**Programming Language Proposal Documentation**

**BOOP: Beginners’ Object-Oriented Programming**

**BSCS 3-3**

**Batisla-Ong, John Franky**

**Buenconsejo, Gian Paolo**

**De Villa, Edrick**

**Dela Cruz, John Paulo**

**Lopez, Toni Gaile**

January 2024

**Features**

* High-level
* Object-oriented
* Time-oriented
* Statically typed
* Type safe
* Event driven
* Math friendly (can read and use math symbols)
* Integrates basic math functions
* Beginner Friendly OOP

**Inspirations**

* The user-friendly syntax of C#.
* The strive for learning, growth, and development.
* The concept of reliability in the language design to reduce the likelihood of bugs and errors naturally makes the development process smoother and more enjoyable.
* R programming math symbols.

**Target users**

* Users starting at object-oriented language
* Users starting to learn programming
* Mathematicians entering the field of programming
* Users who have existing mathematical knowledge/skills

**New principles**

* Time-oriented
* Math-friendly

**TABLE OF CONTENTS**

1. **Introduction**  5
2. **Syntactic Elements of the Language**  8
   * 1. **Character Sets**  8
     2. **Identifiers**  9
     3. **Operation Symbols**  10
     4. **Arithmetic**  10

**Arithmetic**  10

**Assignment** 10

**Boolean**  11

**Unary**  13

**Mathematical**  14

* + 1. **Keywords and Reserved Words**  15
    2. **Noise Words**  25
    3. **Comments** 27
    4. **Whitespaces**  27
    5. **Delimiters** 27
    6. **Free and Fixed-Field Formats** 29
    7. **Expressions** 29
    8. **Statements**  31

**Declaration statements**  31

**Collections**  32

**Methods**  34

**Static variables** 36

**Input/Output statements** 36

**Assignment statements** 38

**Conditional statements**  38

**Iterative statements**  41

**Time variables declaration** 44

1. **Time-oriented Principle** 46

**Fortime statement**  46

**Repeating functions**  46

**Delayed functions**  47

**Repeating and Delayed functions** 48

**Async functions** 49

**Async, Repeating, and Delayed functions** 50

**Time-based events** 51

1. **Math-friendly Principle** 53

**Math symbols**  53

**Math Functions**  55

**Sample Functions that use Math Symbols** 57

**BOOP Language Server** 58

1. **Syntax** 61
2. **Introduction**

Beginners’ Object-Oriented Programming Language (BOOP) is a high-level, object-oriented, time-oriented, statically typed, type safe, event driven, math friendly, and beginner friendly programming language. The language is designed for users who are just starting to learn programming. BOOP is primarily focused on Object-Oriented Programming (OOP). The goal of this language is to provide an uncomplicated learning experience for novice programmers so that they will be able to learn and grasp the essence of programming as well as to fuel their ever-burning passion for development.

Regarding the programming language’s name, we used the acronym BOOP for several reasons. Since naming a programming language is a branding itself, we must choose a name that clearly communicates the capabilities of our programming language. Therefore, aside from its abbreviated meaning, the onomatopoeic word “BOOP” elicits a gentle and light-hearted impression on humans. Since the word does not sound daunting, grave, or serious, programmers will immediately have the impression that it is an easy or a simple programming language. This perception aligns with the purpose of our programming language, an OOP directed for beginners. Furthermore, the word “BOOP” can generate curiosity since it is a playful and cute name for a programming language.

One of BOOP’s inspirations is the user-friendly syntax of C#. By implementing this in our programming language, it can help the user to easily understand the language as well as enable the user to learn OOP with ease. Another inspiration for the creation of BOOP is the math symbols and math functions of R programming. The ability of R programming to use math symbols and functions allows users to express mathematical concepts and perform them directly in code. This feature in R inspired us to adapt it in BOOP since math symbols offer an advantage of enhanced expressiveness, clarity, and efficiency when performing mathematical expressions in the code. This added feature will also further enhance the user’s learning process since they can simply use their existing mathematical knowledge to translate into the code. Additionally, allowing math symbols can reduce the user’s mental workload as it provides a familiar and less intimidating method to express mathematical concepts.

As for the target users, we have selected users who are still relatively new in programming or object-oriented programming since we have also experienced the challenges of being new to programming. When you’re a beginner, there are tons of programming languages out there resulting in the difficulty of choosing a specific language to learn. Furthermore, the programming languages being recommended to beginners are mostly traditional. Traditional programming languages can have steep learning curves due to their complex syntax and advanced features. Thus, making it difficult for beginners to get started with coding. Moreover, in a traditional programming language, beginners will be more likely to encounter hard-to-debug issues. Since we are making a beginner-friendly object-oriented programming language, safety in language design and user-friendliness are some of our utmost priorities in our programming language. Taking these factors into account will make the target users reduce the likelihood of encountering hard-to-debug errors as well as lower the barriers to entry into the programming world. Additionally, since our programming language is also math-friendly, mathematicians as well as users who have existing mathematical knowledge or skills are the target audience of BOOP too.

Given that BOOP is all about object-oriented programming, the language encourages the usage of its principles, namely: abstraction, encapsulation, inheritance, and polymorphism. Classes are limited to the inheritance of one parent class only to avoid unambiguous cases. With this in mind, the language makes use of interfaces to define contracts. This ensures a clearer and more readable code.

BOOP will not be BOOP if it does not have any distinguishing principles that will set it apart from any other existing programming languages. We have added these important principles namely: (1) list and dict are now keywords, (2) math symbols can be used and read together with some basic mathematical functions, (3) time is a data type and can be represented as variables, (4) events that functions can subscribe to, and (4) the revolutionary utilization of modules. For the first principle, list keyword is added since it can be scaled up after declaration, elements can be accessed using indexes, and it has useful methods such as Add() and Remove(). Meanwhile, dict keyword is also included, since its ability to have fast lookup times and enforce unique key-to-value mapping is a noteworthy feature in a programming language.

One of the trademarks of the BOOP language is the integration of mathematical functions and symbols. This unique feature gives programmers a strong and expressive toolkit for solving various mathematical problems. Because of the language's dedication to accuracy, precision, and adaptability, novices may utilize its capabilities for a wide range of applications.

The incorporation of these mathematical symbols and functions into BOOP language allows developers to build code that closely mirrors mathematical expressions, improving both clarity and performance. The benefits stem from not just the simplification of difficult computations and expressions, but also from the promotion of a more intuitive coding experience. However, these characteristics must be used with caution, considering potential drawbacks such as greater complexity for people inexperienced with sophisticated mathematical ideas—although the integration of the different functions and symbols are just on the basic level. The BOOP language may stand as a strong tool for a wide range of applications, from scientific computing to data analysis and beyond, by coherently combining various mathematical features.

In departure from conventional programming languages, BOOP introduces Time as a first-class data type that can be represented as variables namely the second and millisecond types, which derive from the base class of time. These time variables serve to represent time in code and can be used as arguments to run BOOP statements that rely on time flow when executed.

Additionally, BOOP introduces a new type of function, called the “Repeating” function defined with the ‘repeat’ keyword. This function basically serves as a loop that executes for a given amount of time or iterations. Repeating functions can break and continue.

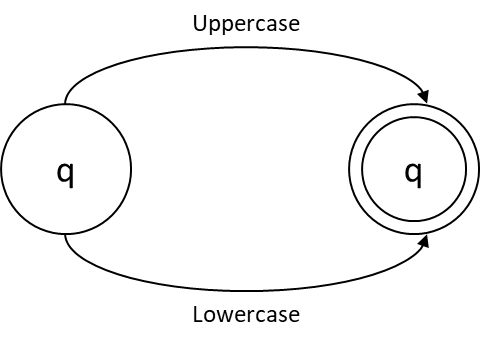
What’s more is that BOOP also makes use of events that functions can subscribe to. BOOP encourages the use of events for maintainability and scalability.

**II. Syntactic Elements Of Language**

**1. Character Sets**

|  |  |
| --- | --- |
| **Uppercase** | { A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z } |
| **Lowercase** | { a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z } |
| **Alphabet** | { **Uppercase**, **Lowercase** } |
| **Digits** | { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } |
| **Alphanumeric** | { **Alphabet**, **Digits** } |
| **Special\_Characters** | { , `, ~, @, !, $, #, ^, \*, %, &, (, ), [, ], {, }, <, >, +, =, \_, -, |, /, \, ;, :, ‘, “, ,, ., ? } |
| **Math\_Symbols** | { √, **x̅**, ∏, Σ, π, θ, ι } |
| **Symbols** | { **Special\_Characters**, **Math\_Symbols** } |
| **Character** | { **Alphanumeric**, **Symbols** } |

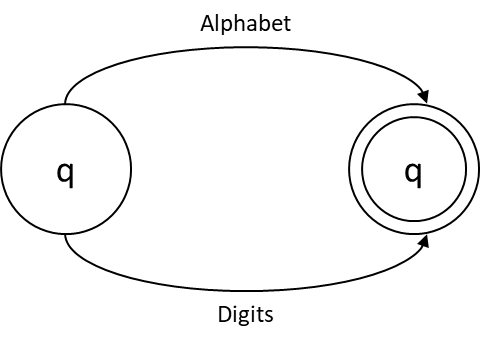
**Machine for Alphabet**

****

**Regular expression**

Uppercase + Lowercase

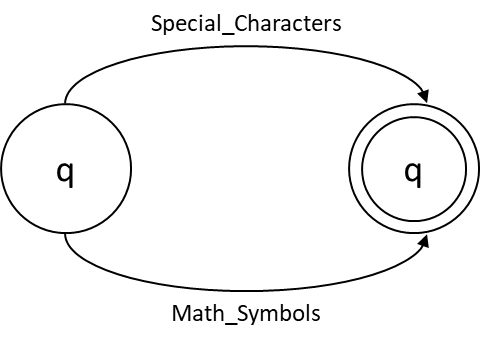
**Machine for Alphanumeric**

****

**Regular expression**

Alphabet + Digits

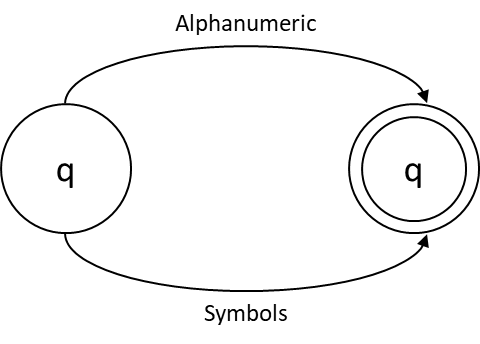
**Machine for Symbols**

****

**Regular expression**

Special\_Characters + Math\_Symbols

**Machine for Character**

****

**Regular expression**

Alphanumeric + Symbols

**2. IDENTIFIERS**

**Naming conventions**

* module, class, struct, interface, enum, func, repeat, event, const, bool, and public members should be in Pascal case. (e.g. PascalCase)
* var, local variables, method parameters, and private members should be in the camel case. (e.g. camelCase)
* Interface names should end in “-able” or be prefixed with a capital ’I’ if it looks stupid with the former. (e.g. Printable, and IDeath because there is no such thing as Deathable, however, Killable would be accepted)

|  |
| --- |
| **Code**  public class ExampleClass (Printable)  {  public int SomeNumber;  private string hiddenMessage;  public func ToString () -> string  {  return hiddenMessage;  }  public func SetMessage(string message)  {  var exampleLocalVar = 5;  hiddenMessage = message;  }  } |

**Machine for Identifiers**

**A black background with a black square

Description automatically generated with medium confidence**

**Regular expression**

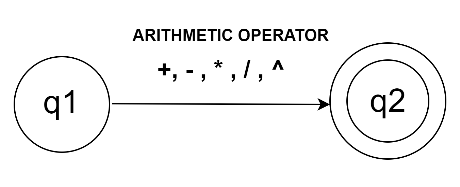
(Alphabet + \_)(Alphanumeric + \_)\*

**3. OPERATION SYMBOLS**

**3.1**. **Arithmetic**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Operator** | **Example expression** | **Description** |
| Addition | + | a + n | Adds the value of a and b. |
| Subtraction | - | a - n | Subtracts the value of a and b. |
| Multiplication | \* | a \* n | Multiplies the value of a and b. |
| Division | / | a / n | Divides the value of a and b. |
| Modulo | % | a % n | Divides the value of a and b, but it returns the remainder. |
| Exponent | ^ or \*\* | a ^ n or a \*\* n | Computes the base power a to the exponent n. |

**Machine for Operation Symbols**



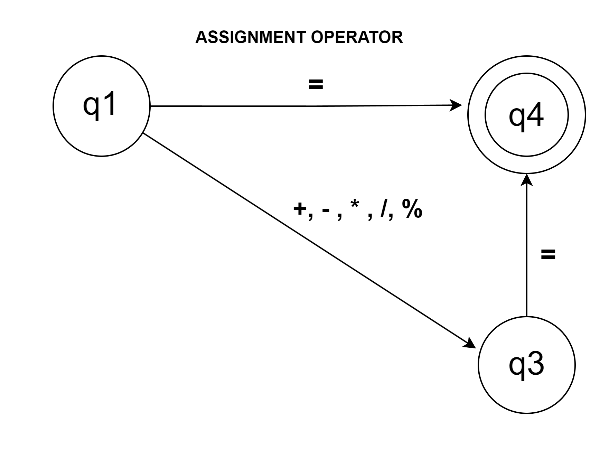
**Regular expression**

((+) + (–) + (\* + \*\*) + (/) + (%) + (^))

