EduBASIC Language Reference

Table of Contents

Data Types, Variables, Arithmetic, and Boolean Operations

Data Types

EduBASIC provides four built-in scalar data types:

- **Integer** (32-bit signed integer)
- **Real** (64-bit floating-point number)
- **Complex** (128-bit complex number with real and imaginary parts)

- **String** (text data)

Additionally, **structures** can be created to group related data together. Structures are untyped identifier-value dictionaries that allow you to store multiple values under named members.

Type Sigils

EduBASIC uses type sigils (special characters) at the end of variable names to indicate the variable's type. Sigils are **required** for all variables and apply **only** to variable identifiers—they are never used in literals.

Data Type	Sigil	Example Variable	Size
Integer	`%`	`count%`	32-bit
Real	`#`	`value#`	64-bit
String	`$`	`name$`	Variable
Complex	`&`	`z&`	128-bit
Structure	*(none)*	`player`	Variable

Rules:

- Sigils are suffixes (appear at the end of the variable name)
- All variables must have a sigil (except structures)
- Literals never use sigils
- Variable names are case-insensitive

Variable Names

Variable names must:

- Be alphanumeric (letters and numbers)
- End with the appropriate type sigil (except structures)
- Have a maximum length of 64 characters
- Start with a letter

Examples:

```
LET studentCount% = 10
LET roomTemperature# = 98.6
LET playerName$ = "Alice"
LET impedance& = 3+4i
LET player = { name$: "Bob", score%: 100 }      ' Structure (no sigil)
```

Variable Declaration

Variables in EduBASIC are **implicitly declared**—you simply assign a value with the `LET` or `LOCAL` statements to create a variable. **No variables need to be declared ahead of time**, whether they are scalars, structures, or arrays.

The `DIM` statement is used **only** for resizing arrays (changing the size of an array). Undeclared arrays are presumed to have size 0, so you can use `DIM` to resize them even if they haven't been assigned yet. All other variables are created automatically when first assigned with `LET` or `LOCAL` .

LET Statement

The `LET` statement assigns a value to a variable, creating module-level (global) variables.

Syntax:

```
LET variable = expression
LET array[] = expression
```

Rules:

- Creates module-level (global) variables that persist throughout the program
- Can assign to scalars, structures, or arrays
- For arrays, the expression can be an array literal, another array, or an array slice
- The `LET` keyword is optional in some contexts but recommended for clarity
- Variables are created automatically on first assignment—no declaration needed
- Type is determined by the variable's sigil (or lack of sigil for structures)

Examples:

```
LET count% = 10
LET name$ = "Alice"
LET result# = 3.14159
LET complex& = 3+4i

' Array assignments
LET numbers%[] = [1, 2, 3, 4, 5]
LET copy%[] = numbers%[]
LET slice%[] = numbers%[2 TO 4]

' Structure assignment
LET player = { name$: "Bob", score%: 100 }
```

SWAP Statement

The `SWAP` statement exchanges the values of two variables of the same type.

Syntax:

```
SWAP variable1 WITH variable2
SWAP array1[] WITH array2[]
```

Rules:

- Exchanges the values of two variables
- Both operands must be of the same type (both integers, both reals, both strings, both arrays, etc.)
- Can swap scalars or arrays
- Arrays must be of the same element type
- The operation is atomic (both values are swapped simultaneously)

Examples:

```
LET x% = 5
LET y% = 10
SWAP x% WITH y%
PRINT "x: ", x%, " y: ", y%    ' Prints: x: 10 y: 5

LET arr1%[] = [1, 2, 3]
LET arr2%[] = [4, 5, 6]
SWAP arr1%[] WITH arr2%[]
PRINT arr1%[]    ' Prints: 4, 5, 6
PRINT arr2%[]    ' Prints: 1, 2, 3

' Swap strings
LET a$ = "Hello"
LET b$ = "World"
SWAP a$ WITH b$
PRINT a$, " ", b$    ' Prints: World Hello
```

Default Values

When a variable is first created (before any assignment), it has a default value based on its type:

Type	Default Value	Notes
Integer (`%`)	`0`	32-bit signed integer
Real (`#`)	`0.0`	64-bit floating-point
Complex (`&`)	`0+0i`	128-bit complex number (real and imaginary parts both 0)
String (`$`)	`""`	Empty string
Structure (no sigil)	`{ }`	Empty structure (no members)
Array (`[]`)	`[ ]`	Empty array (size 0)

Examples:

```
LET count%          ' count% = 0 (default)
LET value#          ' value# = 0.0 (default)
LET z&              ' z& = 0+0i (default)
LET name$           ' name$ = "" (default)
LET player          ' player = { } (default, empty structure)
LET numbers%[]      ' numbers%[] = [ ] (default, empty array, size 0)
```

Important Notes:

- Default values apply when a variable is first referenced before any assignment
- Arrays default to size 0 (empty array)
- Structures default to an empty structure with no members
- You can use `DIM` to resize an array even if it hasn't been assigned yet (it's treated as size 0)

Literals

Integer Literals

Integer literals can be written in three forms:

Decimal:

```
123
0
9001
```

Hexadecimal (prefix with `&H`):

```
&HFF
&H7F2A
&H00FF
```

Binary (prefix with `&B`):

```
&B101101
&B1101_0011
&B11111111
```

Underscores can be used in binary literals for readability.

Real Literals

Real (floating-point) numbers can be written as:

Decimal notation:

```
3.14
10.
.25
42
```

Scientific notation:

```
1E6
3.2E-4
6.022E23
1.5e+10
```

The exponent uses `E` or `e` followed by an optional sign (`+` or `-`).

Complex Literals

Complex numbers consist of a real part and an imaginary part.

Imaginary-only (real part is zero):

```
4i
3.14i
.25i
1E-3i
```

Full complex numbers:

```
3+4i
3-4i
10.5+2.5E-3i
1E3-2i
.5+.25i
```

Notes:

- The imaginary unit can be lowercase `i` or uppercase `I`
- No spaces are required between the real and imaginary parts
- The `+` or `-` immediately before the imaginary term distinguishes it from subtraction in expressions

String Literals

String literals are enclosed in double quotes:

```
"Hello, world!"
"EduBASIC"
"Line 1\nLine 2"
```

Array Literals

Array literals are written using square brackets with comma-separated values:

```
[1, 2, 3, 4, 5]
["Alice", "Bob", "Charlie"]
[1.5, 2.7, 3.14]
[ ]
```

Rules:

- Array literals use square brackets `[]`
- Elements are separated by commas
- Elements can be any data type (Integer, Real, Complex, String)

- Empty array literal `[ ]` creates an array with size 0
- Array literals create arrays starting at index 1 (one-based)
- All elements in an array literal must be of compatible types (type coercion applies)

Examples:

```
LET numbers%[] = [1, 2, 3, 4, 5]              ' Integer array
LET names$[] = ["Alice", "Bob", "Charlie"]    ' String array
LET mixed#[] = [1.5, 2.7, 3.14]               ' Real array
LET empty%[] = [ ]                            ' Empty array (size 0)
LET complex&[] = [1+2i, 3+4i, 5+6i]           ' Complex array
```

Structure Literals

Structure literals are written using curly braces with comma-separated key-value pairs:

```
{ name$: "Alice", score%: 100, level%: 5 }
{ x%: 100, y%: 200 }
{ }
```

Rules:

- Structure literals use curly braces `{ }`
- Members are specified as `key: value` pairs
- Pairs are separated by commas
- Keys are member names (without type sigils in the literal)
- Values can be any data type (Integer, Real, Complex, String, Array, Structure)
- Empty structure literal `{ }` creates a structure with no members
- Structure literals can contain nested structures and arrays
- Member names in literals do not use type sigils (the type sigil is part of the member name when accessing)

Examples:

```
LET player = { name$: "Alice", score%: 100, level%: 5 }
LET point = { x%: 100, y%: 200 }
LET person = { firstName$: "John", lastName$: "Doe", age%: 30, email$:
"john@example.com" }
LET config = { width%: 640, height%: 480, title$: "My Game", fullscreen%: FALSE%
}
LET empty = { }                                   ' Empty structure (no members)
```

```
' Structure with array members
LET player = { name$: "Alice", scores%[]: [100, 95, 87, 92], inventory$[]:
["sword", "shield", "potion"] }

' Structure with nested structures
LET person = { name: { first$: "John", last$: "Doe" }, address: { street$: "123
Main St", city$: "Springfield", zip%: 12345 } }

' Structure with both arrays and nested structures
LET game = { player: { name$: "Hero", stats: { hp%: 100, mp%: 50 }, items$[]:
["sword", "shield"] }, enemies%[]: [10, 15, 20] }
```

Type Coercion

EduBASIC automatically converts between numeric types when they are mixed in expressions or assigned to variables. Type coercion follows a hierarchy: **Integer → Real → Complex**.

Upcasting (Type Promotion)

Upcasting occurs automatically when numeric types are mixed in expressions. The result type is always the highest type in the hierarchy:

- **Integer + Integer** → Integer
- **Integer + Real** → Real
- **Integer + Complex** → Complex
- **Real + Real** → Real
- **Real + Complex** → Complex
- **Complex + Complex** → Complex

Examples:

```
LET a% = 5
LET b# = 3.14
LET c& = 2+3i

LET result1# = a% + b#        ' Integer + Real → Real (8.14)
LET result2& = a% + c&         ' Integer + Complex → Complex (7+3i)
LET result3& = b# + c&        ' Real + Complex → Complex (5.14+3i)
LET result4& = c& + c&         ' Complex + Complex → Complex (4+6i)
```

Arithmetic Operations:

- All arithmetic operations (`+` , `-` , `*` , `/` , `^` , `MOD`) follow the same promotion rules
- Division (`/`) always produces a Real result, even when dividing two integers

- Modulo (`MOD`) works with both Integer and Real operands, producing a Real result when either operand is Real
- Exponentiation (`^`) follows normal promotion rules

Examples:

```
LET quotient# = 15 / 4         ' Integer / Integer → Real (3.75)
LET remainder% = 17 MOD 5      ' Integer MOD Integer → Integer (2)
LET remainder# = 17.5 MOD 5     ' Real MOD Integer → Real (2.5)
LET remainder# = 17 MOD 5.5    ' Integer MOD Real → Real (0.5)
LET remainder# = 17.5 MOD 5.5   ' Real MOD Real → Real (1.0)
LET power# = 2 ^ 8             ' Integer ^ Integer → Real (256.0)
LET mixed# = 5 + 3.14          ' Integer + Real → Real (8.14)
LET complex& = 5 + 3i           ' Integer + Complex → Complex (5+3i)
```

Downcasting (Type Demotion)

Downcasting occurs when assigning a value to a variable with a lower type in the hierarchy. Downcasting may result in loss of precision or information:

- **Real → Integer**: Truncates the decimal part (rounds toward zero)
- **Complex → Real**: Extracts only the real part, discards imaginary part
- **Complex → Integer**: Extracts real part and truncates to integer

Examples:

```
LET realValue# = 3.9
LET intValue% = realValue#     ' Real → Integer: 3 (truncated, not rounded)

LET complexValue& = 5.7+2.3i
LET realResult# = complexValue&    ' Complex → Real: 5.7 (imaginary part lost)
LET intResult% = complexValue&      ' Complex → Integer: 5 (real part truncated)

LET anotherComplex& = 3.14+0i
LET intFromComplex% = anotherComplex&    ' Complex → Integer: 3
```

Important Notes:

- Downcasting to Integer truncates (rounds toward zero), it does not round
- Downcasting from Complex to Real or Integer loses the imaginary component
- No automatic downcasting occurs in expressions—only in assignments

Assignment Coercion

When assigning a value to a variable, the value is coerced to match the variable's type:

```
LET integerVar% = 42          ' Integer → Integer (no change)
LET integerVar% = 42.7        ' Real → Integer: 42 (truncated)
LET integerVar% = 42.7+0i     ' Complex → Integer: 42 (real part, truncated)

LET realVar# = 42             ' Integer → Real: 42.0 (promoted)
LET realVar# = 42.7           ' Real → Real (no change)
LET realVar# = 42.7+3i        ' Complex → Real: 42.7 (imaginary part lost)

LET complexVar& = 42           ' Integer → Complex: 42+0i (promoted)
LET complexVar& = 42.7        ' Real → Complex: 42.7+0i (promoted)
LET complexVar& = 42.7+3i     ' Complex → Complex (no change)
```

Literal Type Inference

Literal types are inferred from their syntax:

- Numbers without decimal point or exponent → Integer
- Numbers with decimal point or exponent → Real
- Numbers with `i` or `I` suffix → Complex

Examples:

```
LET a% = 42           ' Integer literal
LET b# = 42.0         ' Real literal
LET c# = 42           ' Integer literal coerced to Real
LET d& = 42+0i        ' Complex literal
LET e& = 3.14+2i      ' Complex literal
```

Summary

Arithmetic Operations:

Operation	Result Type	Notes
Integer op Integer	Integer	Except division (`/`) which yields Real; MOD yields Integer
Integer op Real	Real	Integer promoted to Real
Integer op Complex	Complex	Integer promoted to Complex

Operation	Result Type	Notes
Real op Real	Real	MOD yields Real
Real op Complex	Complex	Real promoted to Complex
Complex op Complex	Complex	MOD not applicable to complex numbers

Mathematical Operators (Trigonometric, Hyperbolic, Exponential, Logarithmic, Roots):

Operator Type	Operand Type	Result Type	Notes
SIN, COS, TAN, ASIN, ACOS, ATAN	Integer/Real	Real	Standard trigonometric functions
SIN, COS, TAN, ASIN, ACOS, ATAN	Complex	Complex	Complex extensions of trigonometric functions
SINH, COSH, TANH, ASINH, ACOSH, ATANH	Integer/Real	Real	Standard hyperbolic functions
SINH, COSH, TANH, ASINH, ACOSH, ATANH	Complex	Complex	Complex extensions of hyperbolic functions
EXP, LOG, LOG10, LOG2	Integer/Real	Real	Standard exponential and logarithmic functions
EXP, LOG, LOG10, LOG2	Complex	Complex	Complex extensions of exponential and logarithmic functions
SQRT, CBRT	Integer/Real	Real	Standard root functions
SQRT, CBRT	Complex	Complex	Complex extensions of root functions
ROUND, FLOOR, CEIL, TRUNC, EXPAND	Integer/Real	Real	Rounding and truncation functions
ROUND, FLOOR, CEIL, TRUNC, EXPAND	Complex	Error	Not applicable to complex numbers
SGN	Integer/Real	Integer	Sign function

Operator Type	Operand Type	Result Type	Notes
SGN	Complex	Error	Not applicable to complex numbers

Assignment Coercion:

Assignment	Result Type	Notes
Assign Real to Integer	Integer	Truncated (not rounded)
Assign Complex to Real	Real	Imaginary part lost
Assign Complex to Integer	Integer	Real part truncated, imaginary lost

Structures

Structures are untyped identifier-value dictionaries that allow you to group related data together. Structure variables do not use type sigils and are created automatically when first assigned.

Structure Creation

Structures can be created in two ways:

1. Using structure literals (recommended):

```
LET player = { name$: "Alice", score%: 100, level%: 5 }
```

2. By assigning values to members individually using associative array syntax:

```
LET player[name$] = "Alice"
LET player[score%] = 100
LET player[level%] = 5
```

Both methods are equivalent. Structure literals are more concise for creating structures with multiple members, while individual assignment is useful for updating specific members or building structures incrementally.

Member Access

Structure members are accessed using associative array syntax:

```
PRINT player[name$]    ' Prints: Alice
LET player[score%] += 10
IF player[level%] > 10 THEN PRINT "Advanced player"
```

Structure Examples

```
' Create a structure for a point
LET point = { x%: 100, y%: 200 }

' Create a structure for a person
LET person = { firstName$: "John", lastName$: "Doe", age%: 30, email$:
"john@example.com" }

' Access structure members
PRINT person[firstName$], " ", person[lastName$]
LET person[age%] += 1

' Structures can contain any data type
LET config = { width%: 640, height%: 480, title$: "My Game", fullscreen%: FALSE%
}
```

Nested Structures and Arrays

Structure members can themselves be arrays or structures, allowing for nested data structures:

```
' Structure with array members
LET player = { name$: "Alice", scores%[]: [100, 95, 87, 92], inventory$[]:
["sword", "shield", "potion"] }

' Structure with nested structures
LET person = { name: { first$: "John", last$: "Doe" }, address: { street$: "123
Main St", city$: "Springfield", zip%: 12345 } }

' Structure with both arrays and nested structures
LET game = { player: { name$: "Hero", stats: { hp%: 100, mp%: 50 }, items$[]:
["sword", "shield"] }, enemies%[]: [10, 15, 20] }

' Access nested structure members
PRINT person[name][first$], " ", person[name][last$]
PRINT person[address][street$]

' Access array members within structures
```

```
PRINT player[scores%][1]    ' First score
LET player[scores%][2] = 98  ' Update second score
```

Structure Comparison

Structures support the equality operator (=). Two structures are equal if all of their members are equal (recursively, including nested structures and array members).

```
LET point1 = { x%: 100, y%: 200 }
LET point2 = { x%: 100, y%: 200 }

IF point1 = point2 THEN PRINT "Points are equal"    ' TRUE%

LET point3 = { x%: 100, y%: 201 }

IF point1 = point3 THEN PRINT "Equal"    ' FALSE% (y values differ)
```

Important Notes:

- The equality operator (`=`) is defined for structures
- Two structures are equal if all their members are equal (including nested structures and arrays)
- Inequality operators (`<>` , `<` , `>` , `<=` , `>=`) are **not** defined for structures
- Structure variables do not use type sigils (no `%` , `#` , `$` , or `&`)
- Structures are untyped—members can hold any data type
- Structure members can be arrays (using `[]` syntax)
- Structure members can be other structures (nested structures)
- Members are created automatically when first assigned
- No declaration is needed before using a structure

Arrays

Arrays are **one-based by default** (index 1 is the first element). Arrays in EduBASIC are backed by JavaScript arrays, providing full expressiveness while maintaining BASIC syntax.

