

**RULES BOOK FOR BRAVO COMPILER**

**PROJECT PARTICIPANTS:**

***Syed Samama Ali B19103062***

***Sufiyan Mohammad Salman B20103065***

***Abu Ubaida B20103007***

***Muhammad Ibrahim khan B20103041***

***Muhammad Shehroz B20103052***

***Faraz Javed B20103022***

***Muhammad Mairaj B20103043***

***Mohammad Zoren Khan B20103033***



**Preface:**

The Bravo Programming Language is a modern, high-level language designed to be easy to learn, write, and read. It is a compiled language that is highly efficient and runs on a variety of platforms. The language has a simple syntax and a small set of rules, making it ideal for both novice and experienced programmers.

The Bravo Compiler is a tool used to translate source code written in the Bravo Programming Language into machine code that can be executed by a computer. It is a highly optimized compiler that produces fast and efficient code. The compiler has been designed to ensure that programs written in Bravo are correct, efficient, and secure.

This document outlines the language specification for the Bravo Programming Language, including the keywords, datatypes, operators, and other components that make up the language. It also includes the regular expressions that define the rules for various language components, such as identifiers, constants, and keywords.

We hope that this specification provides a clear and comprehensive guide for developers who wish to use the Bravo Programming Language and its accompanying compiler to build powerful and efficient applications.

**RULES FOR BRAVO COMPILER**

**Control Structures:**

1. **KEYWORDS:**

**CONDITIONAL STATEMENTS:**

* **if:** executes a code block conditionally based on a true/false condition.
* **elseif:** allows checking additional conditions after an "if" statement, and executing a different code block if those conditions are met.
* **else:** executes a code block when the condition in an "if" or "elseif" statement is false.

**LOOPS:**

* **while:** starts a while loop that executes a code block as long as the loop condition is true.
* **for:** starts a for loop that allows iterating over a range of values or elements in a collection.
* **for-each** : A for each loop (also known as a for-in loop) is a loop that iterates over a collection of elements, such as an array or list. It allows you to execute a set of statements for each element in the collection.

**DATA STRUCTURE:**

* **Array:** Array is a data structure that allows you to store a collection of elements of the same data type in contiguous memory locations. The elements in the array are accessed using an index, which represents the position of the element within the array.

The structure of an array consists of several components. These include:

1. The **data type** of the elements in the array, which determines the size and layout of each element in memory.
2. The **size of the array**, which specifies the number of elements that can be stored in the array.
3. The **index** of each element, which represents its position within the array.
4. The **memory address** of the first element in the array, which is used to access the array in memory.

Arrays can be one-dimensional or multi-dimensional, and can store data types such as integers, floating-point numbers, characters. They can be dynamically allocated and deallocated using the appropriate operators, and can be passed as parameters to functions and returned as values from functions.

**LOOP CONTROL:**

* **continue:** skips the rest of the current iteration in a loop and moves to the next one.
* **break:** terminates a loop prematurely and moves to the next statement outside the loop.

**FUNCTIONS:**

* **return:** allows a function to return a value to the caller and terminate its execution.
* **void:** used to indicate that a function does not return a value.
* **function/def:** used to define a function or a method that performs a specific task or returns a value.

**OBJECT-ORIENTED PROGRAMMING:**

* **class:** used to define a new class, which is a blueprint for creating objects that share common attributes and behaviors.
* **interface:** allows defining a contract that specifies a set of methods that a class must implement to comply with the interface.
* **final:** allows declaring a variable or a method as final, which means they cannot be changed or overridden, respectively.
* **static:** allows declaring a variable or a method as static, which means they belong to the class rather than an instance of the class.
* **public:** an access modifier that allows a member to be accessed from any class or package.
* **private:** an access modifier that allows a member to be accessed only from within the same class.
* **protected:** an access modifier that allows a member to be accessed from within the same class or its subclasses.
* **extends:** used to indicate that a class inherits attributes and behaviors from a superclass.
* **implements:** used to indicate that a class implements one or more interfaces, and provides the required implementation for their methods.

**EXCEPTION HANDLING:**

* **try:** starts a try block that contains code that may throw an exception.
* **catch:** starts a catch block that catches and handles exceptions thrown in the corresponding try block.
* **finally:** starts a finally block that contains code that is always executed, whether or not an exception is thrown in the try block.
* **throw:** allows throwing an exception manually from a code block.