**3.2** **Assignment operators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Operator** | **Example expression** | **Description** |
| Assignment | = | a = 1 | Assigns the value of a to 1. |
| Addition Assignment | += | a += 1 | Adds 1 to the current value of a and returns the sum. |
| Subtraction Assignment | -= | a -= 1 | Subtracts 1 from the current value of a and returns the difference. |
| Multiplication Assignment | \*= | a \*= 1 | Multiplies 1 to the current value of a and returns the product. |
| Division Assignment | /= | a /= 1 | Divides 1 to the current value of a and returns the quotient. |
| Modulo Assignment | %= | a %= 1 | Divides 1 to the current value of a and returns the remainder. |

**Machine for Assignment Operators**



**Regular expression**

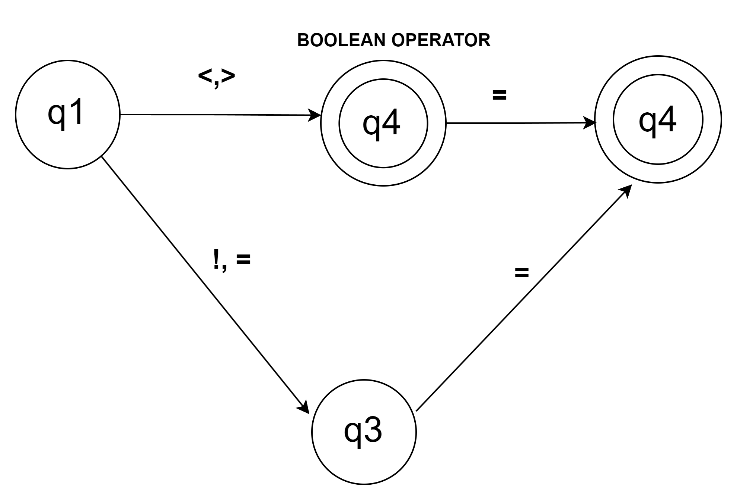
(=) + ((+) + (-) + (\*) + (/) + (%))=

**3.3** **Boolean operations**

**Relational**

|  |  |  |
| --- | --- | --- |
| **Relational operator** | **Example expression** | **Description** |
| Is equal to (==) | a == b | Equates to true if a is equal to b, and false if they are not equal |
| Is not equal to (!=) | a != b | Equates to true if a is not equal to b, and false if they are equal. |
| Is greater than (>) | a > b | Equates to true if a is greater than b, and false if a is less than or equal to b. |
| Is less than (<) | a < b | Equates to true if a is less than b, and false if a is greater than or equal to b. |
| Is greater than or equal to(>=) | a >= b | Equates to true if a is greater than or equal to b, and false if a is less than b. |
| Is less than or equal to (<=) | a <= b | Equates to true if a is less than or equal to b, and false if a is greater than b. |

**Machine for Boolean Operations**



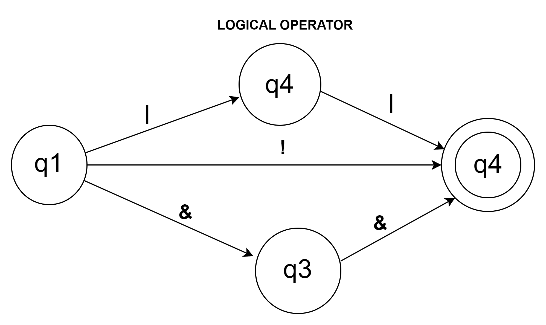
**Regular expression**

((< + >) + (< + >) = ) + (! + =) =

**Logical**

|  |  |  |
| --- | --- | --- |
| **Logical Operators** | **Example Expression** | **Description** |
| NOT (!) | !a | Equates to true if a is false and false if a is true. |
| AND (&&) | a && b | Equates to true if both a and b are true; otherwise, it's false. |
| OR (||) | a || b | Equates to true if either a or b or both are true |

**Machine for Logical Operators**



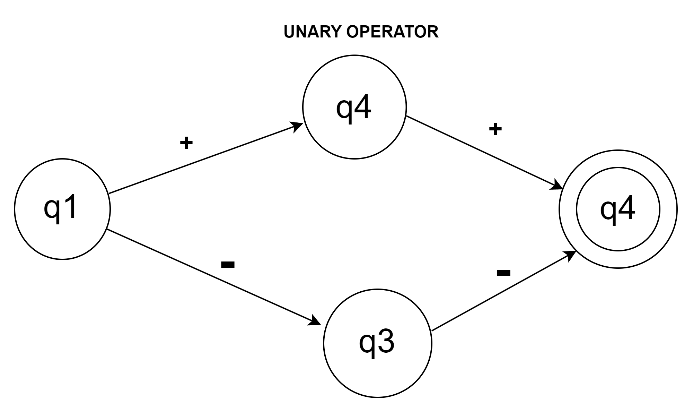
**Regular Expression**

|| + && + !

**3.4 Unary Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Example expression** | **Description** |
| Unary plus **(**+**)** | +a | a | Indicates that *a* is a positive value. |
| Unary minus **(**-**)** | -a | Indicates that *a* is a negative value. |
| Increment **(**++**)** | ++a | a++ | Increases the value of operand *a* by 1. |
| Decrement **(**--**)** | --a | a-- | Decreases the value of operand *a* by 1. |
| Imaginary | aι | Indicates that *a* is an imaginary number. |

**Machine for Unary Operators (excluding** ι)**:**



**Regular expression**

((+) + (++)) + ((-) + (--))

**3.5** **Mathematical**

**Symbols**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Example Expression** | **Description** |
| Square root **(**√**)** | √a | Calculates the square root of a. |
| Mean **(**x̄**)** | x̄ (a, b, c, d)  x̄ (array) | Represents the mean/average of data a, b, c, and d.  Represents the mean/average of data in the array. |
| Product notation **(**∏**)** | ∏(a, b) | Represents the product of numbers from a to b |
| Summation notation **(**Σ**)** | Σ(a, b) | Represents the sum of numbers from a to b |
| Pi **(**π**)** | 2πr  πr² | The circumference of a circle with radius 'r' is given by 2πr.  The area of a circle with radius 'r' is given by πr². |
| Theta **(**θ**)** | sin(θ) | In trigonometry, sin(θ) represents the sine of an angle θ |
| Imaginary unit ***(***ι***)*** | 2ι *5*ι *+ 20* | Represents the imaginary unit; used for denoting an imaginary number. |
| Absolute Value **(**||**)** | |a| | Returns the positive value of a. |

**Functions**

|  |  |  |
| --- | --- | --- |
| Sine (**Sin(θ)**)  Inverse Sine (**Asin(θ)**) | Sin(30)  Asin(0.5) | Returns the sine value of 30 degrees.  Returns the angle whose sine is 0.5, in radians. |
| Cosine (**Cos(θ)**)  Inverse Cosine (**Acos(θ)**) | Cos(45)  Acos(0.5) | Returns the cosine value of 45 degrees.  Returns the angle whose cosine is 0.5, in radians. |
| Tangent (**Tan(θ)**)  Inverse Tangent (**Atan(θ)**) | Tan(60)  Atan(1) | Returns the tangent value of 60 degrees.  Returns the angle whose tangent is 1 in radians. |
| Secant (**Sec(θ)**)  Inverse Secant (**Asec(θ)**) | Sec(30)  Asec(2) | Returns the secant value of 30 degrees.  Returns the angle whose secant is 2 in radians. |
| Cosecant (**Csc(θ)**)  Inverse Cosecant (**Acsc(θ)**) | Csc(45)  Acsc(1.5) | Returns the cosecant value of 45 degrees.  Returns the angle whose cosecant is 1.5 in radians. |
| Cotangent (**Cot(θ)**)  Inverse Cotangent **(Acot(θ)**) | Cot(30)  Acot(0.5) | Returns the cotangent value of 90 degrees.  Returns the angle whose cotangent is 0.5 in radians. |
| Greatest Common Divisor  **GCD()** | GCD(a,b,c)  GCD(a[]) | Returns the greatest common divisor of a, b, and c.  Returns the greatest common divisor of the elements of array a |
| Least Common Multiple  **LCM()** | LCM(a,b,c)  LCM(a[]) | Returns the least common multiple of a, b, and c.  Returns the least common multiple of the elements of array a |
| Ceiling  **Ceil(x)** | Ceil(-2.5) Ceil(2.4) | Rounds up and returns the smallest integer greater than or equal to a given number. |
| Floor  **Floor(*x*)** | Floor(-11.69)  Floor(7.77) | Rounds down and returns the largest integer less than or equal to a given number. |
| Logarithm  **Log(a)**  **Log(base, a)** | Log (a)  Log (base,a) | The common logarithm of a.  The logarithm of a with a specified base. |
| Permutation  **Perm(n, r)** | Perm(5,3) | Represents the number of ways to arrange 3 items out of 5 in different orders. |
| Combination  **Comb(n, r)** | Comb(5,3) | Represents the number of ways to arrange 3 items out of 5 without considering the order. |

**4. KEYWORDS AND RESERVED WORDS**

**Keywords**

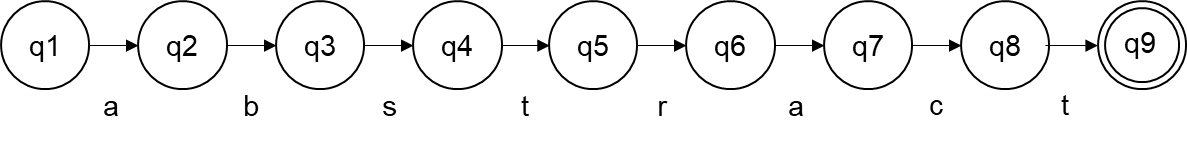
|  |  |
| --- | --- |
| **Keyword** | **Definition** |
| abstract | Represents a member that needs to be implemented by a derived class. |
| async | Defines an asynchronous function. |
| await | Waits for a task to complete. |
| base | Used in a derived class to access the parent class members. |
| bool | A boolean value. |
| break | Terminates the closest iteration statement. |
| case | Used to specify different code blocks to execute based on the value of an expression. |
| catch | Used for blocks of code to be executed when a specific exception is thrown. |
| char | A single character. |
| class | An object constructor for creating objects. |
| const | An immutable variable. |
| continue | Used to skip the current iteration in an iteration statement. |
| default | An expression that produces the [default value](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/default-values) of a type. |
| delayed | Defines a delayed function. |
| dict | Represents a collection of keys and values. |
| do | Execute statement/s before checking the condition for continuing loop iteration. |
| double | A 64-bit signed decimal number. |
| else | Executes the statement/s if the condition is false. |
| enum | A special class that represents a group of constants. |
| event | Used to declare an event. |
| false | Represents the logical false state. |
| float | A 32-bit signed decimal number. |
| for | Indicates a block of code to be executed a specified number of times. |
| foreach | Used to iterate over elements in a collection. |
| format | Used to format strings. |
| fortime | Used to execute a loop for a certain amount of time. |
| from | Used to define imported members’ module location. |
| func | Used to define functions/methods. |
| get | Used to define a get accessor. |
| if | Executes the statement/s if the condition is true. |
| inherit | Used to indicate inheritance. Can be used in place of a colon. |
| int | A 32-bit signed integer. |
| interface | An abstract class, which can only contain abstract methods and properties. |
| list | Represents a collection of a data type. |
| millisec | A time variable used to represent milliseconds. |
| module | Defines a module. |
| null | Represents the absence of a value or reference. |
| override | A new implementation of the method inherited from a base class. |
| private | An access modifier that is accessible only within the body of the class or the struct in which they are declared. |
| protected | An access modifier that is accessible only within its class and by derived class instances. |
| public | An access modifier that has no restrictions on its accessibility. |
| repeat | Defines a repeating function. |
| ref | Used to reference types. |
| return | Used to exit a function on which it may/may not return a value. |
| second | A variable to represent time in seconds. |
| set | Used to define a set accessor. |
| static | Used to declare class-level variables and methods. |
| string | An array of characters. |
| struct | A value type that can encapsulate data and related functionality. |
| switch | Elects a statement list to execute based on a pattern match with an expression. |
| this | Refers to the current instance of a class or struct. |
| throw | Used to raise an exception in code. |
| time | Base class for time variables. |
| true | Represents the logical true state. |
| try | Used to indicate a block of code where exceptions may be thrown. |
| use | Used for the inclusion of external libraries’ functionalities. |
| virtual | Represents a member that can be overridden by a derived class. |
| void | A return type indicating that a function does not return a value. |
| where | Used to constraint types for arguments in generic types. |
| while | Condition for continuing loop iteration. |

**Reserved words**

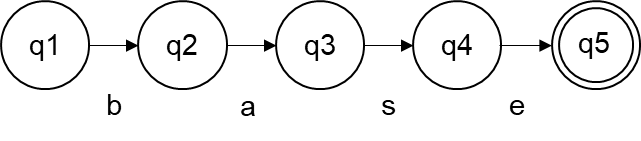
All keywords in BOOP are reserved words and cannot be used as identifiers.