Important: When referring to an entire array (not a single element), you must use the `[]` syntax. For example, `numbers%[]` refers to the entire array, while `numbers%` (without `[]`) would refer to a scalar variable. This distinction is required because `numbers%` could be either a scalar or an array name, so `[]` is necessary to unambiguously refer to the entire array.

Array Creation

Arrays are created automatically when first assigned. No declaration is needed:

```
LET numbers%[] = [1, 2, 3, 4, 5]
LET names$[] = ["Alice", "Bob", "Charlie"]
```

Important: Undeclared arrays (arrays that haven't been assigned yet) are presumed to have size 0. You can assign to them or resize them with `DIM` before use.

Array Resizing

The `DIM` statement is used **only** for resizing arrays. Undeclared arrays are presumed to have size 0, so you can use `DIM` to resize them even if they haven't been assigned yet.

Syntax:

```
DIM arrayName[size]
DIM arrayName[start TO end]
DIM arrayName[size1, size2, ...]
DIM arrayName[start1 TO end1, start2 TO end2, ...]
```

Rules:

- Arrays are one-based by default (index 1 is the first element)
- `DIM` resizes an array to the specified size or range
- If the array doesn't exist, `DIM` creates it
- If the array already exists, `DIM` resizes it (existing elements may be preserved or cleared depending on the new size)
- For multi-dimensional arrays, specify dimensions separated by commas
- Custom ranges use `start TO end` syntax

Resize to a new size:

```
DIM numbers%[10]    ' Resize to 10 elements (indices 1-10), creates array if it doesn't exist
LET numbers%[] = [1, 2, 3]
DIM numbers%[20]    ' Resize existing array to 20 elements
```

Resize with custom range:

```
DIM data%[0 TO 11]    ' Resize to indices 0-11, creates array if it doesn't exist
```

Multi-dimensional arrays:

```
DIM matrix#[5, 10]    ' Resize to 5×10 matrix, creates array if it doesn't exist
DIM grid%[1 TO 10, 1 TO 20]    ' Resize with custom ranges
```

Array Literals

Arrays can be initialized using array literals with square brackets:

```
LET numbers%[] = [1, 2, 3, 4, 5]
LET names$[] = ["Alice", "Bob", "Charlie"]
LET mixed#[] = [1.5, 2.7, 3.14]
LET empty%[] = [ ]
```

Array literals create arrays starting at index 1 (one-based).

Array Indexing

Arrays are accessed using square brackets:

```
LET value% = numbers%[3]
LET numbers%[5] = 100
LET name$ = names$[2]
```

Arrays support multi-dimensional indexing:

```
LET value# = matrix#[2, 3]
LET matrix#[1, 5] = 42.5
```

Array Slicing

Arrays support slicing similar to string slicing:

```
LET sub%[] = numbers%[2 TO 5]        ' Elements 2 through 5
LET tail%[] = numbers%[3 TO ...]     ' Elements 3 to end
LET head%[] = numbers%[... TO 5]     ' Elements 1 to 5
LET all%[] = numbers%[... TO ...]    ' Entire array
```

Slicing creates a new array containing the specified range of elements.

Array Concatenation

Arrays can be concatenated using the `+` operator:

```
LET combined%[] = numbers%[] + more%[]
LET all%[] = [1, 2] + numbers%[] + [9, 10]
```

Concatenation creates a new array containing all elements from the left array followed by all elements from the right array.

Array Length

Use the `||` operator to get the length of an array:

```
LET size% = | numbers%[] |
```

Returns the number of elements in the array (the highest index for one-based arrays). The `||` operator works for arrays, reals (absolute value), and complex numbers (norm). Note that spaces are required inside the `||` operator.

Array Manipulation Statements

PUSH Statement:

Adds an element to the end of an array.

```
PUSH numbers%[], 10
PUSH names$[], "David"
```

POP Statement:

Removes the last element from an array and assigns it to a variable.

```
POP numbers%[] INTO value%
POP names$[] INTO name$
```

SHIFT Statement:

Removes the first element from an array and assigns it to a variable.

```
SHIFT numbers%[] INTO value%
SHIFT names$[] INTO name$
```

UNSHIFT Statement:

Adds an element to the beginning of an array.

```
UNSHIFT numbers%[], 0
UNSHIFT names$[], "Alice"
```

Array Search Operators

Array search operations are implemented as binary operators:

FIND Operator:

Finds the first element in an array that matches a value. Returns the element value if found, or 0 (for numeric arrays) or "" (for string arrays) if not found.

```
LET found% = numbers%[] FIND 5
LET found$ = names$[] FIND "Bob"
```

INDEXOF Operator:

Finds the index of the first occurrence of a value in an array. Returns 0 if not found.

```
LET index% = numbers%[] INDEXOF 5
LET index% = names$[] INDEXOF "Bob"
```

INCLUDES Operator:

Checks if an array includes a value. Returns TRUE% (-1) if found, FALSE% (0) if not found.

```
IF numbers%[] INCLUDES 5 THEN PRINT "Found"
IF names$[] INCLUDES "Bob" THEN PRINT "Found"
```

Array Operators

REVERSE Operator:

Returns a new array with elements in reverse order.

```
LET reversed%[] = REVERSE numbers%[]
```

JOIN Operator:

Joins array elements into a string with a separator.

```
LET joined$ = names$[] JOIN ", "    ' "Alice, Bob, Charlie"
```

Array Manipulation with Operators:

Array manipulation can be accomplished using array slicing and concatenation with `LET` statements:

Remove elements:

```
' Remove 3 elements starting at index 2 (removes indices 2, 3, 4)
LET numbers%[] = numbers%[1 TO 1] + numbers%[5 TO ...]
```

Insert elements:

```
' Insert [10, 20] at index 2 (existing elements from index 2 shift right)
LET numbers%[] = numbers%[1 TO 1] + [10, 20] + numbers%[2 TO ...]
```

Replace elements:

```
' Replace 3 elements at index 2 with [10, 20] (replaces indices 2, 3, 4)
LET numbers%[] = numbers%[1 TO 1] + [10, 20] + numbers%[5 TO ...]
```

Arithmetic Operations

Arithmetic Operators

Operator	Description	Example
`+`	Addition	`LET totalSum% = 5 + 3`
`-`	Subtraction	`LET difference% = 10 - 4`
`*`	Multiplication	`LET product% = 6 * 7`
`/`	Division	`LET quotient# = 15 / 4`
`MOD`	Modulo	`LET remainder% = 17 MOD 5`
`^` or `**`	Exponentiation	`LET powerResult# = 2 ^ 8`
`!`	Factorial	`LET factorialNum% = 5!`

Assignment Operators

The `LET` keyword is **required** for all assignments. EduBASIC also supports C-style compound assignment operators:

```
LET totalCount% = 10
LET totalCount% += 5
LET totalCount% -= 3
LET totalCount% *= 2
LET totalCount% /= 4
LET totalCount% ^= 2
```

Swapping Variables

The `SWAP` command exchanges the values of two variables. Both variables must be of the same type. Can swap scalars or arrays.

Syntax:

```
SWAP variable1 WITH variable2
SWAP array1[] WITH array2[]
```

Example:

```
LET x% = 5
LET y% = 10
SWAP x% WITH y%
PRINT "x: ", x%, " y: ", y%        ' Prints: x: 10 y: 5

LET firstName$ = "John"
LET lastName$ = "Doe"
SWAP firstName$ WITH lastName$
PRINT firstName$, " ", lastName$    ' Prints: Doe John

LET arr1%[] = [1, 2, 3]
LET arr2%[] = [4, 5, 6]
SWAP arr1%[] WITH arr2%[]
PRINT arr1%[]    ' Prints: 4, 5, 6
PRINT arr2%[]    ' Prints: 1, 2, 3
```

Unit Conversion Operators

EduBASIC provides postfix operators to convert between degrees and radians:

Operator	Description	Example
`DEG`	Convert radians to degrees	`LET degrees# = (3.14159 / 2) DEG`
`RAD`	Convert degrees to radians	`LET radians# = 90 RAD`

Example usage with trigonometric functions:

```
LET angle# = 45 RAD
LET result# = SIN angle#
LET angleDegrees# = ASIN result# DEG
```

Absolute Value / Norm / Array Length

Use vertical bars `||` for absolute value (for reals), norm (for complex numbers), or array length (for arrays). Spaces are required inside the `||` operator:

```
LET absoluteValue# = | -5 |
LET complexNorm# = | 3+4i |
LET arrayLength% = | numbers%[] |
LET stringLength% = | text$ |
```

Arithmetic Operators and Mathematical Functions

EduBASIC provides a comprehensive set of arithmetic operators and mathematical functions:

Trigonometric Functions:

Function	Description	Example
`SIN x#`	Sine of x (radians)	`LET result# = SIN angle#`
`COS x#`	Cosine of x (radians)	`LET result# = COS angle#`
`TAN x#`	Tangent of x (radians)	`LET result# = TAN angle#`
`ASIN x#`	Arcsine of x (returns radians)	`LET angle# = ASIN ratio#`
`ACOS x#`	Arccosine of x (returns radians)	`LET angle# = ACOS ratio#`
`ATAN x#`	Arctangent of x (returns radians)	`LET angle# = ATAN slope#`

Hyperbolic Functions:

Function	Description	Example
`SINH x#`	Hyperbolic sine of x	`LET result# = SINH value#`
`COSH x#`	Hyperbolic cosine of x	`LET result# = COSH value#`
`TANH x#`	Hyperbolic tangent of x	`LET result# = TANH value#`

Function	Description	Example
`ASINH x#`	Inverse hyperbolic sine of x	`LET result# = ASINH value#`
`ACOSH x#`	Inverse hyperbolic cosine of x	`LET result# = ACOSH value#`
`ATANH x#`	Inverse hyperbolic tangent of x	`LET result# = ATANH value#`

Exponential and Logarithmic Functions:

Function	Description	Example
`EXP x#`	e raised to the power x	`LET result# = EXP power#`
`LOG x#`	Natural logarithm (base e)	`LET result# = LOG value#`
`LOG10 x#`	Common logarithm (base 10)	`LET result# = LOG10 value#`
`LOG2 x#`	Binary logarithm (base 2)	`LET result# = LOG2 value#`

Root Functions:

Function	Description	Example
`SQRT x#`	Square root of x	`LET result# = SQRT number#`
`CBRT x#`	Cube root of x	`LET result# = CBRT number#`

Rounding and Truncation Operators:

Function	Description	Example
`ROUND x#`	Round to nearest integer (ties round up)	`LET result# = ROUND 3.5` returns 4
`FLOOR x#`	Round toward negative infinity (−∞)	`LET result# = FLOOR 3.7` returns 3, `FLOOR -3.2` returns -4
`CEIL x#`	Round toward positive infinity (+∞)	`LET result# = CEIL 3.2` returns 4, `CEIL -3.7` returns -3
`TRUNC x#`	Round toward zero	`LET result# = TRUNC 3.9` returns 3, `TRUNC -3.9` returns -3
`EXPAND x#`	Round away from zero	`LET result# = EXPAND 3.1` returns 4, `EXPAND -3.1` returns -4

Sign Operator:

Function	Description	Example
`SGN x#`	Sign of x: -1, 0, or 1	`LET result% = SGN value#`

Complex Number Functions:

Function	Description	Example
`REAL z&`	Real part of complex number	`LET realPart# = REAL z&`
`IMAG z&`	Imaginary part of complex number	`LET imagPart# = IMAG z&`
`CONJ z&`	Complex conjugate	`LET conjugate& = CONJ z&`
`CABS z&`	Absolute value (magnitude)	`LET magnitude# = CABS z&`
`CARG z&`	Argument (phase angle) in radians	`LET phase# = CARG z&`
`CSQRT z&`	Complex square root	`LET result& = CSQRT z&`

Random Number Generation:

Function	Description	Example
`RND#`	Random real in range [0, 1)	`LET random# = RND#`

Boolean Operations

EduBASIC uses integers to represent boolean values:

- `0` represents **false**
- Any non-zero value (typically `1`) represents **true**

There is no separate boolean data type.

Boolean Operators

Operator	Description
`AND`	Output bit is 1 when both input bits are 1
`OR`	Output bit is 1 when at least one input bit is 1
`NOT`	Output bit is 1 when the input bit is 0

Operator	Description
`XOR`	Output bit is 1 when the input bits are different
`NAND`	Output bit is 1 when at least one input bit is 0
`NOR`	Output bit is 1 when both input bits are 0
`XNOR`	Output bit is 1 when both input bits are the same
`IMP`	Output bit is 1 when first bit is 0 or second bit is 1

Since there is no separate boolean data type, all boolean operators are **bitwise operators** that work on integers.

Boolean Constants:

- `FALSE% = 0`
- `TRUE% = -1`

Any non-zero value is treated as true in conditional expressions, but the canonical `TRUE%` value is `-1`. This is because `-1` in binary representation has all bits set to 1 (two's complement), which makes bitwise operations behave correctly. For example, `TRUE% AND TRUE%` yields `TRUE%` (`-1 AND -1 = -1`), while if `TRUE%` were `1`, then `1 AND 1 = 1` would only preserve the least significant bit.

Comparison Operations

Standard comparison operators are available for all data types, including arrays:

Operator	Description	Returns TRUE% (-1) when...
`=`	Equal to	Both operands have the same value
`<>`	Not equal to	Operands have different values
`<`	Less than	Left operand is smaller than right operand
`>`	Greater than	Left operand is larger than right operand
`<=`	Less than or equal	Left operand is smaller or equal to right
`>=`	Greater than or equal	Left operand is larger or equal to right

Comparison operations return integer values: `FALSE%` (0) or `TRUE%` (-1).

For arrays, comparison operators work element-wise:

- Arrays are equal (`=`) if they have the same length and all corresponding elements are equal
- Arrays are not equal (`<>`) if they have different lengths or any corresponding elements differ
- Ordering operators (`<` , `>` , `<=` , `>=`) compare arrays lexicographically (element by element)
- If arrays have different lengths, the shorter array is considered less than the longer array
- If arrays have the same length, comparison proceeds element by element until a difference is found

Examples:

```
LET arr1%[] = [1, 2, 3]
LET arr2%[] = [1, 2, 3]
LET arr3%[] = [1, 2, 4]
LET arr4%[] = [1, 2]

IF arr1%[] = arr2%[] THEN PRINT "Equal"    ' TRUE%
IF arr1%[] <> arr3%[] THEN PRINT "Different"    ' TRUE%
IF arr1%[] < arr3%[] THEN PRINT "Less"    ' TRUE% (3 < 4)
IF arr4%[] < arr1%[] THEN PRINT "Shorter is less"    ' TRUE% (shorter array)
```

Operator Precedence

EduBASIC follows standard mathematical operator precedence:

1. **Constants**: `RND#` , `INKEY$` , `PI#` , `E#` , `DATE$` , `TIME$` , `NOW%` , `TRUE%` , `FALSE%` (highest precedence)
2. Parentheses `()`
3. **Prefix operators**: `SIN` , `COS` , `TAN` , `ASIN` , `ACOS` , `ATAN` , `SINH` , `COSH` , `TANH` , `ASINH` , `ACOSH` , `ATANH` , `EXP` , `LOG` , `LOG10` , `LOG2` , `SQRT` , `CBRT` , `FLOOR` , `CEIL` , `ROUND` , `TRUNC` , `EXPAND` , `SGN` , `REAL` , `IMAG` , `CONJ` , `CABS` , `CARG` , `CSQRT` , `INT` , `ASC` , `CHR` , `STR` , `VAL` , `HEX` , `BIN` , `UCASE` , `LCASE` , `LTRIM` , `RTRIM` , `TRIM` , `REVERSE` , `EOF` , `LOC` , `NOTES`
4. **Postfix operators**: `!` (factorial), `DEG` , `RAD`
5. Absolute value / norm / array length / string length `| |`
6. Unary `+` and `-`
7. Exponentiation `^` or `**` (right-associative)
8. Multiplication `*` , Division `/` , and Modulo `MOD`
9. Addition `+` and Subtraction `-` (also array concatenation)

10. Array search operators: `FIND` , `INDEXOF` , `INCLUDES`
11. String/Array operators: `INSTR` , `JOIN` , `REPLACE` , `LEFT` , `RIGHT` , `MID`
12. Comparison operators (`=` , `<>` , `<` , `>` , `<=` , `>=`)
13. `NOT` (unary logical)
14. **Logical AND**: `AND` , `NAND`
15. **Logical OR**: `OR` , `NOR`
16. **Logical XOR**: `XOR` , `XNOR`
17. **Logical IMP**: `IMP` (lowest precedence)

When operators have the same precedence, evaluation proceeds left to right, except for exponentiation which is right-associative.