**OTHER:**

* **self/this:** used to refer to the current object or instance in a class method or constructor.
* **parent/super:** used to refer to the superclass of the current class or to call its methods from within a subclass.

1. **COMMENTS**

* Start of a single line comment:
  + Use the dollar sign symbol (**//**) followed by a space to indicate a single-line comment. Any text that follows the dollar sign on the same line will be ignored by the compiler or interpreter.
* Start and ending of a multiline comment:
  + Use the sequence **/\*** to start a multiline comment, and use **\*/** to end the comment. Any text between these markers will be ignored by the compiler or interpreter, even if it spans multiple lines.

1. **DATA TYPES**

It is important to have a solid understanding of the data types that the compiler will be working with. Here are some common data types that our compiler might encounter:

* **int (integer):** This data type represents whole numbers without a fractional component. It is important for the compiler to know how to handle arithmetic operations on integers, such as addition, subtraction, multiplication, and division.
* **float (floating-point number):** This data type represents numbers with a fractional component, and is often used to represent real-world values such as measurements or scientific data. The compiler needs to be able to handle floating-point arithmetic operations, and to be aware of the limitations of floating-point precision.
* **char (character):** This data type represents a single character, such as a letter, digit, or punctuation mark. The compiler needs to be able to recognize and handle characters in order to parse and analyze source code.
* **String (string):** This data type represents a sequence of characters, and is often used to represent text or other data that can be interpreted as text. The compiler needs to be able to recognize and manipulate strings in order to parse and analyze source code.
* **bool (Boolean):** This data type represents a binary value that can be either true or false. It is often used in conditional statements and logical operations. The compiler needs to be able to recognize and handle Boolean expressions in order to generate correct code.

By understanding and working with these data types, a compiler can parse and analyze source code, generate correct machine code, and provide useful error messages to the programmer.

1. **PUNCTUATORS:**

Punctuators are characters used in programming languages to separate, group, and access various elements of the code. Here are some common punctuators that our compiler might encounter:

* **; (Semicolon):** This punctuator is used to terminate lines of code.
* **, (Comma):** This punctuator is used as a separator in lists or arguments. For example, in a function call, the arguments might be separated by commas.
* **. (Dot):** This punctuator is used to access attributes and methods of objects.
* **() (Parenthesis):** This punctuator is used to declare functions and to group expressions. In function declarations, the parameters are usually enclosed in parentheses.
* **{} (Braces):** These punctuators are used to define the scope of methods and to group statements into blocks. In our programming languages, the body of a function or a conditional statement is enclosed in braces.
* **[] (Brackets):** These punctuators are used to access elements in arrays and other data structures.
* **// : (Colon):** This punctuator is used to define labels and to separate the parts of a ternary expression.

By recognizing and correctly interpreting these punctuators, a compiler can understand the structure of the code and generate correct machine code.

1. **OPERATORS**

Operators are symbols or keywords used in programming languages to perform operations on data. Here are some common operators that our compiler might encounter:

**a. Arithmetic Operators**

Arithmetic operators are used to perform mathematical operations on numeric data types. These include:

* **+** (addition)
* **-** (subtraction)
* **\*** (multiplication)
* **/** (division)
* **%** (modulus or remainder)

**b. Relational Operators**

Relational operators are used to compare values and return a Boolean result (true or false). These include:

* **<** (less than)
* **<=** (less than or equal to)
* **>** (greater than)
* **>=** (greater than or equal to)
* **!=** (not equal to)
* **==** (equal to)

1. **IDENTIFIERS**

**Identifiers** in our programming language are names given to variables, functions, and other user-defined elements. Here are some rules and regular expressions (regex) for identifying valid identifiers in our compiler:

a. Name should start with an alphabet

Identifiers must begin with an alphabet (either uppercase or lowercase).

b. Followed by any alphanumeric character

After the first character, identifiers can contain any alphanumeric character (uppercase or lowercase letters, digits 0-9, and the underscore **\_**).

1. **Constants**

Constants are fixed values that do not change during the execution of a program. Here are the regex patterns for identifying different types of constants in our compiler:

* + **Integer:**

Integers are whole numbers without any fractional component.

* **Float:**

Floats are numbers with a decimal component.

* **Boolean:**

Boolean constants are either true or false. These are represented using the keywords **true** and **false**.

* **Character:**

A character constant is a single character enclosed within single quotes.

* **String:**

A string constant is a sequence of characters enclosed within double quotes.