**Machine for Keywords**

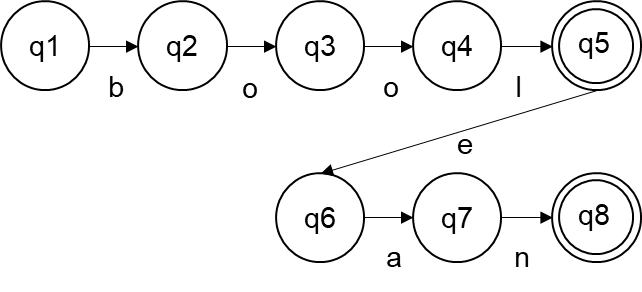
abstract



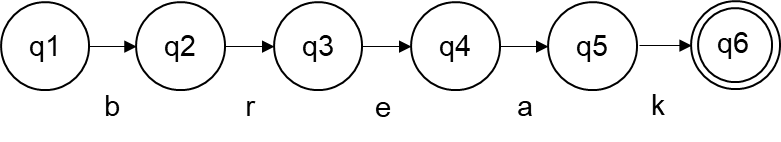
base



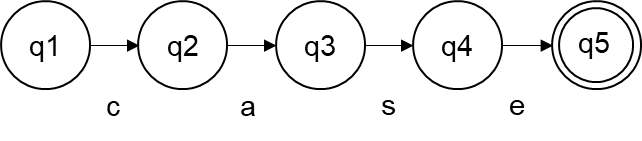
bool



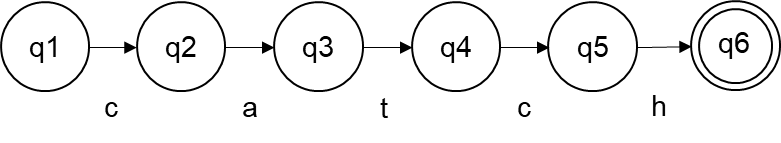
break



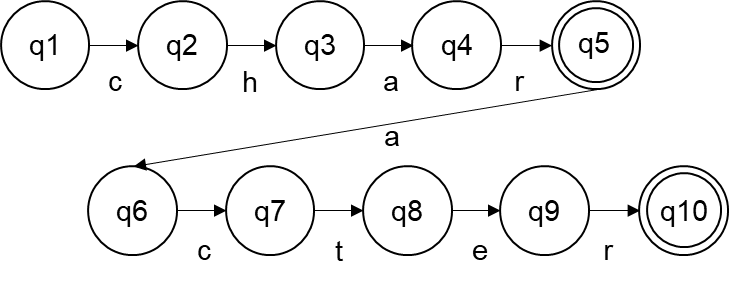
cas



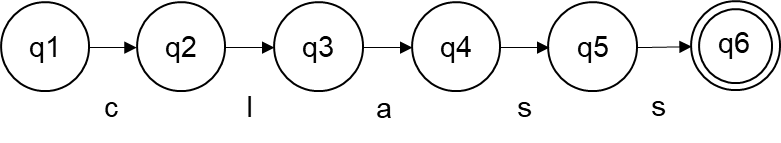
catch



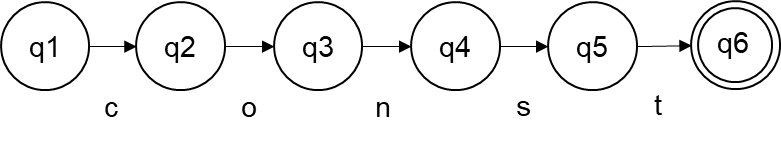
char



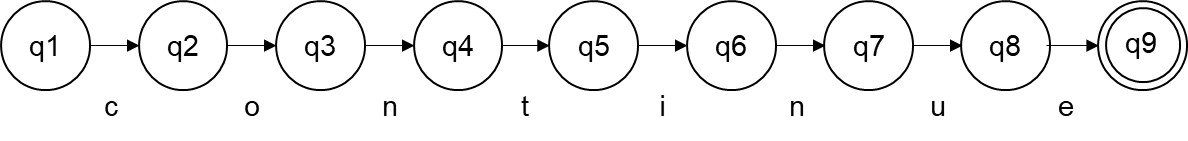
class



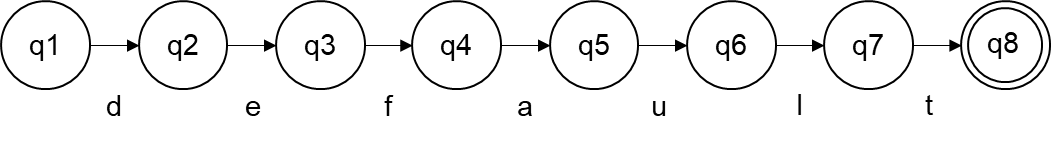
const



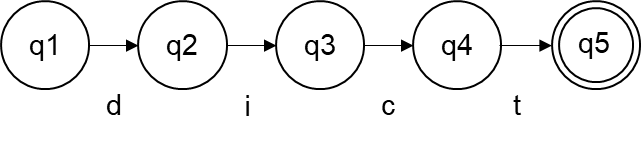
continue



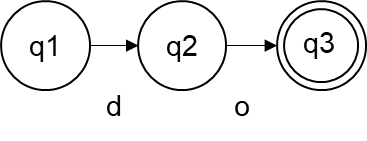
default



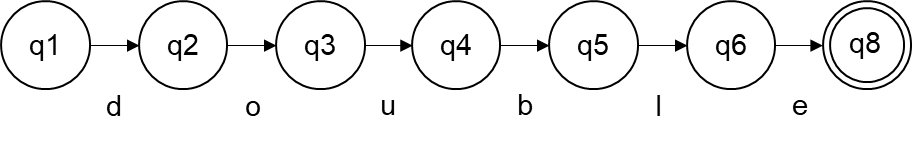
dict



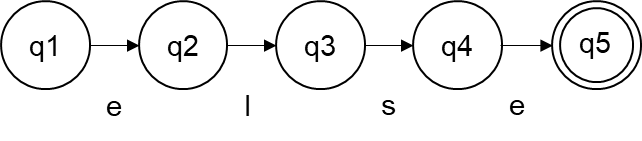
do



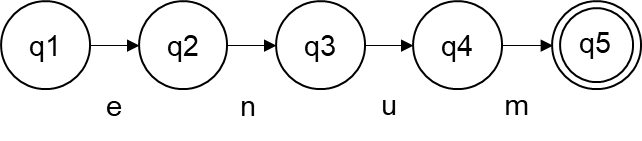
double



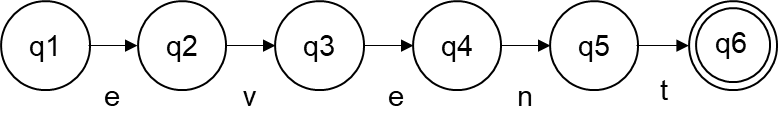
else



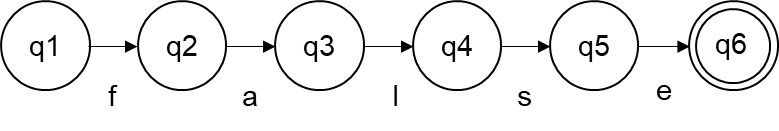
enum



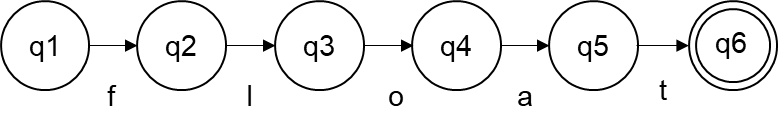
event



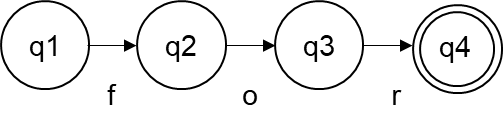
false



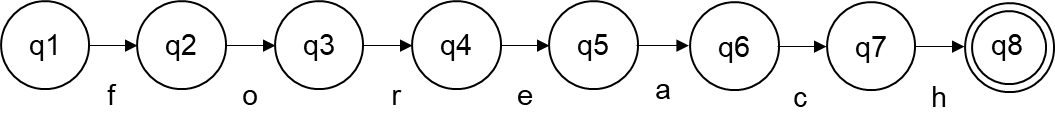
float



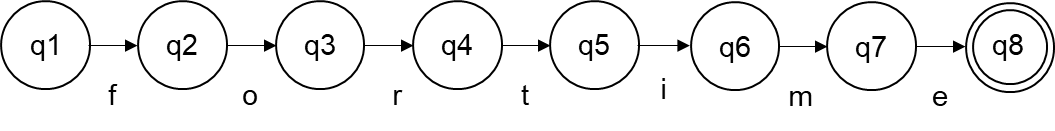
for



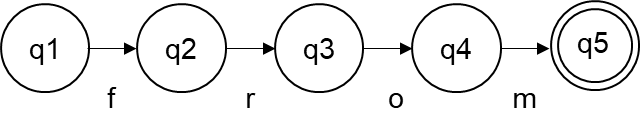
foreach



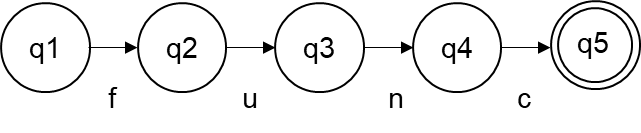
fortime



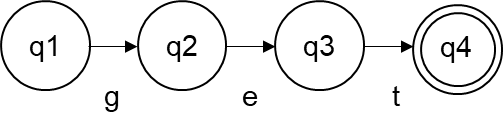
from



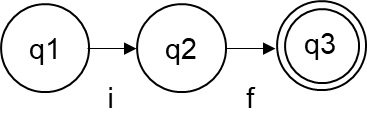
func



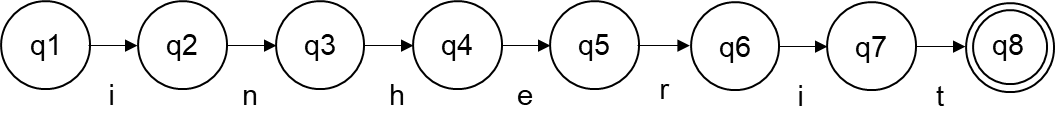
get



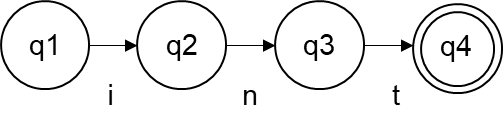
if



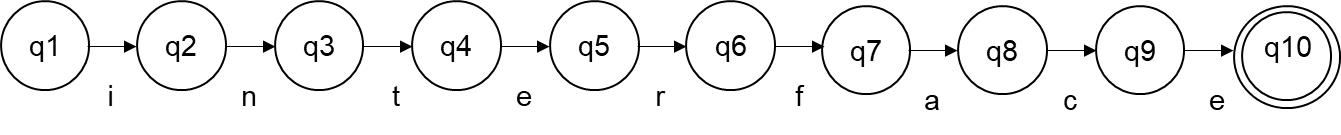
inherit



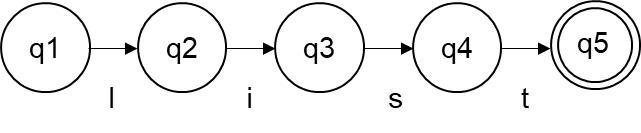
int



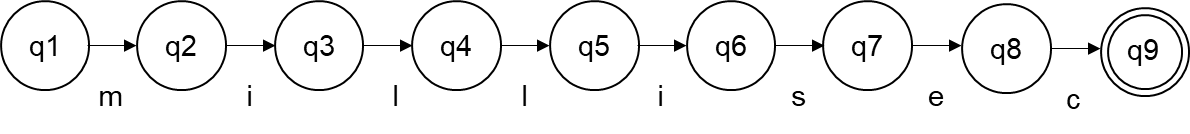
interface



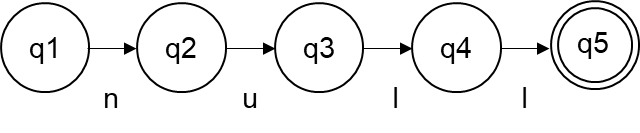
list



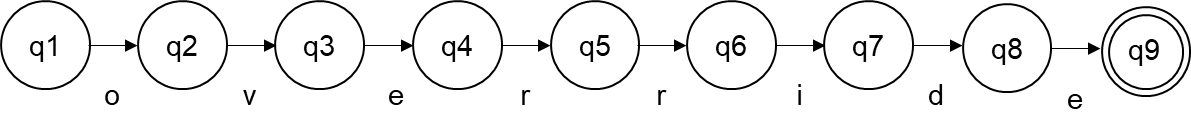
millisec



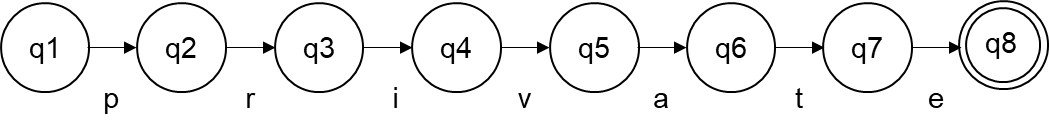
null



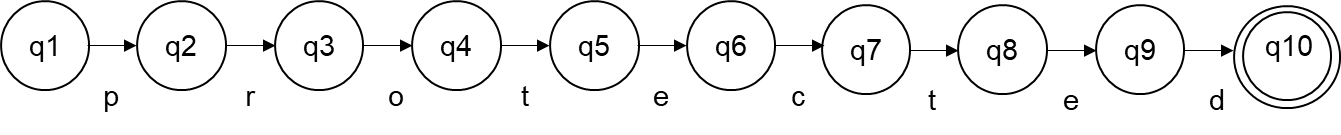
override



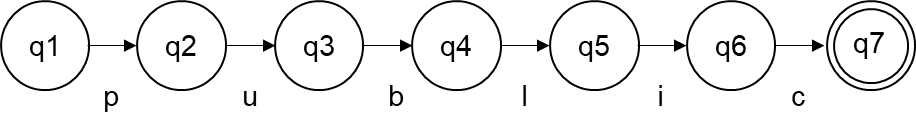
private



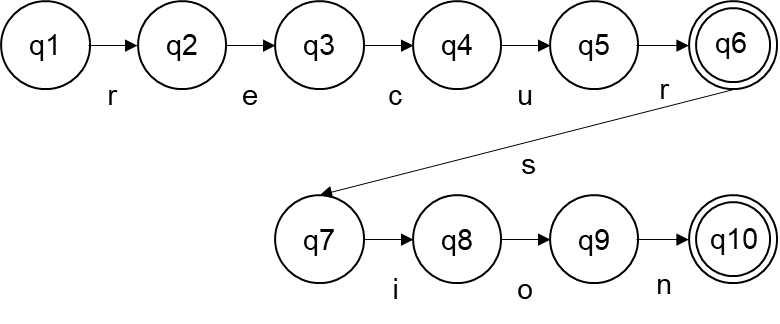
protected



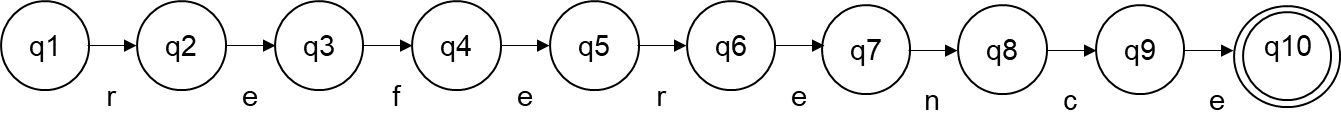
public



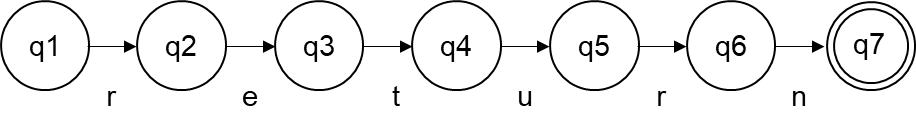
repeat



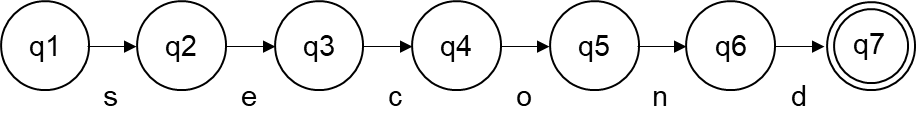
ref



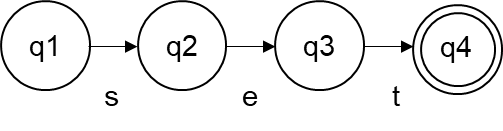
return



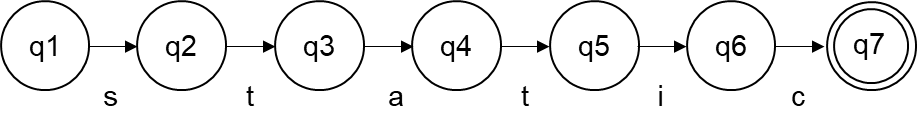
second



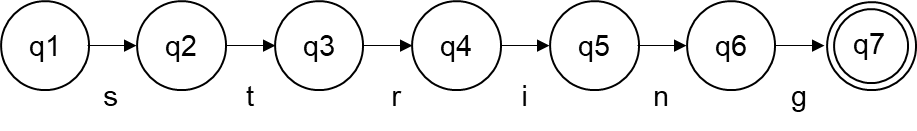
set



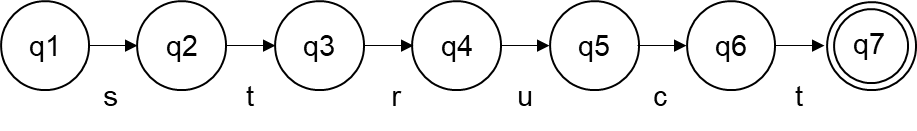
static



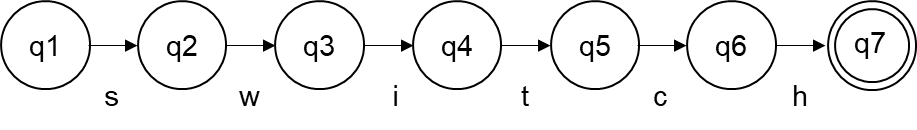
string



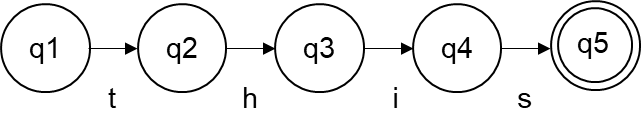
struct



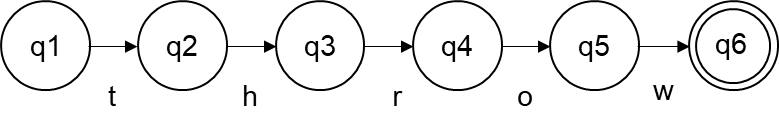
switch



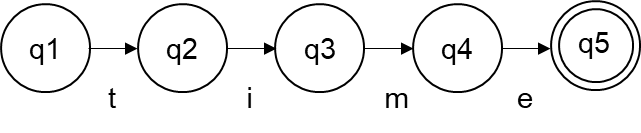
this



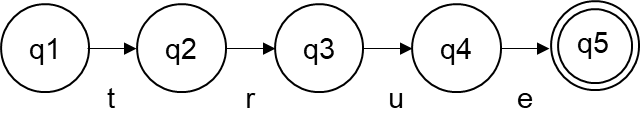
throw



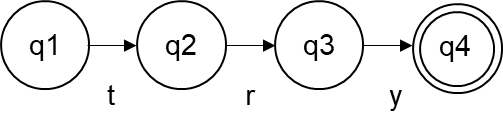
time



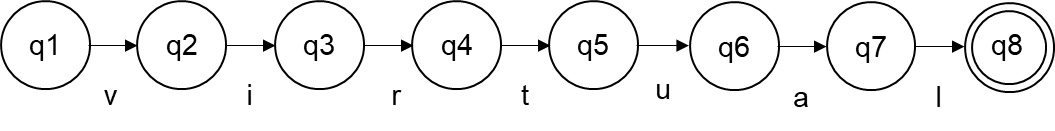
true



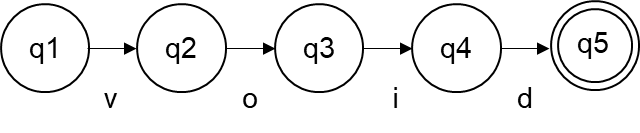
try



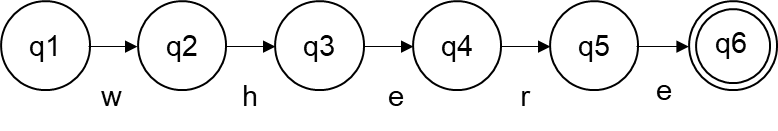
virtual



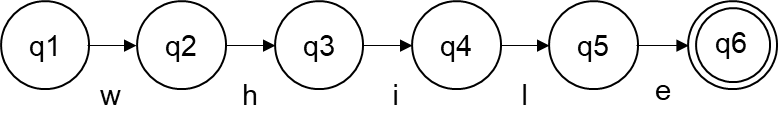
void



where



while



**5. NOISE WORDS**

|  |  |  |  |
| --- | --- | --- | --- |
| **Original notation** | **Shorthand** | **Noise word** | **Description** |
| boolean | bool | ean | A boolean value. |
| Ceiling | Ceil | ing | A mathematical function that rounds up and returns the smallest integer greater than or equal to a given number. |
| character | char | acter | A single character. |
| Combination | Comb | ination | The number of ways that a certain number of objects can be taken from a larger number of objects if the order does not matter |
| constant | const | ant | An immutable variable. |
| Cosecant | Csc | o,e,ant | The cosecant of an angle is the reciprocal of the sine. It is equal to the ratio of the length of the hypotenuse to the length of the side opposite the angle. |
| Cosine | Cos | ine | A trigonometric function about the ratio of the length of the adjacent side to the length of the hypotenuse |
| dictionary | dict | ionary | Represents a collection of keys and values. |
| enumeration | enum | eration | A special class that represents a group of constants. |
