**Preston Powell**

**Extra Credit Specification Document**

**Compiler Class**

The **Compiler** class handles the conversion of all input files into string objects to be analyzed. These strings are passed first to an instance of the **Lexer** class, and then to an instance of the **Parser** class, for lexical and syntactical analysis respectively.

**Lexer Class:**

The **Lexer** class is a lexical analyzer that takes an input string and evaluates it one character at a time, checking for different types of characters and the patterns they create to classify strings of characters into lexical *token* objects.

*Tokens* have values to represent *type*, token *code,* and their literal *values*. These token objects are appended to an array and output into the terminal upon running the **Lexer** class’s run function.

**Parser Class:**

The **Parser** class is a syntactical analyzer that takes an array of *token* values as input and determines if they are acceptable for the rules of the language in their current positions. It will return “BASED CODE” in the terminal if the input is based (acceptable)

**LITERAL VALUES:**

real\_literal (0):

Real\_literals include all whole numbers with digits between 0 and 9 inclusive.

natural\_literal (1):

Natural\_literals include all numbers with digits [0-9] separated by a “/” character.

bool\_literal (2):

Boolean\_literals include values of “nocap” representing an open logic gate, and “cap” representing a closed logic gate.

char\_literal (3):

Char\_literals include any single ASCII character or escape sequence enclosed in single quotation marks.

string\_literal (4):

String\_literals include any combination of ASCII characters or escape sequences enclosed in double quotation marks.

**DATA TYPES:**

word (40):

Words are strings of characters.

char (41):

Chars are single characters.

natural (42):

Naturals are whole numbers.

real (43):

Reals are fractional numbers.

bool (44):

Bools are truth values of either type “nocap” (true) or “cap” (false).

**VARIABLE RULES:**

Variables can be declared in multiple ways. Any variable can be created in one code statement and initialized in a subsequent code statement. Additionally, a keyword specifying the data type of the variable can be added before the variable identifier but is not necessary. All variables are *token* objects that are expressed by string representations of their alphabetical, numeric, or alphanumeric values. Variables and variable operations can be given precedence by being placed inside of parentheses. All variables can be acted upon by addition, subtraction, multiplication, division, modulus, and exponentiation operators.

**CONTROL STRUCTURE RULES:**

**SEQUENCE:** Code statements can be listed in concurrence and operated on in order of which statement is made first in the document. Each statement must be followed by a semicolon to signify the end of the statement.

**SELECTION:** Selection statements are identified by the two keywords *try* acting as an “if” declaration, and *catch* acting as an “else” declaration.

A *try* keyword must be followed by a set of parentheses containing a *bool* expression that would evaluate to either *nocap* or *cap*.

A *catch* keyword does not require a *bool* expression to run, as whether it runs or not is dependent on whether the *try* statement runs. If the *try* runs, *catch* doesn’t and vice versa.

A code block, beginning with a left curly bracket and ending with a right one, must follow both the *try* and *catch* declarations. It will contain any code to be evaluates should either statement run.

**LOOPS:** Loops are identified by the *loop* keyword followed by a *bool* expression in parentheses, and a code block containing the code to be run should the bool expression evaluate to *nocap*.

**FUNCTION CALLS:** Function calls are identifiers that are followed by parentheses. These parentheses contain a list of parameter declarations, each beginning with a *$* to signify its nature. Constructors must be followed by a code block with the function’s instructions, calls of established functions do not require this.

**KEYWORDS & SPECIAL SYMBOLS:**

try, catch: select statement keywords

loop: loop keyword

word: string keyword

char: character keyword

num: natural keyword

frac: real keyword

bool: Boolean keyword

nocap: “true”

cap: “false”

START: beginning of program

STOP: end of program

#: comment symbol

+: addition operator

-: subtraction/unary negation operator

\*: multiplication operator

/: division operator

^: exponentiation operator

(, ): precedence/function operators

>, >o=, <, <o=, !==: comparative operators

&: logical and

?: logical or

!: logical not

;: statement end

,: parameter separator

{, }: code block constraints

$: parameter identifier

**SEMANTICS:**

The language’s semantics have not been defined yet. Check back for future updates.

**HOW TO WRITE A PROGRAM:**

To begin writing a program, just create a simple text file.

Type “START” in all caps. This signifies the beginning of the program and must be included to produce valid code.

Typing “END” in all caps is how you end a program. Place this at the end of your code to terminate the program.

You can now declare variables or functions, create mathematic or Boolean expressions, and comment on code by placing a *#* at the beginning of any line or position in a line at which you wish for your input not to be read.

All function names must include letters/underscores, contain a comma separated list of zero or more parameters in parentheses, and have their code encased in a code block beginning at the next *{*, and ending at the next *}*.

Word literals must be placed inside double quotations marks, char literals must be placed inside single quotation marks, and both can contain either ASCCI value(s) or escape sequence(s).

Each code statement must end with a semicolon. Multiple statements in sequence are allowed within either a code block, or just between the program’s START and END markers.

Code blocks and functions can be nested and are not required to contain any code.

After typing your code, pass your file’s path into the Compiler. Upon running the program, you should either receive a detailed lexical error statement, or a significantly less detailed syntactical error statement. However, if your code is valid, you will receive a message in terminal that reads “BASED CODE”.