Random Number Generation

EduBASIC provides the `RND#` operator to generate random numbers and the `RANDOMIZE` command to seed the random number generator.

Random Number Operator:

- `RND#` - Returns a random real number in the range [0, 1)

Random Number Generator Seed:

The `RANDOMIZE` statement initializes the random number generator with a seed value.

Syntax:

```
RANDOMIZE
RANDOMIZE seedValue%
```

Rules:

- Without an argument, seeds the generator with the current Unix timestamp (`NOW%`)
- With an argument, seeds the generator with the specified integer value
- Using the same seed value produces the same sequence of random numbers
- Different seeds produce different random sequences

Examples:

```
RANDOMIZE     ' Seed with current timestamp (different each run)
RANDOMIZE 12345     ' Seed with specific value (reproducible)
RANDOMIZE NOW%    ' Explicitly seed with current timestamp
```

Mathematical Constants:

- `PI#` - Returns the mathematical constant π (pi) as a real number (approximately 3.141592653589793)
- `E#` - Returns the mathematical constant e (Euler's number) as a real number (approximately 2.718281828459045)

Examples:

```
RANDOMIZE
LET randomNumber# = RND#

RANDOMIZE 12345
LET dice% = INT (RND# * 6) + 1

LET randomInRange# = RND# * 100

' Seed with current time
RANDOMIZE NOW%
LET currentTime% = NOW%

' Mathematical constants
LET circumference# = 2 * PI# * radius#
LET area# = PI# * radius# * radius#
LET exponential# = E# ^ power#
```

Control Flow

EduBASIC supports both structured programming constructs and traditional BASIC flow control. While `GOTO` and `GOSUB` are available for backwards compatibility and educational purposes, modern structured programming approaches are encouraged for maintainable code.

Labels

Labels mark specific locations in code that can be targeted by `GOTO` and `GOSUB` statements. Unlike traditional BASIC which uses line numbers, EduBASIC uses descriptive labels.

Syntax:

```
LABEL labelName
```

Rules:

- Label names follow the same rules as variable names (alphanumeric, starting with a letter)
- Label names are case-insensitive
- Labels do **not** use type sigils
- Each label must be unique within its scope
- Labels can only appear at the beginning of a statement

Example:

```
LABEL StartProgram
PRINT "Beginning program..."

LABEL MainLoop
LET counter% = counter% + 1
IF counter% < 10 THEN GOTO MainLoop

LABEL EndProgram
PRINT "Program complete!"
```

GOTO Statement

The `GOTO` statement transfers control unconditionally to a labeled location.

Syntax:

```
GOTO labelName
```

Note: While `GOTO` is supported, excessive use creates "spaghetti code" that is difficult to read and maintain. Prefer structured alternatives like loops and procedures when possible.

Example:

```
LET count% = 0

LABEL LoopStart
PRINT count%
LET count% = count% + 1
IF count% < 5 THEN GOTO LoopStart

PRINT "Done!"
```

GOSUB and RETURN Statements

`GOSUB` calls a subroutine at a labeled location. `RETURN` returns control to the statement following the `GOSUB` call.

Syntax:

```
GOSUB labelName
...
LABEL labelName
    ' subroutine code
RETURN
```

Example:

```
PRINT "Starting..."
GOSUB PrintHeader
PRINT "Main program"
GOSUB PrintFooter
END

LABEL PrintHeader
    PRINT "===================="
    PRINT "  PROGRAM HEADER"
    PRINT "===================="
RETURN

LABEL PrintFooter
    PRINT "===================="
    PRINT "  PROGRAM FOOTER"
    PRINT "===================="
RETURN
```

Note: For modern code, prefer `SUB` procedures (described later) over `GOSUB` .

Important: `RETURN` is used exclusively for returning from `GOSUB` subroutines. `SUB` procedures use implicit return at `END SUB` or explicit early exit with `EXIT SUB` .

IF Statement

The `IF` statement executes code conditionally based on a boolean expression.

Single-line `IF` :

```
IF condition THEN statement
```

Block `IF`:

```
IF condition THEN
    statements
END IF
```

`IF-ELSE`:

```
IF condition THEN
    statements
ELSE
    statements
END IF
```

`IF-ELSEIF-ELSE`:

```
IF condition1 THEN
    statements
ELSEIF condition2 THEN
    statements
ELSEIF condition3 THEN
    statements
ELSE
    statements
END IF
```

Examples:

```
IF score% >= 90 THEN PRINT "Grade: A"

IF temperature# > 100 THEN
    PRINT "Water is boiling!"
    LET state$ = "gas"
END IF

IF age% < 13 THEN
    PRINT "Child"
ELSEIF age% < 20 THEN
    PRINT "Teenager"
ELSEIF age% < 65 THEN
    PRINT "Adult"
ELSE
```

```
    PRINT "Senior"
END IF
```

UNLESS Statement

The `UNLESS` statement is syntactic sugar for `IF NOT`, making negative conditions more readable.

Syntax:

```
UNLESS condition THEN statement

UNLESS condition THEN
    statements
END UNLESS

UNLESS condition THEN
    statements
ELSE
    statements
END UNLESS
```

Examples:

```
UNLESS gameOver% THEN GOSUB UpdateGame

UNLESS password$ = "secret" THEN
    PRINT "Access denied!"
    END
END UNLESS

UNLESS balance# >= price# THEN
    PRINT "Insufficient funds"
ELSE
    LET balance# -= price#
    PRINT "Purchase complete"
END UNLESS
```

Note: `UNLESS condition` is exactly equivalent to `IF NOT condition`.

SELECT CASE Statement

The `SELECT CASE` statement provides multi-way branching based on the value of an expression. This is EduBASIC's implementation of QuickBASIC's `SELECT CASE`.

Syntax:

```
SELECT CASE expression
    CASE value1
        statements
    CASE value2
        statements
    CASE value3, value4, value5
        statements
    CASE IS > value6
        statements
    CASE value7 TO value8
        statements
    CASE ELSE
        statements
END SELECT
```

`CASE` Clauses:

- **Single value:** `CASE 5` matches when expression equals `5`
- **Multiple values:** `CASE 1, 2, 3` matches when expression equals `1` , `2` , or `3`
- **Relational:** `CASE IS > 10` matches when expression is greater than `10`
    - Available operators: `=` , `<>` , `<` , `>` , `<=` , `>=`
- **Range:** `CASE 10 TO 20` matches when expression is between `10` and `20` (inclusive)
- **Default:** `CASE ELSE` matches when no other case matches (optional)

Examples:

```
SELECT CASE grade%
    CASE 90 TO 100
        PRINT "A"
    CASE 80 TO 89
        PRINT "B"
    CASE 70 TO 79
        PRINT "C"
    CASE 60 TO 69
        PRINT "D"
    CASE ELSE
        PRINT "F"
END SELECT
```

```
SELECT CASE command$
    CASE "QUIT", "EXIT", "Q"
        PRINT "Goodbye!"
        END
    CASE "HELP", "?"
        GOSUB ShowHelp
    CASE "SAVE"
        GOSUB SaveGame
    CASE ELSE
        PRINT "Unknown command"
END SELECT
```

```
SELECT CASE age%
    CASE IS < 0
        PRINT "Invalid age"
    CASE 0 TO 2
        PRINT "Infant"
    CASE 3 TO 12
        PRINT "Child"
    CASE 13 TO 19
        PRINT "Teenager"
    CASE IS >= 20
        PRINT "Adult"
END SELECT
```

FOR Loop

The `FOR` loop iterates a counter variable through a range of values.

Syntax:

```
FOR variable = startValue TO endValue
    statements
NEXT variable

FOR variable = startValue TO endValue STEP stepValue
    statements
NEXT variable
```

Rules:

- The loop variable must be numeric (integer or real)
- `STEP` is optional (defaults to `1`)
- `STEP` can be positive or negative
- The loop variable can be used inside the loop
- The `NEXT` statement can optionally specify the variable name for clarity

Examples:

```
FOR i% = 1 TO 10
    PRINT i%
NEXT i%
```

```
FOR count% = 0 TO 100 STEP 10
    PRINT count%
NEXT count%
```

```
FOR x# = 1.0 TO 0.0 STEP -0.1
    PRINT x#
NEXT x#
```

```
FOR row% = 1 TO 5
    FOR col% = 1 TO 5
        PRINT "*";
    NEXT col%
    PRINT
NEXT row%
```

WHILE Loop

The `WHILE` loop repeats while a condition is true, testing the condition before each iteration.

Syntax:

```
WHILE condition
    statements
WEND
```

Examples:

```
LET count% = 0
WHILE count% < 10
    PRINT count%
    LET count% += 1
WEND
```

```
LET input$ = ""
WHILE input$ <> "quit"
```

```
    INPUT "Enter command: ", input$
    PRINT "You entered: ", input$
WEND
```

```
WHILE NOT EOF fileHandle%
    LINE INPUT line$ FROM #fileHandle%
    PRINT line$
WEND
```

UNTIL Loop

The `UNTIL` loop is syntactic sugar for `WHILE NOT`, repeating until a condition becomes true.

Syntax:

```
UNTIL condition
    statements
UEND
```

Examples:

```
LET count% = 0
UNTIL count% >= 10
    PRINT count%
    LET count% += 1
UEND
```

```
LET input$ = ""
UNTIL input$ = "quit"
    INPUT "Enter command (or 'quit'): ", input$
    PRINT "You entered: ", input$
UEND
```

Note: `UNTIL condition` is exactly equivalent to `WHILE NOT condition`.

DO Loop

The `DO` loop provides flexible looping with conditions that can be tested at the beginning or end of the loop.

`DO WHILE` (condition tested at top):

```
DO WHILE condition
    statements
LOOP
```

`DO UNTIL` (condition tested at top):

```
DO UNTIL condition
    statements
LOOP
```

`DO-LOOP WHILE` (condition tested at bottom):

```
DO
    statements
LOOP WHILE condition
```

`DO-LOOP UNTIL` (condition tested at bottom):

```
DO
    statements
LOOP UNTIL condition
```

Unconditional `DO`:

```
DO
    statements
LOOP
```

Key Difference:

- Top-tested loops (`DO WHILE` / `DO UNTIL`) may never execute if the condition is initially `false` / `true`
- Bottom-tested loops (`DO ... LOOP WHILE` / `DO ... LOOP UNTIL`) always execute at least once

Examples:

```
LET password$ = ""
DO WHILE password$ <> "secret"
    INPUT "Enter password: ", password$
LOOP
```

```
DO
    INPUT "Enter a number (0 to quit): ", num%
    PRINT "You entered: ", num%
LOOP UNTIL num% = 0
```

```
PRINT "Press ESC to exit..."
DO
    LET key$ = INKEY$
    IF key$ = CHR 27 THEN EXIT DO
LOOP
```

EXIT Statement

The `EXIT` statement immediately exits from a loop or procedure.

Syntax:

```
EXIT FOR
EXIT WHILE
EXIT DO
EXIT SUB
```

Examples:

```
FOR i% = 1 TO 100
    IF numbers%[i%] = target% THEN
        PRINT "Found at position: ", i%
        EXIT FOR
    END IF
NEXT i%
```

```
DO
    INPUT "Enter value (negative to quit): ", value#
    IF value# < 0 THEN EXIT DO
    LET sum# += value#
LOOP
```

CONTINUE Statement

The `CONTINUE` statement skips the rest of the current loop iteration and continues with the next iteration.

Syntax:

```
CONTINUE FOR
CONTINUE WHILE
CONTINUE DO
```

Examples:

```
FOR i% = 1 TO 10
    IF i% MOD 2 = 0 THEN CONTINUE FOR    ' Skip even numbers
    PRINT i%    ' Only prints odd numbers: 1, 3, 5, 7, 9
NEXT i%
```

```
LET count% = 0
WHILE count% < 10
    LET count% += 1
    IF count% = 5 THEN CONTINUE WHILE    ' Skip printing 5
    PRINT count%
WEND
```

```
DO
    INPUT "Enter number (0 to quit): ", num%
    IF num% = 0 THEN EXIT DO
    IF num% < 0 THEN CONTINUE DO    ' Skip negative numbers
    PRINT "Positive number: ", num%
LOOP
```

Note: `CONTINUE` is different from `EXIT`. `CONTINUE` skips to the next iteration of the loop, while `EXIT` exits the loop entirely.

SUB Procedures

`SUB` defines a subroutine (procedure) that performs an action. Subroutines can accept parameters and modify variables through reference parameters or module-level variables.

Syntax:

```
SUB procedureName param1, param2, BYREF param3, ...
    statements
END SUB
```

Note: Parameters are passed **by value** by default. Prefix individual parameters with `BYREF` to pass by reference. Parentheses are not permitted in `SUB` declarations.

Calling a `SUB`:

The `CALL` statement invokes a subroutine procedure with arguments.

Syntax:

```
CALL procedureName arg1, arg2, ...
procedureName arg1, arg2, ...
```

Rules:

- The `CALL` keyword is optional—you can call a subroutine by name directly
- Parentheses are not permitted in `CALL` statements or direct subroutine calls
- Arguments are passed by position (first argument to first parameter, etc.)
- Arguments must match the parameter types (type sigils must match)
- Arguments are passed by value by default, unless the parameter is marked with `BYREF`

Examples:

```
CALL DrawBox 10, 5, "*"
DrawBox 20, 3, "#"    ' Same as CALL DrawBox 20, 3, "#"

SUB DrawBox width%, height%, char$
    FOR row% = 1 TO height%
        FOR col% = 1 TO width%
            PRINT char$;
        NEXT col%
        PRINT
    NEXT row%
END SUB
```

Rules:

- Parameters must include type sigils
- Parameters are passed **by value** by default (the subroutine receives a copy)
- Use `BYREF` keyword to pass **by reference** (allows modification of the original variable)
- Use `EXIT SUB` to return early
- `SUB` s can call other `SUB` s recursively

Parameter Passing

EduBASIC supports two parameter passing modes:

By Value (Default):

- The subroutine receives a **copy** of the argument
- Changes to the parameter do **not** affect the original variable
- This is the default behavior for all parameters

By Reference (with `BYREF`):

- The subroutine receives a **reference** to the original variable
- Changes to the parameter **do** affect the original variable
- Use the `BYREF` keyword before the parameter to enable this

Syntax:

```
SUB procedureName valueParam%, BYREF refParam#
    ' valueParam% is passed by value (default)
    ' refParam# is passed by reference (BYREF keyword)
END SUB
```

Example demonstrating the difference:

```
SUB TestParameters byValue%, BYREF byReference%
    LET byValue% = 999
    LET byReference% = 999
END SUB

LET x% = 100
LET y% = 100
CALL TestParameters x%, y%
PRINT "x =", x%        ' Prints: x = 100 (unchanged, passed by value)
PRINT "y =", y%        ' Prints: y = 999 (changed, passed by reference)
```

Local Variables

Variables created with the `LET` statement inside a `SUB` are **module-level** (global). To create variables that are local to the current stack frame, use the `LOCAL` statement.