| function | func | tion | Used to define functions/methods. |
| integer | int | eger | A 32-bit signed integer. |
| Logarithm | Log | arithm | A mathematical function in which the power (or exponent) to which one base number must be raised — multiplied by itself — to produce another number. |
| millisecond | millisec | ond | A variable to represent time in milliseconds. |
| Permutation | Perm | utation | Describes the number of ways things can be ordered or arranged |
| repeatsion | repeat | sion | Used to define a Repeating method. |
| reference | ref | erence | Used to reference types. |
| Secant | Sec | ant | The secant of an angle is the reciprocal of the cosine. It is equal to the ratio of the length of the hypotenuse to the length of the adjacent side. |
| Sine | Sin | e | A trigonometric function about the ratio of the length of the side opposite the angle to the length of the hypotenuse |
| structure | struct | ure | A value type that can encapsulate data and related functionality. |
| Tangent | Tan | gent | A trigonometric function about the ratio of the length of the side opposite the angle to the length of the adjacent side. |

**6. COMMENTS**

**Single-line comments**

To indicate a single-line comment, use two forward slashes (//) at the beginning of a line.

|  |
| --- |
| **boop**  // This is a comment  // This is another comment |

**Multi-line comments**

Enclose a multi-line comment with asterisks (\*) on the inside, and forward slashes (/) outside.

|  |
| --- |
| **boop**  /\* This is a multi-line comment    Everything inside here is  ignored by the compiler.  \*/ |

**Formatting rules**

* The comment should be separated with a space from the delimiters.
* The comment should be in sentence case.

**7. WHITESPACES**

All whitespaces are ignored by the compiler but are used in coding to make the code readable. These include ( ) spaces, (\t) tabs, (\r) carriage returns, (\n) new lines, and (\v) vertical tabs.

**8. DELIMITERS**

**Semicolon (** ; **)**

Used to mark the end of a statement.

|  |
| --- |
| **boop**  MyFunction(); |

**Colon (** : **)**

Used to indicate inheritance. The inherit keyword can also be used in place of the inheritance colon.

|  |
| --- |
| **boop**  public class ChildClass: ParentClass { } |

**Curly braces (** {**,** } **)**

Used to set the scope for a block of code or language construct (e.g. modules, classes, structs, enums, loops, cases, method bodies).

|  |
| --- |
| **boop**  public module SampleLibrary  {  public class Main  {  func Main()  {  for (5)  {  //  }  }  }  } |

**Square brackets (** [**,** ] **)**

Used to define or access the index of a collection.

|  |
| --- |
| **boop**  public int[] Numbers = { 0, 1, 2, 3, 4 };  int myNumber = Numbers[3]; // Retrieves 3 |

**Parentheses (** (**,** ) **)**

Used for defining interface signatures, method signatures, method calls, containing expressions, and order of computation.

|  |
| --- |
| **boop**  public class Word (Printable)  {  public string Format;  public func ToString () -> string  {  return Format;  }  } |

**Angled brackets (** <**,** > **)**

Used for enclosing generic type parameters.

|  |
| --- |
| **boop**  public class Box<T> (ICollection)  {  public list Contents = { T };  } |

**Right arrow (** -> **)**

Used to define return types for functions.

|  |
| --- |
| **boop**  public func ToString() -> string  {  return “”;  } |

**Single quotes ( ‘ )**

Used to enclose string literals.

|  |
| --- |
| **boop**  int number = 5;  Print(format ‘The number is {number}’); |
| **Output**  The number is 5 |

**9. FREE AND FIXED-FIELD FORMATS**

**Free-field format structure**

BOOP adheres to a free-field format structure.

