BitGate Documentation

BitGate programming language is a general-purpose language and consists of a "mix" of imperative and object-oriented language that implements an imperative and object-oriented paradigm. It is a "mix" of Object-Oriented Language, Functional Language and of an Algorithmic Language because it is based on code blocks and in classes. The idea that a language like BitGate should be implemented was born when I was working on logic gates and there was no language that showed the results of two numbers whether they are decimal, Octal, Binary or Hexadecimal. BitGate performs mathematical operations without symbols and performs logical expressions between numbers in decimal, binary, octal or hexadecimal form. The syntax rules are that it works in classes, where each class necessarily consists of a "Headfunc" function in which the mathematical operations, the logical expressions, the "Whether-Do-Otherwise-Do" functions, the "During-Do", functions, void functions etc. will be written.

To declare a class, the word "CLASS" must first be written, followed by a name (i.e., the identifier) which will start strictly with an underscore. Then follow the left bracket and the right bracket where the "Headfunc" function must be declared inside. An example of a class declaration is the following:

```
CLASS _firstClass ()
{
}
```

To declare the Headfunc function, first, the word "INTEGER" must be declared, then the word "HEADFUNC", followed by left and right parenthesis. Then, follow the left bracket and the right bracket where the variables and all functions, pointers, arrays etc. are declared. An example of the Headfunc function declaration is the following:

```
INTEGER HEADFUNC ()
{
}
```

There are two ways to declare a variable. The first way is to declare a variable without assigning a value. Thus, the type of the variable (INTEGER) must be declared first, then its name (ie the identifier) which must start strictly with a underscore (_firstVariable) and at the end there must be the semicolon (;). This variable can be initialized below with the following grammar: First we declare the identifier of the variable (_firstVariable), then follows the symbol "=" with the value of the variable (= 5) and finally the semicolon. The second way is to do the above in a line, ie to declare and initialize the variable immediately such as: "INTEGER _ firstVariable = 5;".

To declare a function, you must first declare the type of the variable (INTEGER, VOID, FLOAT), then a name (i.e., the identifier) which will start strictly with an underscore such as "_firstFunction", followed by left and right parenthesis where within them we can incorporate variables or values. At the end follows the semicolon (;). If we want the function to do some work, then we add after the semicolon the left and the right brace and inside them we declare array, or functions. An example of declaring a function is the following:

```
VOID _firstFunction ();   or        INTEGER _firstFunction (INTEGER _A, INTEGER _B);        or
INTEGER _firstFunction (INTEGER _A, INTEGER _B);
[
]
```

To declare and perform logical expressions or mathematical operations we simply write for example the operations in binary form: 0b011001 AND 0b10010;, for the operations in octal form 0O4322 NAND 0o2421 ;, for the operations in hexadecimal form: 0xFAD XOR 0XABC;, and for operations in decimal form: 20 OR 5; or 20 MULTIPLY 5; or _A SUBSTRACT 10 ; or 20 POWER 4 ;, etc. Non-decimal variables must be declared as "CHARROW" type.

To declare the function "DURING" the grammar to be followed by an example is as follows:
DURING(_integer2>=10) DO
[
]
Where in parentheses we write an expression and in braces we declare array, or functions.

To declare the function "WHETHER" the grammar to be followed by an example is as follows:
WHETHER(_integer2>=10) DO
[
]
OTHERWISE DO
[
]
Where in parentheses we write an expression and in braces we declare array, or functions.

To declare an array the way is the same as declaring variables with the only difference that after the symbol "=" follow braces where inside contains the number that indicates the sum of the positions of the array. An example of an array is the following:
INTEGER _array = [5];
To find the address of the array in the computer memory we use pointers. The memory is displayed on the screen with the WRITE () function as follows:
WRITE (& _ array);

To read an input from the user's keyboard we can use the READ () function as follows:
INTEGER _integer; ➔ First, we declare the variable.
READ (_integer); ➔ The user input is then stored in the variable.

The declaration of Single Line Comment and Multiline Comment is as follows:
Single Line Comment: $$ This is a Singe Line Comment
Multiline Comment: #@ This is a Multiline Comment @#

The language consists of 51 tokens which are declared along with the type of each token in the "parser.y" file which this part is in the image. The creation of its grammar language was based on the programming languages C, C ++ and Java, thus becoming a unique language.

```
26  %token <intNum>      TOKEN_INTEGERCONST
27  %token <floatNum>    TOKEN_FLOATCONST
28  %token <aCharacter>  TOKEN_CHARACTERCONST
29  %token <charrowVar>  TOKEN_CHARROWCONST
30  %token <charrowVar>  TOKEN_IDENTIFIER
31  %token <intNum>      TOKEN_OCTACONST
32  %token <intNum>      TOKEN_BINARYCONST
33  %token <intNum>      TOKEN_HEXACONST
34  %token <trueVar>     TOKEN_TRUE
35  %token <falseVar>    TOKEN_FALSE
36
37  %token <charrowVar>  TOKEN_INTEGER
38  %token <charrowVar>  TOKEN_FLOAT
39  %token <charrowVar>  TOKEN_CHARACTER
40  %token <charrowVar>  TOKEN_CHARROW
41  %token <charrowVar>  TOKEN_BOOL
42  %token <charrowVar>  TOKEN_VOID
43  %token <charrowVar>  TOKEN_WHETHER
44  %token <charrowVar>  TOKEN_DURING
45  %token <charrowVar>  TOKEN_DO
46  %token <charrowVar>  TOKEN_OTHERWISE
47  %token <charrowVar>  TOKEN_RETURN
48  %token <charrowVar>  TOKEN_HEADFUNC
49  %token <charrowVar>  TOKEN_CLASS
50  %token <charrowVar>  TOKEN_READ
51  %token <charrowVar>  TOKEN_WRITE
52  %token <charrowVar>  TOKEN_WHITESPACES
53  %token <charrowVar>  TOKEN_WORD
54  %token <charrowVar>  TOKEN_AND
55  %token <charrowVar>  TOKEN_OR
56  %token <charrowVar>  TOKEN_NAND
57  %token <charrowVar>  TOKEN_NOR
58  %token <charrowVar>  TOKEN_XOR
59  %token <charrowVar>  TOKEN_XNOR
60  %token <charrowVar>  TOKEN_MODMULDI
61  %token <charrowVar>  TOKEN_ADDSUB
62  %token <charrowVar>  TOKEN_POWER
63  %token <charrowVar>  TOKEN_EQUNOT
64  %token <charrowVar>  TOKEN_GREATLESSEQU
65  %token <charrowVar>  TOKEN_NOT
66  %token <charrowVar>  TOKEN_INCRDECR
67  %token <charrowVar>  TOKEN_LEFTPARENTHESIS
68  %token <charrowVar>  TOKEN_RIGHTPARENTHESIS
69  %token <charrowVar>  TOKEN_SEMICOLON
70  %token <charrowVar>  TOKEN_COMMA
71  %token <charrowVar>  TOKEN_EQUAL
72  %token <charrowVar>  TOKEN_LEFTBRACE
73  %token <charrowVar>  TOKEN_RIGHTBRACE
74  %token <charrowVar>  TOKEN_LEFTBRACKET
75  %token <charrowVar>  TOKEN_RIGHTBRACKET
76  %token <charrowVar>  TOKEN_AMPERSAND
77  %token               TOKEN_EOF 0
```