BitGate Language Manual

Creating a programming language is a difficult and arduous process that will take many years to build. Creating a language consists of six stages which are divided into Front - End and Back - End. The stages of Front - End are four and specifically in the design order are: Lexical Analysis, Syntax Analysis, Semantic Analysis, and Intermediate Code Generation. The stages of Back - End are two and specifically in the design order are: Code Optimization and Target Code Generation.

The language I created, was created with the Flex and Bison tools, and is written with the programming language "C". The Flex tool is a Lexical Analyzer construction tool and its file has a "**.l**" extension. Recognizes tokens from a file and is the first phase of translation. The Bison tool is a Syntax Analyzer and its file has a "**.y**" extension. Implements the Syntax Analysis of the language, accepts as input a file in the form of tokens, checks whether the file is in accordance with the grammar of the language, in case of syntax error informs the user and produces the syntax tree. The implementation of the BitGate language has reached up to Syntax Analysis. This means that have been implemented only Lexical Analysis and Syntax Analysis with the result the creation of two files, the first "**. l**" file belongs to Lexical Analysis, i.e. Flex and the second "**.y**" file belongs to Syntax Analysis . These files communicate with each other and run together by executing the "**makefile**" file with the "**make**" command in the terminal. The code was written with the help of the Visual Code IDE and the help of Cygwin from which we installed Flex, Bison and Make.

File of Lexical Analysis – "lexer.l" – Flex:

In this file, which is in the Lexical Analysis folder, I have matched all the tokens that will be used in my language, which I have declared in the "**lexer.h**" file located in the Lexical Analysis folder. In the "**.l**" file, the tokens are matched with static words and regular expressions. Also, in this file are declared the three functions, "**WriteTokenOnScreen**", "**charrowToOctaBinaryHexa**" and "**yyerror**". "**WriteTokenOnScreen**" is a function that displays on the screen the tokens found when a code file is inserted into it. "**charrowToOctaBinaryHexa**" is a function that takes as input a string (charrow) of a binary, octal or hexadecimal value (0b123 - 0o123 - 0x232FABFE2123) and converts it to a decimal value. The "**yyerror**" function is a function that displays on the screen the data found from an input file that are not tokens. An example of matching tokens with static words and regular expressions in the "**.l**" file and an example of declaring tokens in the "**.h**" file is located below (The examples are taken from the language code):

"**lexer.l**":

```
BINARYCONST          (0b[0-1][0-1]*)|([1-9][0-9]*)

"INTEGER"            { WriteTokenOnScreen(TOKEN_INTEGER); return TOKEN_INTEGER; }
"FLOAT"              { WriteTokenOnScreen(TOKEN_FLOAT); return TOKEN_FLOAT; }
```

```
{BINARYCONST}       { WriteTokenOnScreen(TOKEN_BINARYCONST); yylval.intNum = char
rowToOctaBinaryHexa(yytext); return TOKEN_BINARYCONST; }
```

"**lexer.h**":

```
#define TOKEN_INTEGER             1
#define TOKEN_FLOAT               2
#define TOKEN_BINARYCONST         22
```

File of Syntax Analysis – "parser.y" – Bison:

In this file, which is located in the Syntax Analysis folder, the declaration of the grammar that follow our language is made, of all tokens from the beginning because here it is necessary to declare the types of each token, the order of "reading" the tokens from Bison, and **error-verbose**, which is a Bison function that helps us a lot in finding syntax errors. So, the file "**lexer.h**" is no longer useful to us. Some code examples are listed below (The examples are taken from the language code):

```
%error-verbose
```

Here we declare the error-verbose function, which is a Bison function that helps us a lot in finding syntax errors.

```
%union
{
    int intNum;
    float floatNum;
}
%token <intNum>      TOKEN_INTEGERCONST          "INTEGERCONST"
%token <floatNum>    TOKEN_FLOATCONST            "FLOATCONST"
```

Here we declare the types of tokens, so Bison knows what type each token is.

```
%right               TOKEN_EQUAL
%right               TOKEN_GREATLESSEQU

%left                TOKEN_AND
%left                TOKEN_OR
```