**10. EXPRESSIONS**

*Rules for evaluating expressions:*

1. **Arithmetic/Mathematical Operation**

*Precedence of Arithmetic/Mathematical Operation:*

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Description** | **Associativity** |
| 1 | \* | Multiplication | Left-to-Right |
| / | Division |
| % | Modulus |
| ^ or \*\* | Exponent |
| 2 | + | Addition | Left-to-Right |
| - | Subtraction |

1. **Boolean Operation (Logical and Relational)**

*Precedence of Boolean Operation (Logical and Relational):*

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Description** | **Associativity** |
| 1 | ! | Logical NOT | Right-to-left |
| 2 | < | Less than | Left-to-right |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| 3 | == | Equal to |
| != | Not equal to |
| 4 | && | Logical AND | N/A |
| 5 | || | Logical OR |
| 6 | = | Direct assignment | Right-to-left |
| += | Assignment by Sum |
| -= | Assignment by Difference |
| \*= | Assignment by Product |
| /= | Assignment by Quotient |
| %= | Assignment by Remainder |

1. **Unary Operator**

*Precedence of Unary Operators:*

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Sample Unary Operator** | **Description** | **Associativity** |
| 1 | a++ | Post-Increment | Left-to-Right |
| a-- | Post-Decrement |
| 2 | ++a | Pre-Increment | Right-to-Left |
| --a | Pre-Decrement |

**11. STATEMENTS**

**a. Declaration statements**

i. **Unassigned variable**

Declare an unassigned variable.

|  |
| --- |
| **Syntax**  <modifiers> <data\_type> <identifier>; |
| **boop**  int myNumber;  public int AnotherNumber; |

ii. **Assigned variable**

Declare an assigned variable.

|  |
| --- |
| **Syntax**  <modifiers> <data\_type> <identifier> = <value>; |
| **boop**  int myNumber = 5;  public int AnotherNumber = 10; |

iii. **Multiple variable declaration**

Declare multiple unassigned variables in one statement.

|  |
| --- |
| **Syntax**  <modifiers> <data type> <identifier>, ..., <identifier>; |
| **boop**  int myNumber, anotherNumber; |

iv. **Multiple assigned variables**

Declare multiple assigned variables in one statement.

|  |
| --- |
| **Syntax**  <modifiers> <data\_type> <identifier> = <value>, ..., <identifier> = <value>; |
| **boop**  public int MyNumber = 5, AnotherNumber = 2; |

v. **Constant variables**

Variables are mutable by default, declare them immutable with the const keyword.

|  |
| --- |
| **Syntax**  <modifiers> const <data\_type> <identifier> = <value>; |
| **boop**  public const int Pi = 3.14159; |

vi. **Implicitly typed variables**

Define a local variable without explicitly declaring its type.

|  |
| --- |
| **Syntax**  var <identifier> = <value>; |
| **boop**  var value = 10;  var message = “It’s okay”; |

vii. **Tuple types**

Define a container that can store multiple different data types.

|  |
| --- |
| **Syntax**  <modifiers> (<data\_type>, ..., <data\_type>) <identifier>; |
| **boop**  public (int, int) Coordinates;  public (string, bool) Message = (“Boolean”, true); // Initialized |

**Rules**

* Members declared without an access modifier are private by default.
* (v) Constants must be initialized with a value.
* (vi) Implicitly typed variables can only be declared locally and must be initialized with a value.

**b. Collections declaration**

i. **Arrays (uninitialized)**

Define an uninitialized array of elements.

|  |
| --- |
| **Syntax**  <modifiers> <data\_type>[] <identifier>; |
| **boop**  int[] numbers; |

ii. **Arrays (initialized)**

Declare an array initialized with values.

|  |
| --- |
| **Syntax**  <modifiers> <data\_type>[] <identifier> = { <value>, ..., <value> }; |
| **boop**  int[] numbers = { 0, 1, 2, 3, 4, 5 }; |

iii. **Arrays with size**

Declare an array with size.

|  |
| --- |
| **Syntax**  <modifiers> <data\_type>[] <identifier> = new[<int>]; |
| **boop**  // Declare an array with a size of 5  int[] numbers = new[5]; |

iv. **Multi-dimensional array**

Declare an array of arrays with a initialize or uninitialized size.

|  |
| --- |
| **Syntax**  <modifiers> <data\_type>[,\*] <identifier>; |
| **boop**  int[,] array2D; // uninitialized 2D  int[,] array2D = new[2, 2]; // with size 2D  int[,] array2D = {{1, 2}, {3, 4}}; // initialized 2D  int[,,] array3D; // uninitialized 3D  int[,,] array2D = new[2, 2, 2]; // with size 3D |

v. **Jagged array**

Declare an array of arrays that has an arbitrary number of elements.

|  |
| --- |
| **Syntax**  <modifiers> <data\_type>[ ][ ] <identifier>; |
| **Code**  int[][] myJaggedArray = {{ 1, 2, 3 }, {4}, { 5, 6 }}; |

vi. **Lists**

Declare a list of variables that share the same data type.

|  |
| --- |
| **Syntax**  <modifiers> list <data\_type> <identifier> = { <data\_types> };  // Initialized  <modifiers> list <data\_type> <identifier> = { <list\_contents> }; |
| **Code**  public list string Messages;  public list Messages =  {  “Hello”,  “world”,  “Hi”  }; |

vii. **Dictionaries**

Declare a list of key-value pairs that share the same data types.

|  |
| --- |
| **Syntax**  <modifiers> dict <identifier> = { <data\_types> };  // Initialized  <modifiers> dict <identifier> = { dict\_contents> }; |
| **Code**  public dict ConnectedIds = {int, bool};  // Initialized  public dict ConnectedIds =  {  {“1”, true},  {“2”, true},  {“3”, false}  }; |

**Rules**

* (v) Jagged array values must be first initialized before use.
* (vi, vii) Lists and dictionaries must be initialized with either its type or contents.
* (vi, vii) Lists and dictionaries must have the same data type for all of its contents.

**c. Functions declaration**

i. **Function**

Define a function with parameters and return type. Functions declared without a return type automatically returns void.

|  |
| --- |
| **Syntax**  <modifiers> func <identifier> (<params>) -> <return\_type>  {  // Method body  } |
| **Code**  public func Sum (int a, int b) -> int  {  return a + b;  } |

ii. **Functions with generic type parameters**

Define a function with generic type parameters for enforcing type safety.

|  |
| --- |
| **Syntax**  <modifiers> func <identifier><<generics>>(<params>) -> <return\_type>  {  // Method body  } |
| **Code**  public func Swap<T>(T a, T b)  {  T temp = a;  a = b;  b = temp;  } |

iii. **Abstract method**

Define an abstract method that must be overridden by child classes. Abstract methods do not define a method body.

|  |
| --- |
| **Syntax**  <access\_modifier> abstract func <identifier>(<params>) -> <return\_type>; |
| **Code**  public abstract class Printer  {  public abstract func DisplayMessage(string message);  } |

iv. **Virtual method**

Define a virtual method that can be overridden by child classes to provide new implementations. Virtual methods must define a method body.

|  |
| --- |
| **Syntax**  <access\_modifier> virtual func <identifier>(<params>) -> <return\_type>  {  // Method body  } |
| **Code**  public virtual func DisplayMessage(string message)  {  // Code  } |

v. **Override method**

Define an override method for a virtual or abstract method.

|  |
| --- |
| **Syntax**  <access\_modifier> override func <identifier>(<params>) -> <return\_type>  {  // New implementation  } |
| **Code**  public override func DisplayMessage(string message)  {  // New implementation  PrintLine(format “new: {message}”);  } |

vi. **Repeating function**

Define a Repeating function that executes repeatedly for a given period of time or iterations. A repeat function must define a last parameter of time or int.

|  |
| --- |
| **Syntax**  <modifiers> repeat func <identifier>(<params>, time|int <identifier>);  {  // Code  } |

vii. **Delayed function**

Define a delayed function that executes once after the defined duration has passed. A delayed function must define a last parameter of time.

|  |
| --- |
| **Syntax**  <modifiers> delayed func <identifier>(<params>, time|int <identifier>);  {  // Code  } |

viii. **Asynchronous function**

Define a function to be asynchronous that executes on a separate thread and can be manipulated. An asynchronous function can either be a standard func, repeat or delayed function.

|  |
| --- |
| **Syntax**  <modifiers> async func <identifier>(<params>);  {  // Code  } |

**Rules**

* (iii) Abstract methods can only be declared within an abstract class.
* (iv) Virtual methods must define a method body.
* (v) Overriding functions should have the same access modifiers, return type, and parameters as its overridden functions.
* (vi, vii) Repeat and delayed functions must always define a last parameter of type time.

**d. Static variables**

i. **Static declaration**

Define accessible members without an instance of its class.

|  |
| --- |
| **Syntax**  <modifiers> static <data\_type> <identifier>; |
| **Code**  public class MyMath  {  public static double Pi = 3.14159;  } |

**e. Input/Output Statements**

i. **Print**

|  |
| --- |
| Print(string format)  Print(Printable obj)  Print(string format, Printable obj)  Print(string format, Printable[] obj) |

Prints a message into the console with no line breaks.

|  |
| --- |
| **Code**  Print(“Hello “);  Print(“world.”);  Print(“test”); |
| **Output**  Hello world.test |

ii. **PrintLine**

|  |
| --- |
| PrintLine(string format)  PrintLine(Printable obj)  PrintLine(string format, Printable obj)  PrintLine(string format, Printable[] obj) |

Prints a message into the console including a line break.

|  |
| --- |
| **Code**  PrintLine(“Hello “);  PrintLine(“world.”);  PrintLine(“test”); |
| **Output**  Hello  world.  test |

iii. **ReadLine**

|  |
| --- |
| ReadLine(string format)  ReadLine(Printable obj)  ReadLine(string format, Printable obj)  ReadLine(string format, Printable[] obj) |

Reads the next line input. Passing a string will print the message before reading the input.

|  |
| --- |
| **Code**  ReadLine(“Enter number> “); |
| **Output**  Enter number> |

**Notes**

ReadLine operates as a blocking code, meaning it halts the program's execution and waits for user input before proceeding further.

**f. Assignment statements**

i. **Assignment**

Assign the value of the right-hand side to the left-hand side.

|  |
| --- |
| **Syntax**  <lhs> = <rhs>; |
| **Code**  myNumber = 5;  myNumber = anotherNumber;  myNumber = 6 + 9;  myNumber = RollDice(); |

ii. **Assignment by reference**

Assign variable by referencing its memory address.

|  |
| --- |
| **Syntax**  <lhs> = ref <rhs>; |
| **Code**  int myNumber = 5;  int alsoMyNumber = ref myNumber; |

iii. **Compound assignment**

Assign the value of the right-hand side to the left-hand side after being computed by the arithmetic operator.

|  |
| --- |
| **Syntax**  <lhs> <operator>= <rhs>; |
| **Code**  myNumber += 5;  myNumber -= anotherNumber;  myNumber \*= 6 + 9;  myNumber /= RollDice(); |

**g. Conditional Statements**

i. **If**

Executes a block of code when the condition is true.

|  |
| --- |
| **Syntax**  if (condition) {  // Code } |
| **Code**  bool IsValue = true;  if (IsValue) {  PrintLine(“Lmaok”); } |
| **Output**  Lmaok |

ii. **If-else**

Executes different blocks of code whether the condition is true or false.

|  |
| --- |
| **Syntax**  if (condition) {  // Body } else  {  //  } |
| **Code**  bool IsValue = false;  if (IsValue) {  PrintLine(“Lmaok”); } else  {  PrintLine(“Not lmaok”);  } |
| **Output**  Not lmaok |

iii. **If-else-if**

An if-else-if statement tests multiple conditions to perform different blocks of code. Executes only the first block of code that has a condition that equates to true.

|  |
| --- |
| **Syntax**  if (condition) {  // Code } else if (condition) {  // Else if code } else  {  // Else code  } |
| **Code**  int MyNum = 5;  bool IsValue = false;  if (IsValue) {  PrintLine(“Lmaok”); } else if (MyNum < 10)  {  PrintLine(“Maybe lmaok?”);  } else  {  PrintLine(“Not lmaok”);  } |
| **Output**  Maybe lmaok? |

iv. **Switch**

A switch statement allows the execution of different blocks of code based on the value of an expression.

|  |
| --- |
| **Syntax**  switch expression {  <case>: { // Single statement }  <case>:  {  // Multiple statements  }  default: { // Default case } } |
| **Code**  int MyNum = 3;  switch MyNum {  case 1: { PrintLine(“It’s Monday”); }  case 5:  {  // It’s Friday  PrintLine(“Yay!”);  PrintLine(“It’s Friday”);  }  default:  {  PrintLine(“It’s not Monday or Friday”);  }  } |
| **Output**  It’s not Monday or Friday |

v. **Nested-if**

Executes blocks of code based on multiple nested conditions.

|  |
| --- |
| **Syntax**  if (condition) {  if (condition)  {  // Code  }  } |
| **Code**  int myNum = 3;  bool isEnabled = true;  if (isEnabled) {  PrintLine(“It is enabled.”);  if (myNum > 0)  {  PrintLine(“More than zero.”);  }  } |
| **Output**  It is enabled.  More than zero. |

**h. Iterative Statements**

i. **For**

A standard for-loop that iterates through an iterable object.

|  |
| --- |
| **Syntax**  for <type> a in b // Where b is an Iterable object  {  // Code  } |
| **Code**  int[] num = { 1, 2, 3, 4, 5 }  for int a in num  {  Print(a); } |
| **Output**  12345 |

ii. **For (int)**

Iterates through the block of code for an *int* number of times. To access the index, a built-in local variable iis present.

|  |
| --- |
| **Syntax**  for int {  // Code } |
| **Code**  for 5 {  Print(i);  Print(“Hi mom”); } |
| **Output**  0Hi mom1Hi mom2Hi mom3Hi mom4Hi mom |

iv. **Fortime**

Executes the block of code repeatedly for a defined duration.

|  |
| --- |
| **Syntax**  fortime time {  // Code } |
| **Code**  second duration = 2;  fortime duration {  PrintLine(”Hello”); } |
| **Output**  Hello  Hello  Hello  Hello  ...  // Continues to execute for 2 seconds |

v. **Foreach**

Iterates through elements over a collection.

|  |
| --- |
| **Syntax**  foreach var <id> in <ICollection>  {  // Code  } |
| **Code**  list numbers = { 5, 4, 3, 2, 1 }  foreach int number in numbers) {  PrintLine(format ”{numbers}”); } |
| **Output**  5  4  3  2  1 |

vi. **Nested-for**

A loop inside loop that iterates through the block of code given the conditions.

|  |
| --- |
| **Syntax**  // Both for statements can be any and mixed of the mentioned above  for (initializer; condition; iterator)  {  for (initializer; condition; iterator)  {  // Code  }  } |
| **Code**  for (int i = 0; i < 2; i++)  {  for (int j = 0; j <= 2; j++)  {  PrintLine(format “{i}, {j}”);  }  } |
| **Output**  0, 0  0, 1  0, 2  1, 0  1, 1  1, 2 |

vii. **While**

Executes the block of code repeatedly as long as the condition equates to true.

|  |
| --- |
| **Syntax**  while (condition) {  // Code } |
| **Code**  int i = 0;  while (i <= 5) {  Print(“{i}”);  i++; } |
| **Output**  012345 |

viii. **Do-while**

Executes the block of code once, checks the condition and repeats execution as long as the condition equates to true.

|  |
| --- |
| **Syntax**  do {  // Code } while (condition); |
| **Code**  int i = 0;  do {  print(format ”{i}”);  i++; } while (i <= 5); |
| **Output**  012345 |

**i. Time variables declaration**

Declare a time variable.

|  |
| --- |
| **Syntax**  second <identifier>;  millisecond <identifier>; |
| **Code**  second duration = 10;  millisecond delay = 500; |

Time variables can also be implicitly declared but requires numeric suffixes to be appended.

|  |
| --- |
| **Code**  var duration = 10s; // Ten seconds  var delay = 0.5s; // Five hundred milliseconds |

**j. Other declarations**

**Importing modules**

Import a module to be used in the current file with the use keyword. Members can be directly accessed without indicating the module’s name.

|  |
| --- |
| **Syntax**  use <module>; |
| **Code**  use System;  use Graphics; |

**Importing select members from a module**

Import selected members from a library to be used in the current file.

|  |
| --- |
| **Syntax**  use { <member>, ..., <member> } from <module>; |
| **Code**  use { Constants } from Math;  use { Pi, Gravity } from Math.Constants; |

**Main function**

Like in other programming languages, there is the “main” function that serves as the entry point for the execution of a BOOP program.

|  |
| --- |
| **Code**  func Main()  {  // Body  } |

The main function can also declare a parameter of string[] to pass command-line arguments to a program when it's executed from the command line or terminal.