Syntax:

```
LOCAL variable = expression
LOCAL array[] = expression
```

Rules:

- Creates variables scoped to the current stack frame
- Inside a `SUB` : variables are local to that procedure's stack frame
- Outside any `SUB` : variables are local to the bottom-level stack frame
- Local variables are only accessible within their stack frame
- Local variables are destroyed when their stack frame is popped (when the `SUB` exits)
- Local variables do not conflict with module-level variables of the same name
- Local variables must still include type sigils
- Can create local scalars or arrays
- Local arrays are destroyed when the stack frame is popped

Examples:

```
SUB DrawBox width%, height%, char$
    LOCAL row%
    LOCAL col%

    FOR row% = 1 TO height%
        FOR col% = 1 TO width%
            PRINT char$;
        NEXT col%
        PRINT
    NEXT row%
END SUB

SUB CalculateStats values#[], count%, BYREF average#, BYREF maximum#
    LOCAL sum# = 0
    LOCAL i%
    LOCAL buffer%[] = [0, 0, 0]    ' Local array

    LET maximum# = values#[1]

    FOR i% = 1 TO count%
        LET sum# += values#[i%]
        IF values#[i%] > maximum# THEN
            LET maximum# = values#[i%]
        END IF
    NEXT i%

    LET average# = sum# / count%
END SUB
```

Examples:

```
SUB DrawBox width%, height%, char$
    LOCAL row%
    LOCAL col%

    FOR row% = 1 TO height%
        FOR col% = 1 TO width%
            PRINT char$;
        NEXT col%
        PRINT
    NEXT row%
END SUB

CALL DrawBox 10, 5, "*"
DrawBox 20, 3, "#"
```

```
SUB CalculateStats values#[], count%, BYREF average#, BYREF maximum#
    LOCAL sum# = 0
    LOCAL i%

    LET maximum# = values#[1]

    FOR i% = 1 TO count%
        LET sum# += values#[i%]
        IF values#[i%] > maximum# THEN
            LET maximum# = values#[i%]
        END IF
    NEXT i%

    LET average# = sum# / count%
END SUB

DIM scores#[10]
' ... fill array ...
LET avg# = 0
LET max# = 0
CALL CalculateStats scores#, 10, avg#, max#
PRINT "Average:", avg#, "Maximum:", max#
```

```
SUB InitializeGame
    ' These are module-level variables (global)
    LET score% = 0
    LET level% = 1
    LET lives% = 3
    PRINT "Game initialized!"
END SUB
```

```
CALL InitializeGame
```

END Statement

The `END` statement terminates program execution immediately.

Syntax:

```
END
```

Example:

```
IF criticalError% THEN
    PRINT "Fatal error occurred"
    END
END IF
```

Note: `END` is different from `RETURN`, which returns from a subroutine. `END` terminates the entire program.

SLEEP Statement

The `SLEEP` statement pauses program execution for a specified number of milliseconds.

Syntax:

```
SLEEP milliseconds%
```

Examples:

```
PRINT "Starting in 3 seconds..."
SLEEP 1000    ' Wait 1 second
PRINT "2..."
SLEEP 1000    ' Wait 1 second
PRINT "1..."
SLEEP 1000    ' Wait 1 second
PRINT "Go!"
```

```
' Simple animation loop
FOR i% = 1 TO 10
    PRINT "Frame ", i%
    SLEEP 100    ' Wait 100ms between frames
NEXT i%


' Wait for user to read message
PRINT "Press any key to continue..."
SLEEP 2000    ' Give user 2 seconds to read
```

Rules:

- `milliseconds%` must be a non-negative integer
- The program pauses for the specified duration
- Other operations (like keyboard input) may still be processed during the sleep period, depending on the implementation

Error Handling

EduBASIC provides structured error handling through `TRY...CATCH...FINALLY` blocks. Errors in EduBASIC are represented as strings, making error messages easy to create, compare, and display.

TRY...CATCH...FINALLY Statement

The `TRY...CATCH...FINALLY` statement provides structured exception handling. Errors are represented as strings.

Syntax:

```
TRY
    statements
CATCH errorVariable$
    statements
FINALLY
    statements
END TRY
```

Rules:

- The `TRY` block contains code that might raise an error
- The `CATCH` block executes if an error occurs in the `TRY` block
- The `CATCH` block receives the error message as a string in `errorVariable$`
- The `FINALLY` block always executes, whether an error occurred or not
- `FINALLY` is optional—you can use `TRY...CATCH...END TRY` without `FINALLY`

- `CATCH` is also optional—you can use `TRY...FINALLY...END TRY` to ensure cleanup without handling errors
- Errors are strings, so you can compare them, display them, or process them as needed

Examples:

```
TRY
    OPEN "data.txt" FOR READ AS file%
    READFILE "data.txt" INTO content$
    CLOSE file%
CATCH error$
    PRINT "Error occurred: ", error$
FINALLY
    IF EXISTS "data.txt" THEN
        CLOSE file%
    END IF
END TRY
```

```
TRY
    LET result# = 10 / divisor%
    PRINT "Result: ", result#
CATCH error$
    IF error$ = "Division by zero" THEN
        PRINT "Cannot divide by zero"
    ELSE
        PRINT "Error: ", error$
    END IF
END TRY
```

```
' Try without catch (only finally for cleanup)
TRY
    OPEN "temp.txt" FOR WRITE AS file%
    WRITE "data" TO file%
FINALLY
    CLOSE file%
END TRY
```

```
' Try without finally (only catch for error handling)
TRY
    LET value% = VAL userInput$
    PRINT "Parsed value: ", value%
CATCH error$
    PRINT "Invalid input: ", error$
END TRY
```

Error Propagation:

- If an error occurs in a `TRY` block and is not caught, or if `THROW` is called in a `CATCH` block, the error propagates to the enclosing `TRY` block (if any)
- If no enclosing `TRY` block exists, the program terminates with the error message

THROW Statement

The `THROW` statement raises (throws) an error. Errors are represented as strings.

Syntax:

```
THROW errorMessage$
```

Examples:

```
IF divisor% = 0 THEN
    THROW "Division by zero"
END IF
```

```
IF NOT EXISTS filename$ THEN
    THROW "File not found: " + filename$
END IF
```

```
SUB ValidateInput (value%)
    IF value% < 0 THEN
        THROW "Value must be non-negative"
    END IF
    IF value% > 100 THEN
        THROW "Value must not exceed 100"
    END IF
END SUB

TRY
    ValidateInput userValue%
CATCH error$
    PRINT "Validation failed: ", error$
END TRY
```

Rules:

- `THROW` immediately transfers control to the nearest `CATCH` block
- If no `CATCH` block exists, the program terminates with the error message

- Error messages are strings, so you can construct them dynamically
- `THROW` can be called from anywhere, including inside `SUB` procedures

Nested Error Handling:

```
TRY
    TRY
        OPEN "config.txt" FOR READ AS configFile%
        READFILE "config.txt" INTO config$
    CATCH error$
        IF error$ = "File not found" THEN
            ' Try default config
            READFILE "default.txt" INTO config$
        ELSE
            THROW error$    ' Re-throw if not a "file not found" error
        END IF
    END TRY
CATCH error$
    PRINT "Failed to load configuration: ", error$
    END
END TRY
```

Date and Time Functions

EduBASIC provides simple date and time operators for educational purposes. These operators return formatted strings or integer timestamps that are easy to display and work with.

`DATE$` - Current date as string:

```
LET today$ = DATE$
PRINT "Today is: ", today$    ' Prints: "Today is: 2025-01-15" (YYYY-MM-DD
format)
```

Returns the current date as a string in `YYYY-MM-DD` format (ISO 8601 date format). This format is unambiguous and easy to parse.

`TIME$` - Current time as string:

```
LET now$ = TIME$
PRINT "Current time: ", now$    ' Prints: "Current time: 14:30:45" (HH:MM:SS
format)
```

Returns the current time as a string in `HH:MM:SS` format (24-hour format).

`NOW%` - Current timestamp:

```
LET timestamp% = NOW%
PRINT "Timestamp: ", timestamp%    ' Prints: "Timestamp: 1736968245" (Unix
timestamp)
```

Returns the current Unix timestamp (seconds since January 1, 1970, 00:00:00 UTC) as an integer. Useful for calculating time differences and storing absolute time values.

Examples:

```
' Display current date and time
PRINT "Date: ", DATE$
PRINT "Time: ", TIME$

' Calculate elapsed time
LET startTime% = NOW%
SLEEP 2000    ' Wait 2 seconds
LET endTime% = NOW%
LET elapsed% = endTime% - startTime%
PRINT "Elapsed: ", elapsed%, " seconds"


' Log with timestamp
SUB LogMessage (msg$)
    PRINT DATE$, " ", TIME$, " - ", msg$
END SUB

LogMessage "Program started"


' Check if it's a specific date
IF DATE$ = "2025-12-25" THEN
    PRINT "Merry Christmas!"
END IF
```

Design Notes:

- `DATE$` and `TIME$` return strings for simplicity and ease of display
- `NOW%` returns an integer timestamp for calculations
- All functions use UTC time to avoid timezone complexity in educational contexts
- The string formats (YYYY-MM-DD and HH:MM:SS) are standard and unambiguous
- No date parsing or manipulation functions are provided initially—keep it simple for educational use

Summary: Structured vs. Unstructured Flow Control

EduBASIC provides both structured and unstructured control flow:

Structured (Recommended):

- `IF` / `THEN` / `ELSE` and `UNLESS`
- `SELECT CASE`
- `FOR` loops
- `WHILE` and `UNTIL` loops
- `DO` loops
- `SUB` procedures with parameters
- `TRY...CATCH...FINALLY` error handling

Unstructured (Use Sparingly):

- `GOTO`
- `GOSUB` / `RETURN`

While `GOTO` and `GOSUB` are available for educational purposes and backwards compatibility, structured programming constructs produce more readable, maintainable code. Use labels and `GOTO` only when necessary or when demonstrating the evolution of programming techniques.

Note: EduBASIC does not have user-defined functions. Use `SUB` procedures with `BYREF` parameters to return computed values.

Text I/O

EduBASIC provides a text output system that is separate from the 640×480 graphics system. The text system is rendered as an overlay on top of the graphics display, allowing you to combine text output with graphics operations. Text is displayed in a character grid, and you can control the position, color, and formatting of text output.

Note: All colors in EduBASIC use 32-bit RGBA format, where each component (Red, Green, Blue, Alpha) is 8 bits, allowing for 256 levels per channel and full transparency control.

PRINT Statement

The `PRINT` statement outputs text and values to the text display. It can print scalars, arrays, or a combination of both.

Syntax:

```
PRINT expression1, expression2, ...
PRINT expression1, expression2, ...;
PRINT array[]
PRINT array[];
PRINT
```

Mixed Type Arguments:

The `PRINT` statement accepts expressions of any type (Integer, Real, Complex, String, Array) separated by commas. Each expression is automatically converted to its string representation and concatenated with no spacing between items.

Output Formatting:

- **Comma (`,`):** Separates items with no spacing (concatenated)
- **Semicolon (`;`) at end:** Suppresses the newline, so the next `PRINT` continues on the same line
- **No separator at end:** Ends the line (adds a newline)
- **Empty `PRINT` :** Outputs a blank line (newline only)
- **Array:** When printing an array, all elements are printed in brackets with spaces after commas (e.g., `[1, 2, 3, 4, 5]`)

Type Conversion:

All value types are automatically converted to strings when printed:

- **Integer:** Decimal representation (e.g., `42` → `"42"`)
- **Real:** Decimal representation (e.g., `3.14` → `"3.14"`)
- **Complex:** Format `"real+imaginaryi"` or `"real-imaginaryi"` (e.g., `3+4i` → `"3+4i"`)
- **String:** Printed as-is
- **Array:** All elements printed in brackets with spaces after commas (e.g., `[1, 2, 3]` → `"[1, 2, 3]"`)

Controlling Newlines:

- To avoid a newline at the end of output, end the `PRINT` statement with a semicolon
- To output a blank line, use an empty `PRINT` statement (no arguments)

Examples:

Basic Printing:

```
PRINT "Hello, world!"
PRINT "Name: ", name$, " Age: ", age%
PRINT "X: ", x%, " Y: ", y%
```

Mixed Types:

```
LET count% = 10
LET price# = 19.99
LET name$ = "Widget"
LET z& = 3+4i

PRINT "Item: ", name$, " Count: ", count%, " Price: ", price#, " Complex: ", z&
' Prints: Item: Widget Count: 10 Price: 19.99 Complex: 3+4i
```

Print Arrays:

```
PRINT numbers%[]     ' Prints: [1, 2, 3, 4, 5]
PRINT "Scores: ", scores%[]     ' Prints: Scores: [85, 90, 78, 92]
```

Suppress Newline:

```
PRINT "Enter your name: ";
INPUT name$
```

Print Blank Lines:

```
PRINT
PRINT "Line 1"
PRINT
PRINT "Line 3"
```

Multiple Items on Same Line:

```
PRINT "Count: ", count%, "  ";
PRINT "Total: ", total#
```

String Concatenation Alternative:

You can also use string concatenation with the `+` operator to build strings before printing. This is useful when you need more control over spacing or formatting:

```
' Using PRINT with commas (no spacing)
PRINT "Name: ", name$, "Age: ", age%
```

```
' Prints: Name: AliceAge: 25

' Using string concatenation (with spacing)
PRINT "Name: " + name$ + " Age: " + STR age%
' Prints: Name: Alice Age: 25

' Complex formatting with concatenation
LET message$ = "Result: " + STR result# + " (" + STR count% + " items)"
PRINT message$
```

When to Use Each Approach:

- **Use commas in PRINT:** When you want automatic type conversion and no spacing between items
- **Use string concatenation:** When you need explicit spacing, complex formatting, or want to build the string separately

Special Characters:

- `\n` - Newline
- `\t` - Tab
- `\"` - Literal double quote
- `\\` - Literal backslash

INPUT Statement

The `INPUT` statement reads a value from the user and assigns it to a variable. Unlike traditional BASIC dialects, `INPUT` is separate from the prompt, allowing you to use `PRINT` for more flexible prompt formatting. `INPUT` can read into scalar variables or arrays.

Syntax:

```
INPUT variable
INPUT array[]
```

Rules:

- `INPUT` can read any data type (integer, real, string, complex)
- The variable's type sigil determines what type of value is expected
- The prompt is displayed separately using `PRINT` before the `INPUT` statement
- After the user enters a value and presses Enter, the value is assigned to the variable
- If the input cannot be converted to the variable's type, an error occurs

- For arrays, `INPUT` reads multiple values separated by commas, filling the array from index 1 onwards
- If more values are entered than the array size, excess values are ignored
- If fewer values are entered, remaining array elements are unchanged

Examples:

```
' Reading an integer
PRINT "Enter your age: ";
INPUT age%
PRINT "You are ", age%, " years old"

' Reading a real number
PRINT "Enter temperature: ";
INPUT temperature#
PRINT "Temperature is ", temperature#

' Reading a string
PRINT "Enter your name: ";
INPUT name$
PRINT "Hello, ", name$, "!"

' Reading a complex number
PRINT "Enter complex number (e.g., 3+4i): ";
INPUT z&
PRINT "You entered: ", z&

' Reading multiple values
PRINT "Enter X coordinate: ";
INPUT x%
PRINT "Enter Y coordinate: ";
INPUT y%
PRINT "Position: (", x%, ", ", y%, ")"

' Reading into an array
DIM scores%[5]
PRINT "Enter 5 scores (comma-separated): ";
INPUT scores%[]
PRINT "Scores: ", scores%[]

' Using LOCATE for formatted input
LOCATE 10, 1
PRINT "Name: ";
INPUT playerName$
LOCATE 11, 1
PRINT "Score: ";
INPUT score%
```

Note: Since `INPUT` is separate from the prompt, you have full control over how prompts are displayed. Use `PRINT` with a semicolon at the end to keep the cursor on the same line as the prompt, or use `LOCATE` to position prompts precisely.

LOCATE Statement

The `LOCATE` statement positions the text cursor at a specific row and column in the text display.

Syntax:

```
LOCATE row%, column%
```

Rules:

- Row and column are 1-based (row 1, column 1 is the top-left corner)
- The text display has a fixed character grid size
- After `LOCATE`, subsequent `PRINT` statements output at the specified position

Examples:

```
LOCATE 10, 20
PRINT "This text appears at row 10, column 20"

LOCATE 1, 1
PRINT "Top-left corner"

FOR row% = 1 TO 10
    LOCATE row%, row%
    PRINT "*"
NEXT row%
```

COLOR Statement

The `COLOR` statement sets the global foreground and/or background color for both text and graphics operations. These colors are used by default, but can be overridden in individual statements when needed.