Here we declare which of the tokens are read from left to right and which from right to left. So, we fix the shift / reduce problems and help Bison know exactly what to do and in what order.

```
 { yyerror("Syntax Error in Class Declaration ['Class' Error]"); yyerrok; }
 { yyerror("Syntax Error in Class Declaration [Identifier Error]"); yyerrok; }
 { yyerror("Syntax Error in Class Declaration [Left Bracket Error]"); yyerrok; }
 { yyerror("Syntax Error in Class Declaration [Headfunc Error]"); yyerrok; }
 { yyerror("Syntax Error in Class Declaration [Right Bracket Error]"); yyerrok; }
```

```
%start BitGate

%%

BitGate                         :   TOKEN_CLASS TOKEN_IDENTIFIER TOKEN_LEFTBRACKET Headfunc TOKEN_RIGHTBRACKET
                                    | error TOKEN_IDENTIFIER TOKEN_LEFTBRACKET Headfunc TOKEN_RIGHTBRACKET
                                    | TOKEN_CLASS error TOKEN_LEFTBRACKET Headfunc TOKEN_RIGHTBRACKET
                                    | TOKEN_CLASS TOKEN_IDENTIFIER error Headfunc TOKEN_RIGHTBRACKET
                                    | TOKEN_CLASS TOKEN_IDENTIFIER TOKEN_LEFTBRACKET error TOKEN_RIGHTBRACKET
                                    | TOKEN_CLASS TOKEN_IDENTIFIER TOKEN_LEFTBRACKET Headfunc error
Headfunc                        :   GlobalDeclarations HeadfuncFunction
                                    | error HeadfuncFunction
                                    | GlobalDeclarations error
```

In the two pictures above we see a small part of the grammar that we declare how will be the Class and Headfunc function. In the statement of the above tokens, can occur errors in some position of the tokens. For every possible error that occurs, it has been implemented in the grammar the specific error that will be printed on the screen. Some examples of right and wrong grammar are below.

Correct code without syntax errors:	**Wrong code with syntax errors:**

CLASS _FIRST
{
INTEGER HEADFUNC ()
{
}
}
	CLASS
	{
	INTEGER HEADFUNC (
	{
	}
	}

makefile File – Make:

In this file, which is located in the Syntax Analysis folder, are stated the commands needed to run the language-compiler. That is, the command "**bison -v -d parser.y**" which runs the code located in the file "**parser.y**" and creates the files "**parser.output**", "**parser.tab.c**" and "**parser.tab.h**". The "**flex lexer.l**" command creates the "**lex.yy.c**" file. The command "**gcc parser.tab.c lex.yy.c -lm -o BitGate.exe**" creates the file "**BitGate.exe**". The commands

"**./BitGate.exe Test_Files / test1.bitgate> Test_Files_Outputs / output-test1.txt**",
"**./BitGate.exe Test_Files / test2.bitgate> Test_Files_Outputs / output-test2.txt**",
"**./BitGate.exe Test_Files / test3.bitgate> Test_Files_Outputs / output-test3.txt**",
"**./BitGate.exe Test_Files / test4.bitgate> Test_Files_Outputs / output-test4.txt**",
"**./BitGate.exe Test_Files / test5.bitgate> Test_Files_Outputs / output-test5.txt**",

creates the files "**output-test1.txt**", "**output-test2.txt**", "**output-test3.txt**", "**output-test4.txt**", "**output-test5.txt**" in the folder "**Test_Files_Outputs**" and include the results of the codes we entered as tests. After all the above commands are executed, the command "**rm -f parser.tab.* lex.yy.c BitGate**" and for convenience we run the command "**make clean**", which deletes the files "**parser.tab.c**", "**parser.tab.h**" and "**lex.yy.c**", which is important if we want to recompile the language-compiler because this is how we achieve a clean build. The "**makefile**" is located below:

```
BitGate: Test_Files/test1.bitgate Test_Files/test2.bitgate Test_Files/test3.bitga
te Test_Files/test4.bitgate Test_Files/test5.bitgate lexer.l parser.y
    bison -v -d parser.y
    flex lexer.l
    gcc parser.tab.c lex.yy.c -lm -o BitGate.exe
    ./BitGate.exe Test_Files/test1.bitgate > Test_Files_Outputs/output-test1.txt
    ./BitGate.exe Test_Files/test2.bitgate > Test_Files_Outputs/output-test2.txt
    ./BitGate.exe Test_Files/test3.bitgate > Test_Files_Outputs/output-test3.txt
    ./BitGate.exe Test_Files/test4.bitgate > Test_Files_Outputs/output-test4.txt
    ./BitGate.exe Test_Files/test5.bitgate > Test_Files_Outputs/output-test5.txt

clean:
    rm -f parser.tab.* lex.yy.c BitGate
```

How to run:

To run the language and the compiler, we must first install Visual Code and Cygwin, from which we will install **flex**, **bison**, **make**, **gcc-core** (GNU Compiler Collection (C, OpenMP)), **gcc-g++** (GNU Compiler Collection (C++)) and **gdb** (The GNU Debugger). We also integrate the Cygwin bash into the Visual Code JSON. Then copy the "**lexer.l**" file from the "**Lexical_Analysis**" folder, the "**parser.y**" file from the "**Syntax_Analysis**" folder, and the "**makefile**" file from the "**Syntax_Analysis**" folder and paste all three files in the root folder "**Language Design And Implementation**". Also, from the root folder "**Language Design And Implementation**" we must delete "**BitGate.exe**". Finally, to run the language and the compiler, in the terminal of Visual Studio we run the "**make**" command.