**Notes**

* The name of the main function is case-sensitive, meaning that a program with its main function named “main” will not execute.

**III. Time-oriented Principle**

This section describes the time-oriented principle BOOP has.

**Fortime statement**

The fortime statement is used to execute a block of code repeatedly for a defined duration of time. This functionality is particularly useful for scenarios where tasks need to be performed repeatedly *over a specific time period*. The statement takes a parameter of type time—which can be either be a second or millisecond variable, representing the duration of the execution.

**Syntax**

|  |
| --- |
| **Syntax**  fortime (time)  {  // Code to be executed  } |
| **Code**  func Main()  {  second duration = 2;  fortime (duration)  {  PrintLine("Hello");  }  } |
| **Output**  Hello  Hello  Hello  Hello  Hello  ... |

In the above example, the message “Hello” will be printed repeatedly for the entire duration of two seconds. In a simple scenario of a one-line print, the intervals between these print statements would be extremely short, with the execution rate ranging from hundreds to several thousand lines per second. However, the length of these intervals will ultimately depend on the complexity of the block of code.

**Repeating functions**

A repeating function is a type of function that calls itself repeatedly for a defined duration of time. This is similar to a fortime statement, albeit in function form. A repeating function is declared by prepending the keyword repeat in a func declaration.

In order to define the duration, the function must define a last parameter that represents it. It can be in any identifier, but its type should be of type time. This parameter can also be read as a local variable inside of the function.

**Syntax**

|  |
| --- |
| **Code**  public class Main  {  // Declare a repeating function by adding ‘repeat’ before ‘func’  // A last parameter of time must be defined  public repeat func PrintMessage(string message, time duration)  {  PrintLine(format ‘message: {duration}’);  }  func Main()  {  second duration = 5;  PrintMessage(‘Hello’, duration);  }  } |
| **Output**  Hello 5s  Hello 5s  Hello 5s  Hello 5s  Hello 5s  ...  // Executes for 5 seconds |

If you want to omit the units of the second.

|  |
| --- |
| **Code**  string durationNoUnits = duration.NoUnits; // Retrieve property |

**Notes**

* Repeating functions still run on the main thread, ensuring synchronous execution of the task.
* The repeat function executes the tasks as fast the system allows. For defining repeating functions with regular intervals, see Repeating and Delayed Functions.
* Changing the value of the duration variable will have no effect as its value is based on what is passed when the function was called.

**Delayed functions**

A delayed function is a type of function that calls itself **once** after the defined duration of time has passed. Similar to a repeating function, this function must also define a last parameter of duration. A repeating function is declared by prepending the keyword delayed in a func declaration.

**Syntax**

|  |
| --- |
| **Code**  public class Main  {  public delayed func PrintMessage(string message, time duration)  {  PrintLine(message);  }  func Main()  {  second delay = 5;  PrintLine(‘Hello’);  PrintMessage(‘World’, delay);  }  } |
| **Output**  Hello  *Elapsed: 0ms*  World  *Elapsed: 5000ms* |

It might not be evident in the example output, but the idea is that “Hello” will be printed first, then after the defined duration of five seconds have passed, the second print function will run printing “World”.

Used on their own, delayed functions might not seem useful as they inhibit the execution of the program due to their blocking nature. However, the potential of delayed functions will be utilized when they are used together with repeat and async functions.

**Repeating and Delayed functions**

Although the repeat and delayed keywords cannot be applied to the same function, they can be used together by nesting the functions. The behavior of the program will differ depending on their nesting order.

**repeat(delayed())**

This pattern will repeat the execution of the delayed function at intervals defined by the amount of delay the delayed function has.

**delayed(repeat())**

This pattern on the other hand, will just delay the execution of the repeating function.

**Example**

|  |
| --- |
| **Code**  public class Main  {  repeat func PrintRepeating(string message, second delay, time duration)  {  PrintDelayed(message, delay);  }  delayed func PrintDelayed(string message, time delay)  {  PrintLine(message);  }  func Main()  {  string message = ‘What’s up!’;  second totalTime = 5;  second delay = 1;  PrintRepeating(message, delay, totalTime);  }  } |
| **Output**  What’s up!  *Elapsed: 1000ms*  What’s up!  *Elapsed: 2001ms*  What’s up!  *Elapsed: 3001ms*  What’s up!  *Elapsed: 4002ms* |

In the above example, the repeat function nests the delayed function. This results to the repeated execution of the delayed function at one second intervals for the duration of five seconds, defined by the repeating function.

**Note**

* The program will always try to remain consistent with its time calculations, however, the execution will stop immediately after reaching its duration. In the above example, it would be to normal think that one more print should occur, but in cases that the time had accrued\*, the last interval would happen at *5002* milliseconds. Thus, it would not execute as the repeating function had already been terminated when it reached the end of its duration of *5000* milliseconds.

*\*The time may vary depending on exterior factors such as system latency or interruptions which can affect processing time.*

# **Async functions**

Async functions is a type of function that allows asynchronous, non-blocking execution in the program. Asynchronous functions are declared by prepending func with the async keyword.

**Syntax**

|  |
| --- |
| **Code**  public class Main  {  public async func PrintMessage(string message)  {  Print(message);  }  func Main()  {  second duration = 5;  PrintMessage(‘Hello’, 5);  PrintLine(‘Test’);  }  } |
| **Output**  Test  Hello  Hello  Hello  ...  // Executes for 5 seconds |

Async functions can also be used to await the completion of other async functions.

**Note**

* Like all other programming languages, asynchronous functions run on a separate thread.

**Async, Repeat, and Delayed functions**

Async can also be used with repeat and delayed functions to do more specialized tasks. It is also important to note that async functions do not support returning types.

|  |
| --- |
| **Code**  public class Main  {  public async repeat PrintMessage(string message, time delay, time duration)  {  await PrintLater(message, delay);  }  public async delayed PrintLater(string message, time delay)  {  PrintLine(message);  }  func Main()  {  second duration = 5;  second delay = 1;  PrintMessage(‘Hello’, delay, duration); // Async function  SomeSyncFunc();  }  } |
| **Output**  Hello  *Elapsed: 1000ms*  Hello  *Elapsed: 2000ms*  Hello  *Elapsed: 3000ms*  Hello  *Elapsed: 4000ms*  Hello  *Elapsed: 5000ms* |

It might not be evident in the preceding example, but the idea is the async function “PrintMessage” will run on a separated thread, and the program will continue execution of SomeSyncFunc().

**Time-based Events**

Time-based events can be achieved using delayed functions.

**Example**

|  |
| --- |
| **Code**  // Helper class that prints chat messages  public class ChatHandler  {  // Define an event that fires every time a message is sent  // It will take a parameter of a string and boolean  public event OnSendMessage { string, boolean }  // A method is defined to encapsulate the event  // Events can only be fired from within the class it is declared  public delayed func SendMessage(string message, boolean isHost, time delay)  {  // Invoke the event  OnSendMessage(message, isHost);  }  }  // Another class that will listen to the OnSendMessage event  // A good example would be something that handles the UI  public class ChatListener  {  public string Message;  public bool IsHost;  public func UpdateMessage(string message, bool isHost);  {  Message = message;  IsHost = isHost;  }  }  public class Main  {  func Main()  {  millisecond delay = 100;  var chatHandler = ChatHandler();  var chatListener = ChatListener();  // Subscribe the UpdateMessage method to the event  chatHandler.OnSendMessage => chatListener.UpdateMessage;  // The chat handler sends a message  chatHandler.SendMessage(“Hello world.”, true, delay);  }  } |
| **Output**  true: Hello world.  *Elapsed: 100ms* |

A lot of things are going on with the preceding example so let’s break it down one by one.

The class ChatHandler declares a public event OnSendMessage, that takes parameters of string and bool. This tells that the functions that should subscribe to this event must define the same parameters. The event is public, meaning that other classes can view it and subscribe to it, however, its invocation can only be done inside the class it is declared—ChatHandler. In our example where ChatHandler has no inner functionality, we had just declared a wrapper function SendMessage() to invoke the event inside the class. Note that this is a delayed function and will behave accordingly.

The class ChatListener defines a function UpdateMessage, which will subscribe to the OnSendMessage event. This means that every time the ChatHandler fires the event, the UpdateMessage will execute as well.

**IV. Math-friendly Principle**

This section shows the implementation of mathematical symbols and functions that BOOP offers. The incorporation of these mathematical symbols and functions into BOOP language allows developers to build code that closely mirrors mathematical expressions, improving both clarity and performance.

* 1. **Math Symbols**

|  |  |
| --- | --- |
| Square root **(**√**)** | Pi **(**π**)** |
| Mean **(x̄)** | Theta **(**θ**)** |
| Product notation **(**∏**)** | Imaginary unit **(**ι**)** |
| Summation notation **(**Σ**)** | Absolute Value **(**|n|**)** |

Currently, the basic math symbols that the BOOP Language offers are the following:

**Square root**

|  |
| --- |
| **Code**  // Square root  result = √169; // Equates to 13  result = √GetRandomInt(); // Can also be expressions  result = 3√64; // Radical with an index of 3 (cube root)  result = √(3√64); // Nested radicals |

The square root symbol (√) will operate on the first single expression that follows it. The symbol can also take an index value by putting another expression on the left side of the symbol.

**Mean**

|  |
| --- |
| **Code**  int a = 1, b = 2, c = 3;  int[] num = {1, 2, 3};  float mean1 = **x̄**(a, b, c); // Calculates the mean of variables a, b, and c  float mean2 = **x̄**(num); // Can be an array |

The mean symbol (**x̄**) will operate on an array that follows it.

**Product Notation**

|  |
| --- |
| **Code**  // Product Notation  // Generally the same as 1\*2\*3\*4\*5 or 5!(factorial)  int[] array = {1, 2, 3, 4, 5};  result = ∏(1,5); // Equates to 120  result = ∏(array); // Can also take arrays |

**Summation Notation**

|  |
| --- |
| **Code**  // Summation Notation  result = Σ(1,5); // Equates to 15  result = Σ(array); // Can also take arrays |

**Pi**

|  |
| --- |
| **Code**  // Pi  result = πr^2; // Area of a circle |

**Theta**

|  |
| --- |
| **Code**  // Theta  int θ = 45; // Can be used as a variable  result = cos(θ) // Cosine of angle 45 |

**Note**

* Although it may look like a function call, the symbols actually function as operators.

