE USER INPUT BLOCK", value: takeUserInputBlock });
      continue;
    }

    // Rule 23: 'irout' prints the function or statement result to console or editor
    if (currentChar === "i" && code.substr(i, 5) === "irout") {
      let displayOutputBlock = "";
      i += 5;
      while (i < code.length && code.charAt(i) !== " ") {
        displayOutputBlock += code.charAt(i);
        i++;
      }
      tokens.push({ type: "DISPLAY OUTPUT BLOCK", value: displayOutputBlock });
      continue;
    }

    // Rule 24: Grouping functions
    if (currentChar === "|" && code.charAt(i + 1) === "'" && code.charAt(i + 2) === ";" &&
code.charAt(i + 3) === "'" && code.charAt(i + 4) === "|") {
      let groupFunctionBlock = "";
      i += 5;
      while (i < code.length && code.charAt(i) !== "|") {
        groupFunctionBlock += code.charAt(i);
        i++;
      }
      tokens.push({ type: "GROUP FUNCTION BLOCK", value: groupFunctionBlock });
      continue;
    }

    // Rule 25: Binding as interpreter function or troubleshooter
    if (currentChar === "|" && code.charAt(i + 1) === "'" && code.charAt(i + 2) === "-" &&
code.charAt(i + 3) === "'" && code.charAt(i + 4) === ";" && code.charAt(i + 5) === "'" &&
code.charAt(i + 6) === "|") {
      let interpreterSyntaxAccessBlock1 = "";
      i += 7;
      while (i < code.length && code.charAt(i) !== "|") {
        interpreterSyntaxAccessBlock1 += code.charAt(i);
        i++;
      }
      tokens.push({ type: "INTERPRETER SYNTAX ACCESS BLOCK 1", value:
interpreterSyntaxAccessBlock1 });
      continue;
```

```
    }
  }

  return tokens;
}
```