Syntax:

```
COLOR foregroundColor%
COLOR foregroundColor%, backgroundColor%
COLOR , backgroundColor%
```

Color Format:

- Colors can be specified as:
  - **32-bit RGBA integers**: Format `&HRRGGBBAA` (hexadecimal) - always use this format in examples
  - **Color names**: Standard CSS color names as strings (case-insensitive)
- Each component (R, G, B, A) ranges from 0-255
- Alpha channel controls transparency (0 = fully transparent, 255 = fully opaque)
- Color names use full opacity (alpha = 255) by default
- String expressions are automatically recognized as color names when they match a valid CSS color name

Text Usage:

- Sets the global foreground color for subsequent `PRINT` statements
- With two arguments, sets both global foreground and background colors
- With comma and only background color (`COLOR , backgroundColor%`), sets only the background color (foreground unchanged)
- Background color is only used for text output

Graphics Usage:

- Sets the global foreground color for subsequent graphics operations
- Affects `PSET` , `LINE` , `RECTANGLE` , `OVAL` , `TRIANGLE` , and other drawing operations
- Graphics statements can optionally override the global color using `WITH color%`
- Does not affect `PUT` operations (which use the sprite's stored colors with alpha blending)
- Only the foreground color is used for graphics (background parameter is ignored)

Common Colors:

```
LET black% = &H000000FF       ' Black (opaque)
LET white% = &HFFFFFFFF       ' White (opaque)
LET red% = &HFF0000FF         ' Red (opaque)
LET green% = &H00FF00FF       ' Green (opaque)
LET blue% = &H0000FFFF        ' Blue (opaque)
LET yellow% = &HFFFF00FF      ' Yellow (opaque)
LET transparent% = &HFFFFFF00 ' White (fully transparent)
```

Examples:

```
' Text usage with hex colors
COLOR &HFF0000FF        ' Set global foreground to red
PRINT "This is red text"

COLOR &HFFFFFF00, &H000000FF  ' Transparent text on black background
PRINT "Invisible text on black"

COLOR , &H000000FF      ' Set only background to black (foreground unchanged)
PRINT "Text with black background"

' Text usage with color names
COLOR "red"             ' Set global foreground to red using color name
COLOR "blue", "white"   ' Blue text on white background

' Graphics usage - using global color
COLOR &HFF0000FF    ' Set global foreground to red
PSET (100, 100)     ' Red pixel (uses global color)

COLOR "red"         ' Set global foreground to red using color name
PSET (100, 100)     ' Red pixel (uses global color)

' Graphics usage - overriding global color
COLOR &H00FF00FF    ' Set global foreground to green
LINE FROM (0, 0) TO (100, 100) WITH &HFF0000FF    ' Red line (overrides global)

LINE FROM (0, 0) TO (100, 100) WITH "red"    ' Red line using color name
LINE FROM (0, 0) TO (100, 100) WITH "cornflowerblue"    ' Cornflower blue line
```

Note: The `COLOR` statement sets global colors that persist until changed. The default text color is white on a transparent background. Graphics statements use the global foreground color by default, but can override it with `WITH color%` .

SET Statement

The `SET` statement configures system-wide settings.

Syntax:

```
SET LINE SPACING ON
SET LINE SPACING OFF
SET TEXT WRAP ON
SET TEXT WRAP OFF
SET AUDIO ON
SET AUDIO OFF
SET VOLUME volume#
```

Line Spacing

Text Grid Dimensions:

EduBASIC uses IBM Plex Mono font, where each character is rendered at exactly 8×16 graphics pixels. The text display grid dimensions depend on the line spacing setting:

- **With line spacing OFF (default):** The text grid is **80×30 characters**
  - Width: 640 pixels ÷ 8 pixels per character = 80 columns
  - Height: 480 pixels ÷ 16 pixels per character = 30 rows
  - Characters are rendered with no extra spacing between lines
- **With line spacing ON:** The text grid is **80×24 characters**
  - Width: 640 pixels ÷ 8 pixels per character = 80 columns (unchanged)
  - Height: 480 pixels ÷ (16 pixels per character + 4 pixels spacing) = 24 rows
  - Each line of text has 4 additional pixels of spacing after it, making text more readable but reducing the number of available rows

Text Wrapping

The text wrap setting controls whether long lines of text automatically wrap to the next line when they exceed the width of the text display.

- **With text wrap OFF (default):** Long lines are truncated at the right edge of the display (80 characters)
- **With text wrap ON:** Long lines automatically wrap to the next line, breaking at word boundaries when possible

Note: Text wrapping only affects lines that exceed the display width. Short lines are unaffected.

Audio

The audio setting controls whether audio output is enabled or disabled.

- **With audio ON (default):** All audio output is enabled. `PLAY` , `VOLUME` , and `TEMPO` statements work normally.
- **With audio OFF:** All audio output is disabled. `PLAY` statements are ignored, and no sound is produced.

Note: When audio is OFF, `PLAY` statements execute without error but produce no sound. The `NOTES` operator still returns valid values for voices that have been configured, but no actual audio playback occurs.

Volume

The `SET VOLUME` command sets the global audio volume as a real number between 0.0 and 1.0 (inclusive). The value is automatically clamped to the range [0, 1].

- **Volume 0.0:** Silent (no audio output)
- **Volume 1.0:** Maximum volume (100%)
- **Values outside [0, 1]:** Automatically clamped to the nearest valid value

Note: `SET VOLUME` is an alternative to the `VOLUME` statement. `VOLUME` uses integer values 0-127, while `SET VOLUME` uses real values 0.0-1.0. Both commands control the same global volume setting.

Examples:

```
SET LINE SPACING OFF    ' Use 80×30 character grid (default)
LOCATE 30, 1
PRINT "Bottom row"

SET LINE SPACING ON     ' Use 80×24 character grid
LOCATE 24, 1
PRINT "Bottom row with spacing"

SET TEXT WRAP OFF       ' Truncate long lines (default)
PRINT "This is a very long line that will be truncated at 80 characters..."

SET TEXT WRAP ON        ' Wrap long lines automatically
PRINT "This is a very long line that will automatically wrap to the next line
when it exceeds 80 characters..."

SET AUDIO ON            ' Enable audio output (default)
PLAY 0, "CDEFGAB C"

SET AUDIO OFF           ' Disable audio output
PLAY 0, "CDEFGAB C"     ' No sound produced, but no error

SET VOLUME 1.0          ' Maximum volume
SET VOLUME 0.5          ' Half volume
SET VOLUME 0.0          ' Silent
SET VOLUME 1.5          ' Clamped to 1.0 (maximum)
SET VOLUME -0.5         ' Clamped to 0.0 (silent)
```

Note: All settings persist until changed. The line spacing setting affects the entire text display. Text wrapping only affects lines that exceed the display width. Audio settings affect all audio output globally. Volume values are automatically clamped to [0, 1].

Keyboard Input

EduBASIC provides non-blocking keyboard input through the `INKEY$` operator, which allows programs to detect keypresses without waiting for user input. This is useful for games, interactive applications, and real-time input handling.

INKEY$ Operator

The `INKEY$` operator returns the currently pressed key as a string.

Key Detection:

- `INKEY$` returns the currently pressed key as a string
- Returns empty string (`""`) if no key is pressed
- Non-blocking: returns immediately whether a key is pressed or not
- Supports printable characters and special keys

Return Values:

- **Empty string (`""`):** No key is currently pressed
- **Printable characters:** Returns the character as a string (e.g., `"a"` , `"A"` , `"1"` , `" "`)
- **Special keys:** Returns special key names (see table below)

Special Key Names:

Key	Return Value
Escape	`"ESC"`
Enter/Return	`"ENTER"`
Backspace	`"BACKSPACE"`
Tab	`"TAB"`
Space	`" "` (space character)
Arrow Up	`"ARROWUP"`
Arrow Down	`"ARROWDOWN"`
Arrow Left	`"ARROWLEFT"`
Arrow Right	`"ARROWRIGHT"`

Key	Return Value
Home	`"HOME"`
End	`"END"`
Page Up	`"PAGEUP"`
Page Down	`"PAGEDOWN"`
Insert	`"INSERT"`
Delete	`"DELETE"`
F1-F12	`"F1"` through `"F12"`
Shift	`"SHIFT"`
Control	`"CTRL"`
Alt	`"ALT"`
Caps Lock	`"CAPSLOCK"`

Examples:

```
' Game loop with keyboard input
DO
    LET key$ = INKEY$
    IF key$ <> "" THEN
        IF key$ = "ESC" THEN EXIT DO
        IF key$ = "ARROWUP" THEN LET y% -= 1
        IF key$ = "ARROWDOWN" THEN LET y% += 1
        IF key$ = "ARROWLEFT" THEN LET x% -= 1
        IF key$ = "ARROWRIGHT" THEN LET x% += 1
    END IF
    ' ... game logic ...
LOOP

' Menu navigation
LET key$ = INKEY
IF key$ = "F1" THEN GOSUB ShowHelp
IF key$ = "F2" THEN GOSUB SaveGame
IF key$ = "ENTER" THEN GOSUB SelectOption
```

Note: For blocking input that waits for the user to press Enter, use the `INPUT` statement instead of `INKEY$` .

String Operations

EduBASIC provides several operations for working with string data.

String Concatenation

Strings can be concatenated using the `+` operator or by using commas in `PRINT` statements.

Syntax:

```
LET result$ = string1$ + string2$
LET result$ = string1$ + "literal" + string2$
```

Examples:

```
LET firstName$ = "John"
LET lastName$ = "Doe"
LET fullName$ = firstName$ + " " + lastName$
PRINT fullName$    ' Prints: John Doe

LET greeting$ = "Hello, " + name$ + "!"
LET message$ = "Value: " + STR number%
```

String Operators

EduBASIC provides string operators to extract and manipulate substrings:

`LEFT` - Extract left portion of string:

```
LET text$ = "Hello, world!"
LET firstWord$ = text$ LEFT 5    ' "Hello"
LET firstChar$ = text$ LEFT 1    ' "H"
```

Returns the leftmost `n` characters of the string. If `n` is greater than the string length, returns the entire string.

`RIGHT` - Extract right portion of string:

```
LET text$ = "Hello, world!"
LET lastWord$ = text$ RIGHT 6    ' "world!"
LET lastChar$ = text$ RIGHT 1    ' "!"
```

Returns the rightmost `n` characters of the string. If `n` is greater than the string length, returns the entire string.

String Slicing

`MID` - Extract substring from middle:

```
LET text$ = "Hello, world!"
LET world$ = text$ MID 8 TO 12    ' "world" (positions 8 to 12)
LET middle$ = text$ MID 3 TO 6    ' "llo," (positions 3 to 6)
```

Returns a substring from position `start%` to position `end%` (inclusive). If `start%` is beyond the string length, returns an empty string. If `end%` extends beyond the string, returns characters up to the end of the string.

Single character access:

```
LET text$ = "Hello"
LET char$ = text$ MID 1 TO 1    ' "H" (single character at position 1)
```

String Length

Use the `||` operator to get the length of a string. Spaces are required inside the `||` operator.

Syntax:

```
LET length% = | string$ |
```

Example:

```
LET text$ = "Hello"
LET size% = | text$ |    ' size% = 5
```

String Comparison

Strings can be compared using standard comparison operators (`=` , `<>` , `<` , `>` , `<=` , `>=`). Comparisons are case-sensitive and use lexicographic (alphabetical) ordering.

Examples:

```
IF name$ = "Alice" THEN PRINT "Found Alice"
IF text1$ <> text2$ THEN PRINT "Different"
IF word$ < "middle" THEN PRINT "Comes before 'middle'"
```

STARTSWITH and ENDSWITH Operators

`STARTSWITH` - Check if string starts with prefix:

```
LET filename$ = "document.txt"
IF filename$ STARTSWITH "doc" THEN PRINT "Starts with 'doc'"
IF filename$ STARTSWITH ".txt" THEN PRINT "Starts with '.txt'"    ' FALSE%
```

Returns TRUE% (-1) if the string starts with the specified prefix, FALSE% (0) otherwise. Case-sensitive.

`ENDSWITH` - Check if string ends with suffix:

```
LET filename$ = "document.txt"
IF filename$ ENDSWITH ".txt" THEN PRINT "Ends with '.txt'"
IF filename$ ENDSWITH "doc" THEN PRINT "Ends with 'doc'"    ' FALSE
```

Returns TRUE (-1) if the string ends with the specified suffix, FALSE (0) otherwise. Case-sensitive.

String Operators

`STR` - Convert number to string:

```
LET num% = 42
LET numStr$ = STR num%    ' "42"
LET piStr$ = STR 3.14159  ' "3.14159"
LET complexStr$ = STR (3+4i)    ' "3+4i"
```

Converts any numeric type (integer, real, or complex) to its decimal string representation. For complex numbers, the format is `"real+imaginaryi"` or `"real-imaginaryi"` .

`VAL` - Convert string to number:

```
LET text$ = "123"
LET number% = VAL text$    ' 123
LET decimal$ = "3.14"
LET value# = VAL decimal$  ' 3.14
LET hexValue% = VAL "&HFF"    ' 255 (hexadecimal)
LET binValue% = VAL "&B1010"   ' 10 (binary)
LET complexValue& = VAL "3+4i"    ' 3+4i (complex number)
```

Converts a string to a number. Supports decimal, hexadecimal (with `&H` prefix), binary (with `&B` prefix), and complex number formats. The result type depends on the string content. Hexadecimal and binary strings are parsed as integers.

`HEX` - Convert number to hexadecimal string:

```
LET hexStr$ = HEX 255    ' "FF"
LET hexStr$ = HEX 10     ' "A"
LET hexStr$ = HEX -1     ' "FFFFFFFF" (32-bit two's complement)
```

Converts an integer to its hexadecimal string representation (uppercase, no prefix). Negative numbers are represented using two's complement notation.

`BIN` - Convert number to binary string:

```
LET binStr$ = BIN 10     ' "1010"
LET binStr$ = BIN 255    ' "11111111"
LET binStr$ = BIN -1     ' "11111111111111111111111111111111" (32-bit two's
complement)
```

Converts an integer to its binary string representation. Negative numbers are represented using two's complement notation.

`CHR` - Convert ASCII code to character:

```
LET char$ = CHR 65         ' "A"
LET newline$ = CHR 10      ' Newline character
```

`ASC` - Convert character to ASCII code:

```
LET code% = ASC "A"         ' 65
LET code% = ASC "a"         ' 97
```

`UCASE` - Convert to uppercase:

```
LET upper$ = UCASE "hello"    ' "HELLO"
```

`LCASE` - Convert to lowercase:

```
LET lower$ = LCASE "WORLD"    ' "world"
```

`LTRIM` **- Remove leading spaces:**

```
LET trimmed$ = LTRIM "  text"    ' "text"
```

`RTRIM` **- Remove trailing spaces:**

```
LET trimmed$ = RTRIM "text  "    ' "text"
```

`TRIM` **- Remove leading and trailing spaces:**

```
LET trimmed$ = TRIM "  text  "   ' "text"
```

`INSTR` **- Find substring position:**

```
LET pos% = "Hello world" INSTR "world"    ' 7
LET pos% = "Hello world" INSTR "o" FROM 5    ' 5 (start search at position 5)
```

`REPLACE` **- Replace substring:**

```
LET new$ = "Hello world" REPLACE "world" WITH "EduBASIC"    ' "Hello EduBASIC"
```

File I/O

EduBASIC provides comprehensive file input/output operations for reading and writing both text and binary data. All file operations use UTF-8 encoding for text data, and files can mix text and binary operations seamlessly.

Note: File handles are integer identifiers that reference open files. You must explicitly open files before use and close them when finished.

Opening and Closing Files

OPEN Statement

The `OPEN` statement opens a file and assigns a handle to a variable.

Syntax:

```
OPEN "filename" FOR mode AS fileHandle%
```

Modes:

- `READ` - Open file for reading only
- `APPEND` - Open file for writing, appending to the end
- `OVERWRITE` - Open file for writing, replacing existing content

Rules:

- The file handle variable receives an integer identifier
- Files must be opened before any read/write operations
- Attempting to open a non-existent file in `READ` mode causes an error
- Opening a non-existent file in `APPEND` or `OVERWRITE` mode creates the file
- The file handle is just an integer ID; it does not store file state

Examples:

```
OPEN "data.txt" FOR READ AS inputFile%
OPEN "output.txt" FOR OVERWRITE AS outputFile%
OPEN "log.txt" FOR APPEND AS logFile%
```

CLOSE Statement

The `CLOSE` statement closes an open file.

Syntax:

```
CLOSE fileHandle%
```

Examples:

```
OPEN "data.txt" FOR READ AS file%
' ... read operations ...
CLOSE file%
```

Reading from Files

EduBASIC supports both text and binary reading operations.

LINE INPUT Statement (Text)

The `LINE INPUT` statement reads a complete line of text from a file.

Syntax:

```
LINE INPUT lineVariable$ FROM #fileHandle%
```

Rules:

- Reads one line including the newline character
- The newline character is included in the string
- At end of file, an error occurs (check with `EOF` first)

Examples:

```
OPEN "data.txt" FOR READ AS file%

WHILE NOT EOF file%
    LINE INPUT line$ FROM #file%
    PRINT line$
WEND

CLOSE file%
```

READ Statement (Binary)

The `READ` statement reads binary data from a file based on the variable's type. It can read into scalar variables or arrays.

Syntax:

```
READ variable FROM fileHandle%
READ array[] FROM fileHandle%
```

Rules:

- Reads binary data matching the variable's type
- Integer: reads 4 bytes (32-bit signed integer)
- Real: reads 8 bytes (64-bit floating-point)
- Complex: reads 16 bytes (128-bit complex number)
- String: reads the string's length prefix and data

- For arrays, reads elements sequentially, filling the array from index 1 onwards
- Reading advances the file position
- At end of file, an error occurs (check with `EOF` first)

Examples:

```
OPEN "data.bin" FOR READ AS file%

READ count% FROM file%
DIM numbers%[count%]

FOR i% = 1 TO count%
    READ numbers%[i%] FROM file%
NEXT i%

' Or read entire array at once
DIM data%[100]
READ data%[] FROM file%

CLOSE file%
```

Writing to Files

EduBASIC supports both text and binary writing operations.