**Imaginary Unit**

|  |
| --- |
| **Code**  // Imaginary units  // Imaginary numbers in exponentiation  int imagi = ι ^ 23415; // Equates to -ι  // Sqrt  imagi = √–25; // Equates to 5ι  /\* Note: Imagi numbers/expressions are considered complex numbers (17ι == 0 + 17ι)  and real numbers can be classified as complex as well (0ι + 100 == 100)  due to the existence of 0 \*/  // Imaginary numbers in basic arithmetic operators  // Add  imagi = 2ι + 15ι; // Equates to 17ι imagi = 0 + 2ι; // Equates to 2ι  // Subtract  imagi = 77 - ι√5; // Equates to 77 - 2.2360679775ι  // Multiply imagi = 77ι \* 15ι; // Equates to –1155 (77 \* 15 \* ι \* ι)  // Division  imagi = 100ι/10ι; // Equates to 10  // Modulus  /\* in complex/imaginary nums, modulus refers to the distance of the number  from the origin in the complex plane \*/  imagi = |2ι|; // Aka modulus of ι, equates to 2  imagi = |15 + 2ι|; // Equates to 15.132745950422  imagi = |100ι/10ι|; // Equates to 10 |

**Absolute Value**

|  |
| --- |
| **Code**  // Absolute value  result = |-1337| // Equates to 1337 |

* 1. **Math Functions**

The following are the mathematical functions that the BOOP Language offers:

|  |  |
| --- | --- |
| Sine (**Sin(θ)**)  Inverse Sine (**Asin(θ)**) | Least Common Multiple  **LCM()** |
| Cosine (**Cos(θ)**)  Inverse Cosine (**Acos(θ)**) | Ceiling  **Ceil(x)** |
| Tangent (**Tan(θ)**)  Inverse Tangent (**Atan(θ)**) | Floor  **Floor(*x*)** |
| Secant (**Sec(θ)**)  Inverse Secant (**Asec(θ)**) | Logarithm  **Log(a)**  **Log(base, a)** |
| Cosecant (**Csc(θ)**)  Inverse Cosecant (**Acsc(θ)**) | Permutation  **Perm(n, r)** |
| Cotangent (**Cot(θ)**)  Inverse Cotangent **(Acot(θ)**) | Combination  **Comb(n, r)** |
| Greatest Common Divisor  **GCD()** | |

**Common Math Functions**

**Greatest Common Divisor**

|  |
| --- |
| **Code**  // Greatest Common Divisor  int [] arr = {4, 8, 12};  public int Result = GCD (12, 18); // can directly input the values  public int ArrayResult = GCD (arr) // arrays can also be used |

The greatest common divisor function is used to calculate the greatest common factor of the numbers that are inputted.

**Least Common Multiple**

|  |
| --- |
| **Code**  // Least Common Multiple  int [] arr = {4, 8, 12};  public int Result = LCM (12, 18); // can directly input the values  public int ArrayResult = LCM (arr); // arrays can also be used |

The least common multiple function is used to compute the lowest possible number that can be divisible by both numbers.

**Ceiling and Floor**

|  |
| --- |
| **Code**  // Ceiling  float Num = Ceil (4.45); // returns 5  // Floor  float Num = Floor (4.45); // returns 4 |

**Logarithm**

|  |
| --- |
| **Code**  // Basic Logarithm  int Num = Log (100); // equates to 2  //Logarithm with specified base  int Num = Log (2, 8); // equates to 3 |

Boop Language has two different ways to compute logarithms. First, the basic logarithm with the syntax of Log(). Basic logarithm has a specified base of 10, so any number that was input on the function will compute its logarithm with the basic base of 10. The second one is the logarithm with a specified base. With the syntax of Log(base, a), the user will be the one to input the base, then the number to be computed.

**Permutation and Combination**

|  |
| --- |
| **Code**  // Permutation  int Result = Perm (5,3); // equates to 60  // Combination  int Result = Comb (5,3); // equates to 10 |

Permutation, Perm(n, r), represents the number of permutations of selecting "r" elements from a set of "n" distinct elements, where *order matters*. In the code above, ‘Perm (5,3)’ means finding the number of ways to select 3 elements from a set of 5 distinct elements in a specific order.

Combination, Comb(n, r), on the other hand, represents the number of combinations of selecting "r" elements from a set of "n" distinct elements, where *order does not matter*. The ‘Comb (5,3)’ above means finding the number of ways to select 3 elements from a set of 5 distinct elements without considering the order of selection.

**Trigonometric Functions**

The integration of trigonometric functions in BOOP Language is essential as it remains a fundamental part of mathematical computations and geometry. Understanding trigonometry can enhance one's ability to solve a variety of programming problems, even at a basic level.

|  |
| --- |
| **Code**  int θ = 45;  double CosineAngle = Cos(θ); // theta can be used as a variable  double SineAngle = Sin(30); // equates to 0.5  double ArcSine = Asin(0.5); // equates to 30 degrees  //conversion to inverse function and vice versa  // takes the value of CosineAngle and convert to Inverse Cosine  double ArcCos = Acos(CosineAngle);    // takes the value of SineAngle and convert to Inverse Sine  double ArcSin = Asin(SineAngle);  // takes the value of ArcSine and convert to Sine angle  double Sine = Sin(ArcSine); |

This is a simple code that demonstrates the use of trigonometric functions in programming. Sine, cosine, tangent, secant, cosecant, and cotangent are in degrees. While the inverse functions such as arc sine will be in the radian units as shown in the ‘Asin (0.5)’ example above, where 0.5 is in radians.

* 1. **Sample Functions that use Math Symbols**

**Distance formula**

|  |
| --- |
| **Code**  public static func DistanceFormula<T> (T x1, T x2, T y1, T y2) -> T  {  return √((x2 – x1)^2 + (y2 – y1)^2);  } |

**Pythagorean theorem**

|  |
| --- |
| **Code**  public func CalculateHypotenuse<T> (T a, T b) -> T  {  return √(a^2 + b^2);  }  float result = CalculateHypotenuse<float>(5, 5);  Print(result);  // An old implementation would look like this  use Math.Functions;  public func CalculateHypotenuse<T> (T a, T b) -> T  {  return Sqrt(Pow(a, 2), Pow(b, 2));  } |

**Quadratic formula**

|  |
| --- |
| **Code**  public static func Quadratic<T> (T a, T b, T c) -> (T, T)  {  /\* Since the formula involves the ± sign,  the computation will perform the + and – separately.  \*/  T x = (-b + √(b^2 -4 \* a \* c)) / 2 \* a; // Addition operation  T y = (-b - √(b^2 -4 \* a \* c)) / 2 \* a; // Subtraction operation  return (x, y)  } |

* 1. **BOOP Language Server (Keywords and Automatic Substitution)**

Implementing the Language Server Protocol (LSP) in the context of the BOOP Language will greatly enhance the development experience by introducing features such as automatic math symbol substitution. The team will develop a language server specifically tailored for BOOP Language that understands the syntax of the language. The BOOP Language Server will analyze the user’s code in real-time, identifying keywords and mapping them for their corresponding mathematical symbols.

As the user types, the BOOPLS will provide intelligent suggestions for symbols based on the entered keywords, utilizing the LSP’s Autocompletion capabilities. The IDE, integrated with the BOOPLS, will then seamlessly substitute the suggested symbols for the entered keywords, making the coding process more intuitive and reducing the cognitive load on beginners. The implementation of LSP in BOOP Language will not only assist users in adopting mathematical symbols more efficiently but also contribute to a smoother and more use-friendly development experience.

**Keywords**

The BOOP Language’s mathematical symbols with corresponding keywords to implement the automatic suggestion and substitution are as follows:

|  |  |  |
| --- | --- | --- |
| **Symbol Name** | **Keyword** | **Symbol** |
| Square root | sqrt | √ |
| Mean | mean | **x̄** |
| Product notation | pnotation | ∏ |
| Summation notation | snotation | Σ |
| Pi | pi | π |
| Theta | theta | θ |
| Imaginary unit | imgnr | ι |
| Absolute Value | abs | |n|  ’n’ is a place holder for a number |

**V. Syntax**

This section presents the derivations of the BOOP language. The parser derivation pertains to the derivation automatically generated by the Parser program.

**Declaration statements**

**Variable declaration**

|  |
| --- |
| **Code**  int a; |

**Parser derivation**

INT

dataType

dataType IDENTIFIER

varDeclaration

varDeclaration END\_STATEMENT

varDeclarationStatement

**Assigned variable declaration**

|  |
| --- |
| **Code**  int a = 1; |

**Parser derivation**

INT

dataType

dataType IDENTIFIER

varDeclaration

varDeclaration OP\_ASSIGN

varDeclaration assignmentOperator

varDeclaration assignmentOperator NUM\_LITERAL

varDeclaration assignmentOperator value

varDeclaration

varDeclaration END\_STATEMENT

varDeclarationStatement

**Multiple variables declaration**

|  |
| --- |
| **Code**  int a, b, c; |

**Parser derivation**

INT

dataType

dataType IDENTIFIER

varDeclaration

varDeclaration COMMA

varDeclaration COMMA IDENTIFIER

varDeclaration

varDeclaration COMMA

varDeclaration COMMA IDENTIFIER

varDeclaration

varDeclaration END\_STATEMENT

varDeclarationStatement

**Multiple assigned variables declaration**

|  |
| --- |
| **Code**  int a = 1, b = 2, c = 3; |

**Parser derivation**

INT

dataType

dataType IDENTIFIER

varDeclaration

varDeclaration OP\_ASSIGN

varDeclaration assignmentOperator

varDeclaration assignmentOperator NUM\_LITERAL

varDeclaration assignmentOperator value

varDeclaration

varDeclaration COMMA

varDeclaration COMMA IDENTIFIER

varDeclaration

varDeclaration OP\_ASSIGN

varDeclaration assignmentOperator

varDeclaration assignmentOperator NUM\_LITERAL

varDeclaration assignmentOperator value

varDeclaration

varDeclaration COMMA

varDeclaration COMMA IDENTIFIER

varDeclaration

varDeclaration OP\_ASSIGN

varDeclaration assignmentOperator

varDeclaration assignmentOperator NUM\_LITERAL

varDeclaration assignmentOperator value

varDeclaration

varDeclaration END\_STATEMENT

varDeclarationStatement

**Input statement | Function call | Assignment by expression**

|  |
| --- |
| **Code**  string name = Read(); |

**Parser derivation**

STRING

dataType

dataType IDENTIFIER

varDeclaration

varDeclaration OP\_ASSIGN

varDeclaration assignmentOperator

varDeclaration assignmentOperator IDENTIFIER

varDeclaration assignmentOperator IDENTIFIER LEFT\_PAREN

varDeclaration assignmentOperator IDENTIFIER LEFT\_PAREN RIGHT\_PAREN

varDeclaration assignmentOperator IDENTIFIER LEFT\_PAREN RIGHT\_PAREN END\_STATEMENT

varDeclaration assignmentOperator functionCallStatement

assignmentStatement

**Output statement | Function call with arguments**

|  |
| --- |
| **Code**  Write(‘Hello!’); |

**Parser derivation**

IDENTIFIER

IDENTIFIER LEFT\_PAREN

IDENTIFIER LEFT\_PAREN STRING\_LITERAL

IDENTIFIER LEFT\_PAREN value

IDENTIFIER LEFT\_PAREN value RIGHT\_PAREN

IDENTIFIER LEFT\_PAREN value RIGHT\_PAREN END\_STATEMENT

functionCallStatement

**Assignment statement**

|  |
| --- |
| **Code**  Pi = 3.14f; |

**Parser derivation**

IDENTIFIER

IDENTIFIER assignmentOperator

IDENTIFIER assignmentOperator NUM\_LITERAL

IDENTIFIER assignmentOperator NUM\_LITERAL OP\_DOT

IDENTIFIER assignmentOperator NUM\_LITERAL OP\_DOT NUM\_LITERAL

IDENTIFIER assignmentOperator NUM\_LITERAL OP\_DOT NUM\_LITERAL NUM\_SUFFIX

IDENTIFIER assignmentOperator value END\_STATEMENT

assignmentStatement

**Conditional statements**

**If statement**

|  |
| --- |
| **Code**  if (a > b)  {  return a;  } |

**Parser derivation**

IF

IF LEFT\_PAREN

IF LEFT\_PAREN IDENTIFIER

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET IDENTIFIER

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET IDENTIFIER RIGHT\_PAREN

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET IDENTIFIER RIGHT\_PAREN LEFT\_BRACE

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET IDENTIFIER RIGHT\_PAREN LEFT\_BRACE RETURN

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET IDENTIFIER RIGHT\_PAREN LEFT\_BRACE RETURN IDENTIFIER

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET IDENTIFIER RIGHT\_PAREN LEFT\_BRACE RETURN IDENTIFIER END\_STATEMENT

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET IDENTIFIER RIGHT\_PAREN LEFT\_BRACE RETURN IDENTIFIER END\_STATEMENT RIGHT\_BRACE

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET IDENTIFIER RIGHT\_PAREN LEFT\_BRACE returnStatement RIGHT\_BRACE

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET IDENTIFIER RIGHT\_PAREN LEFT\_BRACE statement RIGHT\_BRACE

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET IDENTIFIER RIGHT\_PAREN LEFT\_BRACE body RIGHT\_BRACE

IF LEFT\_PAREN IDENTIFIER relationalOperator IDENTIFIER RIGHT\_PAREN LEFT\_BRACE body RIGHT\_BRACE

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE body RIGHT\_BRACE

ifStatement

**If-else statement**

|  |
| --- |
| **Code**  if (a > b)  {  return a;  } else  {  return b;  } |

**Parser derivation**

IF

IF LEFT\_PAREN

IF LEFT\_PAREN IDENTIFIER

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET

IF LEFT\_PAREN IDENTIFIER relationalOperator

IF LEFT\_PAREN IDENTIFIER relationalOperator IDENTIFIER

IF LEFT\_PAREN conditionalExpression

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE RETURN

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE RETURN IDENTIFIER

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE RETURN IDENTIFIER END\_STATEMENT

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE returnStatement

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE statement

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE body

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE body RIGHT\_BRACE

ifStatement

statement

body

body ELSE

body ELSE LEFT\_BRACE

body ELSE LEFT\_BRACE RETURN

body ELSE LEFT\_BRACE RETURN IDENTIFIER

body ELSE LEFT\_BRACE RETURN IDENTIFIER END\_STATEMENT

body ELSE LEFT\_BRACE returnStatement

body ELSE LEFT\_BRACE statement

body ELSE LEFT\_BRACE body

body ELSE LEFT\_BRACE body RIGHT\_BRACE

body elseStatement

body statement

body body

body

**If-else-if statement**

|  |
| --- |
| **Code**  if (a > b)  {  return a;  } else if (a == b)  {  return ‘a and b are equal’;  } else  {  return b;  } |