WRITE Statement (Text and Binary)

The `WRITE` statement writes data to a file. For text, it writes the string representation. For binary, it writes the raw binary data. It can write scalars or arrays.

Syntax:

```
WRITE expression TO fileHandle%
WRITE "text" TO fileHandle%
WRITE variable TO fileHandle%
WRITE array[] TO fileHandle%
```

Rules:

- For strings: writes the text followed by a newline
- For numbers: writes the binary representation (not text)
- For arrays, writes all elements sequentially in binary format
- Multiple `WRITE` statements can be used to build file content

- Text and binary operations can be mixed in the same file

Examples:

```
OPEN "output.txt" FOR OVERWRITE AS file%

WRITE "Name: " TO file%
WRITE playerName$ TO file%
WRITE "Score: " TO file%
WRITE score% TO file%
WRITE numbers%[] TO file%    ' Write entire array

CLOSE file%
```

Binary Writing Example:

```
OPEN "data.bin" FOR OVERWRITE AS file%

LET count% = 5
WRITE count% TO file%    ' Write binary integer

FOR i% = 1 TO count%
    WRITE numbers%[i%] TO file%    ' Write binary integers
NEXT i%

CLOSE file%
```

File Navigation

SEEK Statement

The `SEEK` statement positions the file pointer at a specific byte position.

Syntax:

```
SEEK position% IN #fileHandle%
```

Rules:

- Position is always in bytes (0-based)
- Position 0 is the beginning of the file
- For text files, positions refer to UTF-8 byte positions
- Seeking past end of file is allowed (file will extend on write)

Examples:

```
OPEN "data.bin" FOR READ AS file%

SEEK 100 IN #file%    ' Jump to byte 100
READ value% FROM file%

SEEK 0 IN #file%      ' Return to beginning
CLOSE file%
```

EOF Operator

The `EOF` operator is a unary operator that checks if the file pointer is at the end of the file.

Syntax:

```
EOF fileHandle%
```

Returns: Integer (0 = false, -1 = true)

Examples:

```
OPEN "data.txt" FOR READ AS file%

WHILE NOT EOF file%
    LINE INPUT line$ FROM #file%
    PRINT line$
WEND

CLOSE file%
```

LOC Operator

The `LOC` operator is a unary operator that returns the current byte position in the file.

Syntax:

```
LOC fileHandle%
```

Returns: Integer (current byte position, 0-based)

Examples:

```
OPEN "data.bin" FOR READ AS file%

LET startPos% = LOC file%
READ value% FROM file%
LET endPos% = LOC file%
LET bytesRead% = endPos% - startPos%

CLOSE file%
```

Convenience File Operations

EduBASIC provides convenient statements for common file operations.

READFILE Statement

The `READFILE` statement reads an entire file into a string variable.

Syntax:

```
READFILE "filename" INTO contentVariable$
```

Examples:

```
READFILE "config.txt" INTO config$
PRINT config$

READFILE "data.json" INTO jsonData$
' Process jsonData$
```

WRITEFILE Statement

The `WRITEFILE` statement writes an entire string to a file.

Syntax:

```
WRITEFILE "filename" FROM contentVariable$
WRITEFILE contentVariable$ TO "filename"
```

Examples:

```
LET report$ = "Sales Report" + CHR 10 + "Total: $1000"
WRITEFILE "report.txt" FROM report$
WRITEFILE output$ TO "results.txt"
```

LISTDIR Statement

The `LISTDIR` statement lists files in a directory.

Syntax:

```
LISTDIR "path" INTO filesArray$[]
```

Rules:

- Returns an array of filenames (strings)
- Array is 1-based
- Includes files and subdirectories
- The array is created automatically if it doesn't exist

Examples:

```
LISTDIR "." INTO files$[]
FOR i% = 1 TO | files$[] |
    PRINT files$[i%]
NEXT i%

LISTDIR "/Users/name/Documents" INTO docs$[]
```

MKDIR Statement

The `MKDIR` statement creates a directory.

Syntax:

```
MKDIR "path"
```

Examples:

```
MKDIR "backups"
MKDIR "/Users/name/data"
```

RMDIR Statement

The `RMDIR` statement removes an empty directory.

Syntax:

```
RMDIR "path"
```

Examples:

```
RMDIR "temp"
RMDIR "/Users/name/old_data"
```

COPY Statement

The `COPY` statement copies a file.

Syntax:

```
COPY "source" TO "destination"
```

Examples:

```
COPY "data.txt" TO "backup.txt"
COPY "/source/file.bin" TO "/dest/file.bin"
```

MOVE Statement

The `MOVE` statement moves or renames a file.

Syntax:

```
MOVE "source" TO "destination"
```

Examples:

```
MOVE "oldname.txt" TO "newname.txt"
MOVE "/temp/file.txt" TO "/final/file.txt"
```

DELETE Statement

The `DELETE` statement deletes a file.

Syntax:

```
DELETE "filename"
```

Examples:

```
DELETE "temp.txt"
DELETE "/Users/name/old_data.bin"
```

File I/O Examples

Example: Reading and Writing Text:

```
OPEN "students.txt" FOR READ AS inputFile%
OPEN "grades.txt" FOR OVERWRITE AS outputFile%

WHILE NOT EOF inputFile%
    LINE INPUT line$ FROM #inputFile%
    ' Parse line$ to extract name$ and score%
    ' (parsing logic here)

    IF score% >= 90 THEN
        WRITE name$ TO outputFile%
        WRITE "A" TO outputFile%
    END IF
WEND

CLOSE inputFile%
CLOSE outputFile%
```

Example: Binary Data Storage:

```
' Write binary data
OPEN "scores.bin" FOR OVERWRITE AS file%
LET count% = 3
WRITE count% TO file%
WRITE 95 TO file%
WRITE 87 TO file%
WRITE 92 TO file%
CLOSE file%

' Read binary data
OPEN "scores.bin" FOR READ AS file%
READ count% FROM file%
```

```
DIM scores%[count%]
FOR i% = 1 TO count%
    READ scores%[i%] FROM file%
NEXT i%
CLOSE file%
```

Example: Mixed Text and Binary:

```
OPEN "mixed.dat" FOR OVERWRITE AS file%

' Write text header
WRITE "Data File v1.0" TO file%

' Write binary data
LET recordCount% = 10
WRITE recordCount% TO file%
FOR i% = 1 TO recordCount%
    WRITE data%[i%] TO file%
NEXT i%

' Write text footer
WRITE "End of file" TO file%

CLOSE file%
```

Graphics

EduBASIC provides a comprehensive graphics system for drawing shapes, sprites, and images. The graphics display is separate from the text system and is rendered at a fixed resolution of 640×480 pixels.

Coordinate System:

- Graphics coordinates use **(0, 0) at the bottom-left corner** (mathematical convention)
- X increases to the right (0 to 639)
- Y increases upward (0 to 479)
- This is different from the text grid, which uses 1-based coordinates with top-left origin

Color Format:

- All graphics operations use **32-bit RGBA** color format
- Format: `&HRRGGBBAA` (hexadecimal) - always use this format in examples
- Each component (R, G, B, A) ranges from 0-255
- Alpha channel controls transparency (0 = fully transparent, 255 = fully opaque)

Coordinate Syntax:

- Graphics commands use parentheses to denote planar points: `(x, y)`
- This is an exception to EduBASIC's general rule that parentheses are only for grouping expressions
- Coordinates are always written as `(x%, y%)` or `(x#, y#)` in graphics commands

Color Usage:

- Colors are 32-bit RGBA integers in `&HRRGGBBAA` format
- The `COLOR` statement sets global foreground and background colors
- Graphics statements use the global foreground color by default
- Graphics statements can override the global color using `WITH color%` when needed

Note: The text display system overlays the graphics display, allowing you to combine text and graphics in the same program.

CLS Statement

The `CLS` statement clears the graphics screen.

Syntax:

```
CLS
CLS WITH backgroundColor%
```

Rules:

- Clears the entire graphics display (640×480 pixels)
- Without a color, clears to black (`&H000000FF`)
- With `WITH backgroundColor%` , clears to the specified background color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format
- Does not affect the text display overlay
- Automatically switches to the output tab

Examples:

```
CLS    ' Clear to black

CLS WITH &H000033FF    ' Clear to dark blue

' Clear to a custom color
```

```
LET bgColor% = &HFF6600FF    ' Orange background
CLS WITH bgColor%
```

PSET Statement

The `PSET` statement sets a single pixel to a color.

Syntax:

```
PSET (x%, y%)
PSET (x%, y%) WITH color%
```

Rules:

- Coordinates are in graphics pixels (0-based, bottom-left origin)
- X ranges from 0 to 639
- Y ranges from 0 to 479
- Uses global foreground color (set with `COLOR`) by default
- `WITH color%` overrides the global color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format

Examples:

```
COLOR &HFF0000FF    ' Set global color to red
PSET (100, 200)     ' Red pixel (uses global color)

PSET (200, 200) WITH &HFFFFFFFF    ' White pixel (overrides global)

FOR i% = 0 TO 639
    PSET (i%, 240) WITH &HFFFFFFFF    ' White horizontal line
NEXT i%
```

LINE Statement

The `LINE` statement draws a line between two points.

Syntax:

```
LINE FROM (x1%, y1%) TO (x2%, y2%)
LINE FROM (x1%, y1%) TO (x2%, y2%) WITH color%
```

Rules:

- Draws a line from point (x1%, y1%) to point (x2%, y2%)
- Uses global foreground color (set with `COLOR`) by default
- `WITH color%` overrides the global color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format
- Coordinates are in graphics pixels (0-based, bottom-left origin)
- For rectangles, use the `RECTANGLE` statement instead

Examples:

```
COLOR &H00FF00FF    ' Set global color to green
LINE FROM (10, 10) TO (100, 50)    ' Green line (uses global color)

LINE FROM (50, 50) TO (150, 150) WITH &HFFFFFFFF    ' White line (overrides
global)
```

RECTANGLE Statement

The `RECTANGLE` statement draws a rectangle outline or filled rectangle.

Syntax:

```
RECTANGLE FROM (x1%, y1%) TO (x2%, y2%)
RECTANGLE FROM (x1%, y1%) TO (x2%, y2%) WITH color%
RECTANGLE FROM (x1%, y1%) TO (x2%, y2%) FILLED
RECTANGLE FROM (x1%, y1%) TO (x2%, y2%) WITH color% FILLED
```

Rules:

- Draws a rectangle from point (x1%, y1%) to point (x2%, y2%)
- Default is outline only
- `FILLED` draws a filled rectangle
- Uses global foreground color (set with `COLOR`) by default
- `WITH color%` overrides the global color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format
- Coordinates are in graphics pixels (0-based, bottom-left origin)

Examples:

```
COLOR &HFFFFFFFF    ' Set global color to white
RECTANGLE FROM (50, 50) TO (150, 150)    ' White outline (uses global color)
RECTANGLE FROM (200, 200) TO (300, 300) WITH &HFF0000FF FILLED    ' Filled red
rectangle

RECTANGLE FROM (100, 100) TO (200, 200) FILLED    ' Filled rectangle (uses
global color)
```

OVAL Statement

The `OVAL` statement draws an oval (ellipse) outline or filled oval.

Syntax:

```
OVAL AT (x%, y%) WIDTH width# HEIGHT height#
OVAL AT (x%, y%) WIDTH width# HEIGHT height# WITH color%
OVAL AT (x%, y%) WIDTH width# HEIGHT height# FILLED
OVAL AT (x%, y%) WIDTH width# HEIGHT height# WITH color% FILLED
```

Rules:

- Center point is at (x%, y%)
- Width and height are real numbers (can be fractional)
- Default is outline only
- `FILLED` draws a filled oval
- Uses global foreground color (set with `COLOR`) by default
- `WITH color%` overrides the global color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format
- Coordinates are in graphics pixels (0-based, bottom-left origin)
- A circle is a special case where width equals height

Examples:

```
COLOR &HFFFF00FF    ' Set global color to yellow
OVAL AT (320, 240) WIDTH 100 HEIGHT 100    ' Yellow circle outline (uses global
color)

OVAL AT (100, 100) WIDTH 60 HEIGHT 60 WITH &H00FF00FF FILLED    ' Filled green
circle

OVAL AT (200, 200) WIDTH 80 HEIGHT 40 WITH &HFF00FFFF FILLED    ' Filled ellipse
(2:1 width:height)
```

CIRCLE Statement

The `CIRCLE` statement is a convenience alias for `OVAL` when width equals height (a circle).

Syntax:

```
CIRCLE AT (x%, y%) RADIUS radius#
CIRCLE AT (x%, y%) RADIUS radius# WITH color%
CIRCLE AT (x%, y%) RADIUS radius# FILLED
CIRCLE AT (x%, y%) RADIUS radius# WITH color% FILLED
```

Rules:

- Center point is at (x%, y%)
- Radius is a real number (can be fractional)
- Equivalent to `OVAL AT (x%, y%) WIDTH (radius# * 2) HEIGHT (radius# * 2)`
- Default is outline only
- `FILLED` draws a filled circle
- Uses global foreground color (set with `COLOR`) by default
- `WITH color%` overrides the global color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format
- Coordinates are in graphics pixels (0-based, bottom-left origin)

Examples:

```
COLOR &HFFFF00FF    ' Set global color to yellow
CIRCLE AT (320, 240) RADIUS 50    ' Yellow circle outline (uses global color)

CIRCLE AT (100, 100) RADIUS 30 WITH &H00FF00FF FILLED    ' Filled green circle

CIRCLE AT (200, 200) RADIUS 40 FILLED    ' Filled circle (uses global color)
```

ARC Statement

The `ARC` statement draws an arc (a portion of a circle) between two angles.

Syntax:

```
ARC AT (x%, y%) RADIUS radius# FROM angle1# TO angle2#
ARC AT (x%, y%) RADIUS radius# FROM angle1# TO angle2# WITH color%
```

Rules:

- Center point is at (x%, y%)

- Radius is a real number (can be fractional)
- Angles are specified in radians
- `angle1#` is the starting angle, `angle2#` is the ending angle
- The arc is drawn counterclockwise from `angle1#` to `angle2#`
- Uses global foreground color (set with `COLOR`) by default
- `WITH color%` overrides the global color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format
- Coordinates are in graphics pixels (0-based, bottom-left origin)
- Angles use standard mathematical convention: 0 radians points right (positive x-axis), increasing counterclockwise

Examples:

```
COLOR &HFFFF00FF    ' Set global color to yellow
ARC AT (320, 240) RADIUS 50 FROM 0 TO 3.14159    ' Yellow semicircle (0 to π)


' Draw a quarter circle arc
ARC AT (100, 100) RADIUS 30 FROM 0 TO 1.5708 WITH &H00FF00FF    ' Green arc (0
to π/2)


' Draw arc using degrees converted to radians
LET startAngle# = 45 * 3.14159 / 180    ' 45 degrees in radians
LET endAngle# = 135 * 3.14159 / 180     ' 135 degrees in radians
ARC AT (200, 200) RADIUS 40 FROM startAngle# TO endAngle#
```

Note: To draw a full circle, use `CIRCLE` instead. `ARC` is for partial circles only.

TRIANGLE Statement

The `TRIANGLE` statement draws a triangle outline or filled triangle.

Syntax:

```
TRIANGLE (x1%, y1%) TO (x2%, y2%) TO (x3%, y3%)
TRIANGLE (x1%, y1%) TO (x2%, y2%) TO (x3%, y3%) WITH color%
TRIANGLE (x1%, y1%) TO (x2%, y2%) TO (x3%, y3%) FILLED
TRIANGLE (x1%, y1%) TO (x2%, y2%) TO (x3%, y3%) WITH color% FILLED
```

Rules:

- Draws a triangle with vertices at (x1%, y1%), (x2%, y2%), and (x3%, y3%)

- Default is outline only
- `FILLED` draws a filled triangle
- Uses global foreground color (set with `COLOR`) by default
- `WITH color%` overrides the global color
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format
- Coordinates are in graphics pixels (0-based, bottom-left origin)

Examples:

```
COLOR &HFFFFFFFF    ' Set global color to white
TRIANGLE (100, 100) TO (200, 200) TO (150, 250)    ' White outline (uses global
color)

TRIANGLE (50, 50) TO (150, 50) TO (100, 150) WITH &HFF0000FF FILLED    ' Filled
red triangle

TRIANGLE (200, 200) TO (300, 200) TO (250, 300) FILLED    ' Filled triangle
(uses global color)
```

PAINT Statement

The `PAINT` statement fills a bounded area with a color.