**Parser derivation**

IF

IF LEFT\_PAREN

IF LEFT\_PAREN IDENTIFIER

IF LEFT\_PAREN IDENTIFIER RIGHT\_ANGLED\_BRACKET

IF LEFT\_PAREN IDENTIFIER relationalOperator

IF LEFT\_PAREN IDENTIFIER relationalOperator IDENTIFIER

IF LEFT\_PAREN conditionalExpression

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE RETURN

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE RETURN IDENTIFIER

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE RETURN IDENTIFIER END\_STATEMENT

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE returnStatement

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE statement

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE body

IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE body RIGHT\_BRACE

ifStatement

statement

body

body ELSE

body ELSE IF

body ELSE IF LEFT\_PAREN

body ELSE IF LEFT\_PAREN IDENTIFIER

body ELSE IF LEFT\_PAREN IDENTIFIER REL\_EQUALITY

body ELSE IF LEFT\_PAREN IDENTIFIER relationalOperator

body ELSE IF LEFT\_PAREN IDENTIFIER relationalOperator IDENTIFIER

body ELSE IF LEFT\_PAREN conditionalExpression

body ELSE IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN

body ELSE IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE

body ELSE IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE RETURN

body ELSE IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE RETURN STRING\_LITERAL

body ELSE IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE RETURN value

body ELSE IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE RETURN value END\_STATEMENT

body ELSE IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE returnStatement

body ELSE IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE statement

body ELSE IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE body

body ELSE IF LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE body RIGHT\_BRACE

body ELSE ifStatement

body elseStatement

body statement

body body

body

body ELSE

body ELSE LEFT\_BRACE

body ELSE LEFT\_BRACE RETURN

body ELSE LEFT\_BRACE RETURN IDENTIFIER

body ELSE LEFT\_BRACE RETURN IDENTIFIER END\_STATEMENT

body ELSE LEFT\_BRACE returnStatement

body ELSE LEFT\_BRACE statement

body ELSE LEFT\_BRACE body

body ELSE LEFT\_BRACE body RIGHT\_BRACE

body elseStatement

body statement

body body

body

**Switch statement**

|  |
| --- |
| **Code**  switch n  {  case 1: return 'Monday';  case 2: return 'Tuesday';  case 3: return 'Wednesday';  } |

**Parser derivation**

SWITCH

SWITCH IDENTIFIER

SWITCH IDENTIFIER LEFT\_BRACE

SWITCH IDENTIFIER LEFT\_BRACE CASE

SWITCH IDENTIFIER LEFT\_BRACE CASE NUM\_LITERAL

SWITCH IDENTIFIER LEFT\_BRACE CASE value

SWITCH IDENTIFIER LEFT\_BRACE CASE value COLON

SWITCH IDENTIFIER LEFT\_BRACE CASE value COLON RETURN

SWITCH IDENTIFIER LEFT\_BRACE CASE value COLON RETURN STRING\_LITERAL

SWITCH IDENTIFIER LEFT\_BRACE CASE value COLON RETURN value

SWITCH IDENTIFIER LEFT\_BRACE CASE value COLON RETURN value END\_STATEMENT

SWITCH IDENTIFIER LEFT\_BRACE CASE value COLON returnStatement

SWITCH IDENTIFIER LEFT\_BRACE CASE value COLON statement

SWITCH IDENTIFIER LEFT\_BRACE caseStatement

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE NUM\_LITERAL

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON RETURN

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON RETURN STRING\_LITERAL

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON RETURN value

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON RETURN value END\_STATEMENT

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON returnStatement

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON statement

SWITCH IDENTIFIER LEFT\_BRACE caseStatement caseStatement

SWITCH IDENTIFIER LEFT\_BRACE caseStatement

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE NUM\_LITERAL

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON RETURN

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON RETURN STRING\_LITERAL

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON RETURN value

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON RETURN value END\_STATEMENT

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON returnStatement

SWITCH IDENTIFIER LEFT\_BRACE caseStatement CASE value COLON statement

SWITCH IDENTIFIER LEFT\_BRACE caseStatement caseStatement

SWITCH IDENTIFIER LEFT\_BRACE caseStatement

SWITCH IDENTIFIER LEFT\_BRACE caseStatement RIGHT\_BRACE

switchStatement

**Iterative statements**

**For statement**

|  |
| --- |
| **Code**  for int n in 5  {  Print(n);  } |

**Parser derivation**

FOR

FOR INT

FOR dataType

FOR dataType IDENTIFIER

FOR varDeclaration

FOR varDeclaration IN

FOR varDeclaration IN NUM\_LITERAL

FOR varDeclaration IN value

FOR varDeclaration IN value LEFT\_BRACE

FOR varDeclaration IN value LEFT\_BRACE IDENTIFIER

FOR varDeclaration IN value LEFT\_BRACE IDENTIFIER LEFT\_PAREN

FOR varDeclaration IN value LEFT\_BRACE IDENTIFIER LEFT\_PAREN IDENTIFIER

FOR varDeclaration IN value LEFT\_BRACE IDENTIFIER LEFT\_PAREN IDENTIFIER RIGHT\_PAREN

FOR varDeclaration IN value LEFT\_BRACE IDENTIFIER LEFT\_PAREN IDENTIFIER RIGHT\_PAREN END\_STATEMENT

FOR varDeclaration IN value LEFT\_BRACE functionCallStatement

FOR varDeclaration IN value LEFT\_BRACE statement

FOR varDeclaration IN value LEFT\_BRACE body

FOR varDeclaration IN value LEFT\_BRACE body RIGHT\_BRACE

forStatement

statement

body

**While statement**

|  |
| --- |
| **Code**  while (i < 5)  {  Print(i);  } |

**Parser derivation**

WHILE

WHILE LEFT\_PAREN

WHILE LEFT\_PAREN IDENTIFIER

WHILE LEFT\_PAREN IDENTIFIER LEFT\_ANGLED\_BRACKET

WHILE LEFT\_PAREN IDENTIFIER relationalOperator

WHILE LEFT\_PAREN IDENTIFIER relationalOperator NUM\_LITERAL

WHILE LEFT\_PAREN IDENTIFIER relationalOperator value

WHILE LEFT\_PAREN conditionalExpression

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE IDENTIFIER

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE IDENTIFIER LEFT\_PAREN

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE IDENTIFIER LEFT\_PAREN IDENTIFIER

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE IDENTIFIER LEFT\_PAREN IDENTIFIER RIGHT\_PAREN

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE IDENTIFIER LEFT\_PAREN IDENTIFIER RIGHT\_PAREN END\_STATEMENT

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE functionCallStatement

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE statement

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE body

WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN LEFT\_BRACE body RIGHT\_BRACE

whileStatement

statement

body

**Do-while statement**

|  |
| --- |
| **Code**  do  {  Print(i);  } while (i < 5) |

**Parser derivation**

DO

DO LEFT\_BRACE

DO LEFT\_BRACE IDENTIFIER

DO LEFT\_BRACE IDENTIFIER LEFT\_PAREN

DO LEFT\_BRACE IDENTIFIER LEFT\_PAREN IDENTIFIER

DO LEFT\_BRACE IDENTIFIER LEFT\_PAREN IDENTIFIER RIGHT\_PAREN

DO LEFT\_BRACE IDENTIFIER LEFT\_PAREN IDENTIFIER RIGHT\_PAREN END\_STATEMENT

DO LEFT\_BRACE functionCallStatement

DO LEFT\_BRACE statement

DO LEFT\_BRACE body

DO LEFT\_BRACE body RIGHT\_BRACE

DO LEFT\_BRACE body RIGHT\_BRACE WHILE

DO LEFT\_BRACE body RIGHT\_BRACE WHILE LEFT\_PAREN

DO LEFT\_BRACE body RIGHT\_BRACE WHILE LEFT\_PAREN IDENTIFIER

DO LEFT\_BRACE body RIGHT\_BRACE WHILE LEFT\_PAREN IDENTIFIER LEFT\_ANGLED\_BRACKET

DO LEFT\_BRACE body RIGHT\_BRACE WHILE LEFT\_PAREN IDENTIFIER relationalOperator

DO LEFT\_BRACE body RIGHT\_BRACE WHILE LEFT\_PAREN IDENTIFIER relationalOperator NUM\_LITERAL

DO LEFT\_BRACE body RIGHT\_BRACE WHILE LEFT\_PAREN IDENTIFIER relationalOperator value

DO LEFT\_BRACE body RIGHT\_BRACE WHILE LEFT\_PAREN conditionalExpression

DO LEFT\_BRACE body RIGHT\_BRACE WHILE LEFT\_PAREN conditionalExpression RIGHT\_PAREN

doWhileStatement

statement

body

**Math statements**

**Simple expression**

|  |
| --- |
| **Code**  √((x2 - x1)^2 + (y2 - y1)^2) |

**Parser derivation**

MATH\_SQRT

unaryMath

unaryMath LEFT\_PAREN

unaryMath LEFT\_PAREN LEFT\_PAREN

unaryMath LEFT\_PAREN LEFT\_PAREN IDENTIFIER

unaryMath LEFT\_PAREN LEFT\_PAREN IDENTIFIER OP\_SUB

unaryMath LEFT\_PAREN LEFT\_PAREN IDENTIFIER arithmeticOperator

unaryMath LEFT\_PAREN LEFT\_PAREN IDENTIFIER arithmeticOperator IDENTIFIER

unaryMath LEFT\_PAREN LEFT\_PAREN term

unaryMath LEFT\_PAREN LEFT\_PAREN expression

unaryMath LEFT\_PAREN LEFT\_PAREN expression RIGHT\_PAREN

unaryMath LEFT\_PAREN expression

unaryMath LEFT\_PAREN expression OP\_EXPO

unaryMath LEFT\_PAREN expression arithmeticOperator

unaryMath LEFT\_PAREN expression arithmeticOperator NUM\_LITERAL

unaryMath LEFT\_PAREN expression arithmeticOperator value

unaryMath LEFT\_PAREN expression

unaryMath LEFT\_PAREN expression OP\_ADD

unaryMath LEFT\_PAREN expression arithmeticOperator

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN IDENTIFIER

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN IDENTIFIER OP\_SUB

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN IDENTIFIER arithmeticOperator

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN IDENTIFIER arithmeticOperator IDENTIFIER

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN term

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN expression

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN expression RIGHT\_PAREN

unaryMath LEFT\_PAREN expression arithmeticOperator expression

unaryMath LEFT\_PAREN expression

unaryMath LEFT\_PAREN expression OP\_EXPO

unaryMath LEFT\_PAREN expression arithmeticOperator

unaryMath LEFT\_PAREN expression arithmeticOperator NUM\_LITERAL

unaryMath LEFT\_PAREN expression arithmeticOperator value

unaryMath LEFT\_PAREN expression

unaryMath LEFT\_PAREN expression RIGHT\_PAREN

unaryMath expression

expression

**Simple expression**

|  |
| --- |
| **Code**  √((x2 - x1)^2 + (y2 - y1)^2) |

**Parser derivation**

MATH\_SQRT

unaryMath

unaryMath LEFT\_PAREN

unaryMath LEFT\_PAREN LEFT\_PAREN

unaryMath LEFT\_PAREN LEFT\_PAREN IDENTIFIER

unaryMath LEFT\_PAREN LEFT\_PAREN IDENTIFIER OP\_SUB

unaryMath LEFT\_PAREN LEFT\_PAREN IDENTIFIER arithmeticOperator

unaryMath LEFT\_PAREN LEFT\_PAREN IDENTIFIER arithmeticOperator IDENTIFIER

unaryMath LEFT\_PAREN LEFT\_PAREN term

unaryMath LEFT\_PAREN LEFT\_PAREN expression

unaryMath LEFT\_PAREN LEFT\_PAREN expression RIGHT\_PAREN

unaryMath LEFT\_PAREN expression

unaryMath LEFT\_PAREN expression OP\_EXPO

unaryMath LEFT\_PAREN expression arithmeticOperator

unaryMath LEFT\_PAREN expression arithmeticOperator NUM\_LITERAL

unaryMath LEFT\_PAREN expression arithmeticOperator value

unaryMath LEFT\_PAREN expression

unaryMath LEFT\_PAREN expression OP\_ADD

unaryMath LEFT\_PAREN expression arithmeticOperator

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN IDENTIFIER

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN IDENTIFIER OP\_SUB

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN IDENTIFIER arithmeticOperator

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN IDENTIFIER arithmeticOperator IDENTIFIER

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN term

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN expression

unaryMath LEFT\_PAREN expression arithmeticOperator LEFT\_PAREN expression RIGHT\_PAREN

unaryMath LEFT\_PAREN expression arithmeticOperator expression

unaryMath LEFT\_PAREN expression

unaryMath LEFT\_PAREN expression OP\_EXPO

unaryMath LEFT\_PAREN expression arithmeticOperator

unaryMath LEFT\_PAREN expression arithmeticOperator NUM\_LITERAL

unaryMath LEFT\_PAREN expression arithmeticOperator value

unaryMath LEFT\_PAREN expression

unaryMath LEFT\_PAREN expression RIGHT\_PAREN

unaryMath expression

expression

[End of Documentation]