Syntax:

```
PAINT (x%, y%)
PAINT (x%, y%) WITH color%
PAINT (x%, y%) WITH color% BORDER borderColor%
```

Rules:

- Fills the area starting at point (x%, y%)
- Fills until it reaches a boundary
- Uses global foreground color (set with `COLOR`) by default
- `WITH color%` overrides the global color
- With `BORDER` , fills until it reaches pixels of the specified border color
- Without `BORDER` , fills until it reaches pixels of a different color than the starting point
- Color is a 32-bit RGBA integer in `&HRRGGBBAA` format
- Coordinates are in graphics pixels (0-based, bottom-left origin)

Examples:

```
COLOR &HFF0000FF    ' Set global color to red
PAINT (100, 100)    ' Fill area with red (uses global color)

PAINT (200, 200) WITH &H00FF00FF BORDER &H0000FFFF    ' Fill area bounded by
blue pixels
```

GET Statement

The `GET` statement captures a rectangular region of the screen into an integer array (sprite).

Syntax:

```
GET spriteArray%[] FROM (x1%, y1%) TO (x2%, y2%)
```

Rules:

- Captures the rectangular region from (x1%, y1%) to (x2%, y2%)
- Stores the sprite data in an integer array
- Array format: `[width, height, pixel1, pixel2, ...]`
  - First integer: width in pixels
  - Second integer: height in pixels
  - Remaining integers: pixel colors (32-bit RGBA, row by row)
- The array must be large enough or will be automatically sized
- Coordinates are in graphics pixels (0-based, bottom-left origin)

Examples:

```
' Capture a 32x32 sprite
DIM sprite%[32 * 32 + 2]
GET sprite%[] FROM (100, 100) TO (131, 131)

' Capture entire screen
DIM screen%[640 * 480 + 2]
GET screen%[] FROM (0, 0) TO (639, 479)
```

PUT Statement

The `PUT` statement draws a sprite (from a `GET` array) onto the screen.

Syntax:

```
PUT spriteArray%[] AT (x%, y%)
```

Rules:

- Draws the sprite at position (x%, y%)
- The sprite's bottom-left corner is positioned at (x%, y%)
- Sprite array format must match `GET` format: `[width, height, pixel1, pixel2, ...]`
- Pixels are alpha-blended automatically using the alpha channel from the sprite data
- Transparent pixels (alpha = 0) are not drawn
- Semi-transparent pixels are blended with the background
- Coordinates are in graphics pixels (0-based, bottom-left origin)

Examples:

```
' Draw sprite with automatic alpha blending
PUT sprite%[] AT (200, 150)

' Animate sprite
PUT player%[] AT (x%, y%)
```

TURTLE Statement

The `TURTLE` statement provides Logo-style turtle graphics using a simple command string. Turtles draw on the same 640×480 canvas as other graphics primitives.

Syntax:

```
TURTLE turtleNumber%, commandString$
```

Rules:

- `turtleNumber%` identifies which turtle (0, 1, 2, etc.)
- Turtles start at center (320, 240), facing up (north)
- Turtles use the current foreground color set by `COLOR`
- Pen is down by default (drawing enabled)

Turtle Commands:

Command	Description
`FD n`	Move forward `n` pixels
`BK n`	Move backward `n` pixels
`RT n`	Turn right `n` degrees
`LT n`	Turn left `n` degrees
`PU`	Pen up (stop drawing)
`PD`	Pen down (resume drawing)
`HOME`	Return to center, facing up

Examples:

```
' Draw a square
TURTLE 0, "FD 100 RT 90 FD 100 RT 90 FD 100 RT 90 FD 100"

' Draw a triangle
TURTLE 0, "FD 100 RT 120 FD 100 RT 120 FD 100"

' Draw without lines (move only)
TURTLE 0, "PU FD 50 PD FD 100"

' Multiple turtles
COLOR &HFF0000FF    ' Red
TURTLE 0, "FD 100"
COLOR &H00FF00FF    ' Green
TURTLE 1, "RT 90 FD 100"
```

Graphics Examples

Example: Drawing a Simple Scene:

```
CLS WITH &H000033FF    ' Dark blue background

' Draw ground
RECTANGLE FROM (0, 0) TO (639, 50) WITH &H00FF00FF FILLED

' Draw sun
CIRCLE AT (550, 400) RADIUS 40 WITH &HFFFF00FF FILLED

' Draw house
```

```
RECTANGLE FROM (200, 50) TO (400, 200) WITH &HFF0000FF FILLED    ' Red roof
RECTANGLE FROM (250, 50) TO (350, 150) WITH &HFFFFFFFF FILLED    ' White walls
```

Example: Sprite Animation:

```
' Capture sprite
DIM player%[32 * 32 + 2]
GET player%[] FROM (0, 0) TO (31, 31)

LET x% = 100
LET y% = 100

' Animation loop
DO
    CLS

    ' Draw sprite at current position (alpha blending handles transparency)
    PUT player%[] AT (x%, y%)

    ' Update position
    LET x% += 2
    IF x% > 600 THEN LET x% = 0

    ' Small delay
    FOR i% = 1 TO 1000
    NEXT i%
LOOP
```

Audio

EduBASIC provides audio capabilities using both Music Macro Language (MML) for note control and GRIT (Generative Random Iteration Tones) for timbre (sound character). All voices use both systems together, with ADSR envelopes controlling amplitude over time.

GRIT Overview

GRIT (Generative Random Iteration Tones) is a noise synthesis system that generates procedural audio through LFSR-based (Linear Feedback Shift Register) bitstream manipulation. GRIT is inspired by and extends the classic Atari POKEY chip's noise synthesis capabilities.

How GRIT Works:

- GRIT uses **LFSRs** (Linear Feedback Shift Registers) to generate bitstreams that create different timbres

- Each voice has **3 LFSR slots** (A, B, C) that can be enabled, combined, and configured independently
- **8 primary LFSR types** are available, ranging from simple square waves (1-bit) to complex noise (31-bit)
- LFSRs can be **combined** using logical operations (XOR, AND, OR, etc.) to create complex textures
- **Decimation** controls the stepping rate, creating pitched tones from repeating patterns
- **Clock coupling** allows one LFSR to control the timing of another, creating rhythmic patterns
- The entire configuration is encoded in a single **32-bit NoiseCode** value

What GRIT Creates:

- **Pure tones**: Square waves, pulse waves, and fundamental tones
- **Musical instruments**: Bass, leads, pads, and melodic sounds
- **Percussion**: Drums, snares, hi-hats, and impact sounds
- **Sound effects**: Engines, weapons, ambience, and classic game sounds (Pitfall!, Defender, etc.)

Key Features:

- **128 presets** (0-127) provide ready-to-use timbres organized by category
- **Custom NoiseCodes** can be created using bitwise operations for advanced sound design
- **Deterministic**: Same NoiseCode + frequency = same output (reproducible)
- **Real-time**: Suitable for games, interactive applications, and live audio

Note Numbers:

- Note numbers use MIDI standard: 0-127
- Middle C (C4) is note number 60
- Lower numbers are lower pitches, higher numbers are higher pitches

How Audio Works:

- **MML** controls frequency (which notes to play), timing, and velocity for ALL voices
- **GRIT** determines the timbre (sound character) for ALL voices via NoiseCode
- **ADSR Envelopes** control amplitude over time (Attack, Decay, Sustain, Release)
- Every voice uses MML (for notes), GRIT (for timbre), and ADSR (for amplitude shaping)
- The `VOICE` statement configures the GRIT NoiseCode and ADSR envelope for each voice
- The `PLAY` statement uses MML to specify which notes to play

- **Voice Initialization:** All 64 voices (0-63) are initialized with GRIT NoiseCode preset 0 and ADSR preset 0 by default, until explicitly configured via the `VOICE` statement

TEMPO Statement

The `TEMPO` statement sets the playback tempo for MML music.

Syntax:

```
TEMPO beatsPerMinute%
```

Rules:

- Sets the tempo in beats per minute (BPM)
- Affects all subsequent `PLAY` statements using MML
- Default tempo is typically 120 BPM
- Tempo persists until changed by another `TEMPO` statement

Examples:

```
TEMPO 120    ' Set to 120 BPM (moderate tempo)
TEMPO 60     ' Set to 60 BPM (slow)
TEMPO 180    ' Set to 180 BPM (fast)
```

VOLUME Statement

The `VOLUME` statement sets the global volume for all audio output.

Syntax:

```
VOLUME volumeLevel%
```

Rules:

- Sets the global volume level (0-127)
- 0 = silent, 127 = maximum volume
- Default volume is typically 64 (50%)
- Affects all voices regardless of their configuration (MML or GRIT)
- Volume persists until changed by another `VOLUME` statement
- Individual voice velocity (V in MML) is relative to the global volume

Examples:

```
VOLUME 127    ' Maximum volume
VOLUME 64     ' Half volume (default)
VOLUME 32     ' Quarter volume
VOLUME 0      ' Silent
```

VOICE Statement

The `VOICE` statement configures the GRIT timbre (sound character) and ADSR envelope for a voice. All voices use MML (for notes), GRIT (for timbre), and ADSR (for amplitude shaping).

Syntax:

```
VOICE voiceNumber% PRESET noisePreset%
VOICE voiceNumber% WITH noiseCode%
VOICE voiceNumber% PRESET noisePreset% ADSR attack# decay# sustain# release#
VOICE voiceNumber% WITH noiseCode% ADSR attack# decay# sustain# release#
VOICE voiceNumber% PRESET noisePreset% ADSR PRESET adsrPreset%
VOICE voiceNumber% WITH noiseCode% ADSR PRESET adsrPreset%
```

Rules:

- `voiceNumber%` is an integer identifying the voice (0-63, for 64 total voices)
- **Voice Initialization:** All 64 voices are initialized with GRIT NoiseCode preset 0 and ADSR preset 0 by default
- Voices retain their configuration until explicitly changed by another `VOICE` statement
- All voices use GRIT for timbre, MML for notes, and ADSR for amplitude shaping
- **NoiseCode Configuration:**
  - `PRESET noisePreset%` : Use a preset from the 128-entry GRIT preset table (0-127)
  - `WITH noiseCode%` : Use a custom NoiseCode value (32-bit unsigned integer bitfield)

GRIT NoiseCode Presets:

The 128 GRIT presets (0-127) are organized into 8 categories, each optimized for specific sound types:

Preset Range	Category	Description	Example Use Cases
0-15	Pure Tones	Square waves, pulse waves, fundamental tones	Melodies, simple leads, classic game sounds

Preset Range	Category	Description	Example Use Cases
16-31	Bass Instruments	Deep, low-frequency sounds	Basslines, low-end instruments
32-47	Lead Instruments	Melodic lead sounds and arpeggios	Lead melodies, solos, arpeggiated patterns
48-63	Drums & Percussion	Kick drums, snares, hi-hats, percussion	Rhythm sections, drum patterns
64-79	Classic Game Sounds	Pitfall!, Defender, and other classic game sounds	Retro game sound effects, nostalgic tones
80-95	Engines & Motors	Rhythmic mechanical sounds	Engine sounds, motor effects, mechanical textures
96-111	Weapons & Impacts	Sharp, aggressive sounds	Weapon fire, impacts, explosions, sharp effects
112-127	Ambience & Textures	Background textures and atmospheric sounds	Ambient backgrounds, atmospheric layers, textures

Common Preset Examples:

- **Preset 0**: Pure Square 50/50 - Classic square wave, good for melodies
- **Preset 5**: Pulse Decim 2 - Pulse wave with decimation, versatile
- **Preset 10**: 9-bit Decim 8 - Classic Pitfall! style tone
- **Preset 48**: Kick Drum - Deep kick drum sound
- **Preset 64**: Pitfall! - Classic Pitfall! game sound
- **Preset 80**: Engine - Rhythmic engine sound
- **Preset 96**: Weapon - Sharp weapon sound
- **Preset 112**: Wind - Ambient wind texture

Note: Each preset is a single 32-bit NoiseCode value optimized for its category. Presets can be used directly or combined with different ADSR envelopes to create variations.

Custom NoiseCode Bit Layout:

For advanced users who want to create custom NoiseCode values, the 32-bit integer is organized as follows:

Bits	Field	Range	Description
31-30	Reserved	0	Must be 0
29-27	DECIM	0-7	Decimation factor (actual = value + 1, so 0 = ÷1, 7 = ÷8)
26-24	SHAPE	0-7	Wave shaping mode (0=none, 1-7=AM, Ring, etc.)
23-21	COMB	0-7	Combination operation (0=XOR, 1=AND, 2=OR, 3=NAND, 4=NOR, 5=XNOR, etc.)
20-18	C_POLY	0-7	LFSR C polynomial (0=1-bit square, 1=25/75 pulse, 2=4-bit, 3=5-bit, 4=9-bit, 5=15-bit, 6=17-bit, 7=31-bit)
17-15	B_POLY	0-7	LFSR B polynomial (same as C_POLY)
14-12	A_POLY	0-7	LFSR A polynomial (same as C_POLY)
11	C_CLK_B	0-1	Clock coupling: LFSR C clocks LFSR B
10	B_CLK_A	0-1	Clock coupling: LFSR B clocks LFSR A
9	C_INV	0-1	Invert LFSR C output
8	B_INV	0-1	Invert LFSR B output
7	A_INV	0-1	Invert LFSR A output
6	C_EN	0-1	Enable LFSR C
5	B_EN	0-1	Enable LFSR B
4	A_EN	0-1	Enable LFSR A
3-0	Reserved	0	Must be 0

NoiseCode Construction Formula:

```
NoiseCode = (DECIM << 27) | (SHAPE << 24) | (COMB << 21) |
            (C_POLY << 18) | (B_POLY << 15) | (A_POLY << 12) |
            (C_CLK_B << 11) | (B_CLK_A << 10) |
            (C_INV << 9) | (B_INV << 8) | (A_INV << 7) |
            (C_EN << 6) | (B_EN << 5) | (A_EN << 4)
```

Example: Creating a Custom NoiseCode

```
' Create a custom NoiseCode: 9-bit LFSR A enabled, decimation ÷8, no shaping
LET decim% = 7          ' ÷8 (value 7 = decimation 8)
LET shape% = 0          ' No shaping
LET comb% = 0           ' XOR combination
LET aPoly% = 4          ' 9-bit LFSR
LET bPoly% = 0          ' Not used
LET cPoly% = 0          ' Not used
LET cClkB% = 0          ' No clock coupling
LET bClkA% = 0          ' No clock coupling
LET cInv% = 0           ' No inversion
LET bInv% = 0           ' No inversion
LET aInv% = 0           ' No inversion
LET cEn% = 0            ' LFSR C disabled
LET bEn% = 0            ' LFSR B disabled
LET aEn% = 1            ' LFSR A enabled

LET customNoise% = (decim% << 27) OR (shape% << 24) OR (comb% << 21) OR _
                    (cPoly% << 18) OR (bPoly% << 15) OR (aPoly% << 12) OR _
                    (cClkB% << 11) OR (bClkA% << 10) OR _
                    (cInv% << 9) OR (bInv% << 8) OR (aInv% << 7) OR _
                    (cEn% << 6) OR (bEn% << 5) OR (aEn% << 4)

' This creates NoiseCode 0x38004010 (same as preset 10: 9-bit Decim 8)
VOICE 0 WITH customNoise% ADSR PRESET 0
```

Common Patterns:

- **Simple square wave**: Enable only LFSR A with polynomial 0 (1-bit square), no decimation
  - `LET simpleSquare% = &H00000010` (same as preset 0)
- **Pulse wave**: Enable only LFSR A with polynomial 1 (25/75 pulse), no decimation
  - `LET pulseWave% = &H00001010` (same as preset 1)
- **Pitched tone**: Use 9-bit LFSR (polynomial 4) with decimation ÷8
  - `LET pitchedTone% = &H38004010` (same as preset 10)
- **Noise texture**: Enable multiple LFSRs (A and B) with XOR combination
  - `LET noiseTexture% = &H00004010` (17-bit LFSRs with XOR)

- **ADSR Envelope Configuration:**
  - `ADSR attack# decay# sustain# release#` : Configure amplitude envelope with custom values
    - `attack#` : Time to reach full amplitude from zero (typically 0.0-2.0 seconds)
    - `decay#` : Time to fall from full amplitude to sustain level (typically 0.0-2.0 seconds)
    - `sustain#` : Level to hold after decay (0.0-1.0, where 1.0 = full amplitude)
    - `release#` : Time to fall from sustain level to zero (typically 0.0-5.0 seconds)
  - `ADSR PRESET adsrPreset%` : Use a preset from the 16-entry ADSR preset table (0-15)
- If ADSR is not specified in a `VOICE` statement, the voice retains its current ADSR configuration (or uses ADSR preset 0 if never set)
- All voices use MML (via `PLAY`) to control which notes to play and when
- Each voice operates independently
- Voices can be used in `PLAY` statements immediately (they default to preset 0 for both NoiseCode and ADSR)

Examples:

```
' All voices start with NoiseCode preset 0 and ADSR preset 0 by default
' You can use them immediately without configuration:
PLAY 0, "CDEFGAB C"    ' Voice 0 uses default presets (NoiseCode 0, ADSR 0)

' Create voice with GRIT preset (keeps current ADSR, or uses ADSR preset 0 if
never set)
VOICE 0 PRESET 5

' Create voice with GRIT preset and ADSR preset
VOICE 1 PRESET 48 ADSR PRESET 1    ' Drum sound with percussive envelope

' Create voice with custom GRIT NoiseCode and ADSR preset
LET myNoiseCode% = &H12345678
VOICE 2 WITH myNoiseCode% ADSR PRESET 3    ' Custom timbre with pad envelope

' Create voice with GRIT preset and custom ADSR envelope
VOICE 3 PRESET 10 ADSR 0.01 0.1 0.0 0.1    ' Fast attack, no sustain
(percussive)

' Create voice with custom NoiseCode and custom ADSR envelope
VOICE 4 WITH &H00000010 ADSR 0.1 0.2 0.8 0.5    ' Slow attack, high sustain
(pad)
```

```
' Create voice with GRIT preset and ambient ADSR preset
VOICE 5 PRESET 112 ADSR PRESET 4    ' Ambient sound with long release
```

ADSR Envelope Configuration

ADSR (Attack, Decay, Sustain, Release) envelopes control how the amplitude of a voice changes over time. This enables percussive sounds (fast attack, no sustain), sustained tones (slow attack, high sustain), and smooth fade-outs (long release).

Envelope Parameters:

- **Attack**: Time in seconds to reach full amplitude from zero. Fast attack (0.01-0.1s) for percussive sounds, slow attack (0.1-1.0s) for pads and ambient sounds.
- **Decay**: Time in seconds to fall from full amplitude to sustain level. Fast decay (0.05-0.2s) for drums, slower decay (0.2-1.0s) for sustained instruments.
- **Sustain**: Level to hold after decay (0.0-1.0, where 1.0 = full amplitude). 0.0 for percussive sounds, 0.5-0.9 for sustained tones.
- **Release**: Time in seconds to fall from sustain level to zero when note ends. Short release (0.1-0.5s) for staccato, long release (1.0-5.0s) for legato and ambient sounds.

ADSR Presets:

EduBASIC provides 16 ADSR envelope presets (0-15) optimized for different musical contexts:

Preset	Name	Attack	Decay	Sustain	Release	Use Case
0	Default	0.01	0.1	0.7	0.2	General purpose, sustained
1	Percussive	0.01	0.1	0.0	0.1	Drums, kicks, percussive
2	Pluck	0.01	0.2	0.5	0.5	Plucked strings, guitar
3	Pad	0.3	0.5	0.8	1.5	Pads, strings, sustained
4	Ambient	1.0	0.5	0.6	4.0	Ambient textures, long fade
5	Staccato	0.01	0.05	0.0	0.1	Short, sharp notes
6	Legato	0.1	0.2	0.9	2.0	Smooth, connected notes

Preset	Name	Attack	Decay	Sustain	Release	Use Case
7	Bell	0.01	0.3	0.3	1.0	Bells, chimes, metallic
8	Brass	0.05	0.2	0.6	0.3	Brass instruments
9	String	0.2	0.4	0.7	1.0	String instruments
10	Organ	0.01	0.1	1.0	0.1	Organ, sustained tones
11	Piano	0.01	0.15	0.3	0.5	Piano, keyboard
12	Synth Lead	0.01	0.1	0.8	0.3	Synth leads, bright
13	Bass	0.01	0.2	0.4	0.3	Bass instruments
14	Snare	0.001	0.05	0.0	0.2	Snare drums, sharp attack
15	Hi-Hat	0.001	0.01	0.0	0.05	Hi-hats, very short

Default Envelope:
If ADSR is not specified in the `VOICE` statement, ADSR preset 0 (Default) is used.

Examples:

```
' Use ADSR preset with GRIT preset
VOICE 0 PRESET 48 ADSR PRESET 1    ' Drum with percussive envelope

' Use ADSR preset with custom NoiseCode
LET myNoise% = &H12345678
VOICE 1 WITH myNoise% ADSR PRESET 3    ' Custom timbre with pad envelope

' Use custom ADSR values
VOICE 2 PRESET 32 ADSR 0.3 0.5 0.8 1.5    ' Pad sound with custom envelope

' Combine GRIT preset with ambient ADSR preset
VOICE 3 PRESET 112 ADSR PRESET 4    ' Ambient sound with long release
```

PLAY Statement

The `PLAY` statement plays music or sound on a specific voice. **All `PLAY` statements play music in the background**—execution continues immediately without waiting for the music to finish.

Syntax:

```
PLAY voiceNumber%, mmlString$
```

Rules:

- `voiceNumber%` must be a voice created with `VOICE` statement
- `mmlString$` contains the Music Macro Language sequence
- MML controls the frequency (notes), timing, and velocity for ALL voices
- The voice's timbre (configured by `VOICE` NoiseCode) determines what the notes sound like
- The voice's ADSR envelope (configured by `VOICE` ADSR) controls amplitude shaping
- Each `PLAY` statement plays on the specified voice
- Multiple voices can play simultaneously
- **All `PLAY` statements execute asynchronously in the background**—the program continues immediately
- Velocity in MML (`V` command) is relative to the global `VOLUME` setting

MML Syntax Reference

Music Macro Language (MML) controls frequency (notes), timing, and velocity for all voices. MML is used within `PLAY` statements to specify musical sequences.

Notes:

- **Note Names:** `C` , `D` , `E` , `F` , `G` , `A` , `B` (or lowercase `c` , `d` , `e` , `f` , `g` , `a` , `b`)
- **Sharps:** `#` after note name (e.g., `C#` = C sharp)
- **Flats:** `b` after note name (e.g., `Bb` = B flat) - note: lowercase `b` is both the B note and the flat symbol
- **Note Numbers:** `N` followed by MIDI note number (0-127), where 60 = Middle C (C4)

Octaves:

- `O` followed by number (0-9) sets the octave for subsequent notes
- Default octave is typically 4 (middle octave)
- Higher numbers = higher octaves

Note Length:

- `L` followed by number sets the default note length
  - `L1` = whole note

  - `L2` = half note
  - `L4` = quarter note (common default)
  - `L8` = eighth note
  - `L16` = sixteenth note
  - `L32` = thirty-second note
- Individual notes can override length: `C4` = C note with length 4
- **Dotted Notes:** `.` after note adds half the duration (e.g., `C4.` = dotted quarter note)

Rests:

- `R` followed by length creates a rest (silence)
- Example: `R4` = quarter note rest

Ties:

- `&` connects notes together (e.g., `C4&F4` plays C then F without gap)

Velocity:

- `V` followed by number (0-127) sets the velocity (loudness) for subsequent notes
- Default velocity is typically 64 (50%)
- Velocity is relative to the global `VOLUME` setting
- Velocity persists until changed by another `V` command
- Higher velocity values produce louder notes (within the limits of the global volume)
- Example: `V100 C4 D4 V64 E4` plays C and D at velocity 100 (louder), then E at velocity 64 (softer)
- Example: `V127 C D E` plays all three notes at maximum velocity
- Example: `V0 C D E` plays all three notes silently (velocity 0)

Checking Remaining Notes:

- Use the `NOTES` operator to check how many notes remain for a voice
- Syntax: `NOTES voiceNumber%`
- Returns the number of notes remaining in the voice's playback queue
- Returns 0 when the voice has finished playing all notes
- Useful for checking if a voice is still playing music

Examples:

```
' Set up voices with GRIT timbres and ADSR envelopes
VOICE 0 PRESET 5 ADSR PRESET 0    ' Engine sound timbre with default ADSR
VOICE 1 PRESET 10 ADSR PRESET 1   ' Weapon sound timbre with percussive ADSR

TEMPO 120

' Play melody - MML controls notes, GRIT preset 5 controls timbre, ADSR controls
amplitude
' Note: PLAY executes in background, program continues immediately
PLAY 0, "CDEFGAB C"

' Play with velocity (V command)
PLAY 0, "V100 CDEF V64 GAB"

' Play with note numbers (MIDI)
PLAY 0, "N60 N62 N64 N65 N67"

' Check how many notes remain
LET notesLeft% = NOTES 0
IF notesLeft% > 0 THEN PRINT "Voice 0 still playing: ", notesLeft%, " notes"

' Play on different voice - same MML, different GRIT timbre and ADSR
PLAY 1, "N60 L4"    ' Play note 60 (middle C) for quarter note duration
```

MML Examples

Simple Scale:

```
VOICE 0 PRESET 0    ' Use GRIT preset 0 with default ADSR
TEMPO 120
PLAY 0, "CDEFGAB C"    ' MML controls notes
```

Song with Velocity:

```
VOICE 0 PRESET 0 ADSR PRESET 0    ' Use ADSR preset 0 (default)
TEMPO 100
' V command sets velocity (0-127), relative to global VOLUME
PLAY 0, "V80 C L4 D L4 E L2 V100 F L4 G L4 A L2"
```

Using Note Numbers:

```
VOICE 0 PRESET 0 ADSR PRESET 0
TEMPO 120
```

```
' Play C major chord (C=60, E=64, G=67)
PLAY 0, "N60 N64 N67 L2"
```

Multiple Voices:

```
VOICE 0 PRESET 0 ADSR PRESET 0    ' Timbre for voice 0
VOICE 1 PRESET 5 ADSR PRESET 0    ' Different timbre for voice 1
TEMPO 120

' Melody on voice 0 (plays in background)
PLAY 0, "CDEFGAB C"

' Harmony on voice 1 (plays in background, simultaneously with voice 0)
PLAY 1, "CEG CEG"

' Check remaining notes for both voices
LET notes0% = NOTES 0
LET notes1% = NOTES 1
PRINT "Voice 0: ", notes0%, " notes remaining"
PRINT "Voice 1: ", notes1%, " notes remaining"
```

GRIT Noise Example:

```
VOICE 0 PRESET 5 ADSR PRESET 0    ' Engine sound preset
TEMPO 120

' Play noise at different pitches
PLAY 0, "N60 L4 N65 L4 N67 L4"    ' Rising pitch pattern
```

Audio Examples

Example: Simple Melody:

```
VOICE 0 PRESET 0    ' Use GRIT preset 0 with default ADSR
TEMPO 120
PLAY 0, "C D E F G A B C"    ' MML controls notes
```

Example: Song with Dynamics (Velocity):

```
VOICE 0 PRESET 0 ADSR PRESET 0
TEMPO 100

' V command sets velocity (0-127), relative to global VOLUME
' Velocity affects loudness of individual notes
PLAY 0, "V64 C L4 V80 D L4 V96 E L4 V112 F L2"
```

Example: Different GRIT Timbres with ADSR Presets:

```
' Create voices with different GRIT timbres and ADSR preset envelopes
VOICE 0 PRESET 10 ADSR PRESET 1    ' Weapon sound (percussive ADSR)
VOICE 1 PRESET 15 ADSR PRESET 0    ' Engine sound (default ADSR)
VOICE 2 PRESET 20 ADSR PRESET 4    ' Ambience (ambient ADSR)

TEMPO 120

' Play notes with weapon sound timbre (fast attack, no sustain)
PLAY 0, "N80 L8"

' Play notes with engine sound timbre (sustained)
PLAY 1, "N40 L1"    ' Low rumble

' Play notes with ambience timbre (slow fade-in, long fade-out)
PLAY 2, "N50 L1"    ' Background texture
```

Example: Custom GRIT Voice with ADSR Preset:

```
' Create custom GRIT voice with NoiseCode and ADSR preset
LET customNoise% = &HABCD1234
VOICE 0 WITH customNoise% ADSR PRESET 3    ' Custom timbre with pad ADSR preset

TEMPO 120
PLAY 0, "N60 L4 N65 L4 N67 L4"
```

Example: Drum Pattern with Percussive ADSR Preset:

```
' Drum sound: use ADSR preset 1 (percussive)
VOICE 0 PRESET 48 ADSR PRESET 1
TEMPO 120

' Play kick drum pattern
PLAY 0, "N36 L4 R4 N36 L4 R4 N36 L8 N36 L8 R4"
```

Example: Pad Sound with Pad ADSR Preset:

```
' Pad sound: use ADSR preset 3 (pad)
VOICE 1 PRESET 32 ADSR PRESET 3
TEMPO 100
```

```
' Play sustained chord
PLAY 1, "N60 N64 N67 L1"    ' C major chord, whole note
```

Example: Combining GRIT and ADSR Presets:

```
' Use different combinations of presets
VOICE 0 PRESET 48 ADSR PRESET 1    ' Drum timbre with percussive envelope
VOICE 1 PRESET 32 ADSR PRESET 3    ' Lead timbre with pad envelope
VOICE 2 PRESET 112 ADSR PRESET 4   ' Ambient timbre with ambient envelope

TEMPO 120
PLAY 0, "N36 L4"    ' Kick drum
PLAY 1, "N60 L2"    ' Sustained pad
PLAY 2, "N50 L1"    ' Ambient texture
```

Example: Custom ADSR Values:

```
' Use custom ADSR values when presets don't fit
VOICE 0 PRESET 5 ADSR 0.005 0.08 0.0 0.08    ' Very fast, custom percussive
VOICE 1 PRESET 32 ADSR 0.4 0.6 0.85 2.0     ' Very slow, custom pad

TEMPO 120
PLAY 0, "N60 L4"
PLAY 1, "N64 L2"
```

Statement and Operator Reference

This section provides an alphabetical reference of all EduBASIC statements, mathematical functions, and operators.

ACOS

Type: Function (Trigonometric)
Syntax: `ACOS x#`
Description: Returns the arccosine of `x` in radians. Input must be in range [-1, 1].
Example:

```
LET angle# = ACOS 0.5
PRINT angle#    ' Prints: 1.047... (π/3 radians)
```

ACOSH

Type: Function (Hyperbolic)
Syntax: `ACOSH x#`
Description: Returns the inverse hyperbolic cosine of `x` . Input must be ≥ 1.
Example:

```
LET result# = ACOSH 2.0
```

AND

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 AND expression2`
Description: Bitwise AND operation. Output bit is 1 when both input bits are 1.
Example:

```
LET result% = 12 AND 10    ' Binary: 1100 AND 1010 = 1000 (8)
IF (x% > 0) AND (y% < 10) THEN PRINT "Valid"
```

ASIN

Type: Function (Trigonometric)
Syntax: `ASIN x#`
Description: Returns the arcsine of `x` in radians. Input must be in range [-1, 1].
Example:

```
LET angle# = ASIN 0.5
PRINT angle#    ' Prints: 0.524... (π/6 radians)
```

ASINH

Type: Function (Hyperbolic)
Syntax: `ASINH x#`
Description: Returns the inverse hyperbolic sine of `x` .
Example:

```
LET result# = ASINH 1.0
```

ASC

Type: Operator (String)
Syntax: `ASC string$`
Description: Returns the ASCII code of the first character in the string.
Example:

```
LET code% = ASC "A"        ' 65
LET code% = ASC "a"        ' 97
```

ATAN

Type: Function (Trigonometric)
Syntax: `ATAN x#`
Description: Returns the arctangent of `x` in radians.
Example:

```
LET angle# = ATAN 1.0
PRINT angle#    ' Prints: 0.785... (π/4 radians)
```

ATANH

Type: Function (Hyperbolic)
Syntax: `ATANH x#`
Description: Returns the inverse hyperbolic tangent of `x` . Input must be in range (-1, 1).
Example:

```
LET result# = ATANH 0.5
```

ARC

Type: Command (Graphics)
Syntax: `ARC AT (x%, y%) RADIUS radius# FROM angle1# TO angle2#` or `ARC AT (x%, y%) RADIUS radius# FROM angle1# TO angle2# WITH color%`
Description: Draws an arc (a portion of a circle) between two angles. Angles are in radians, with 0 pointing right (positive x-axis), increasing counterclockwise. Uses global foreground color by default, or `WITH color%` to override. Coordinates use bottom-left origin (0,0).
Example:

```
ARC AT (320, 240) RADIUS 50 FROM 0 TO 3.14159    ' Semicircle (0 to π)
ARC AT (100, 100) RADIUS 30 FROM 0 TO 1.5708 WITH &H00FF00FF    ' Green quarter
circle
```

BIN

Type: Operator (String)
Syntax: `BIN integer%`
Description: Converts an integer to its binary string representation. Negative numbers are represented using two's complement notation (32 bits for 32-bit integers).
Example:

```
LET binStr$ = BIN 10     ' "1010"
LET binStr$ = BIN 255    ' "11111111"
LET binStr$ = BIN -1     ' "11111111111111111111111111111111"
```

CABS

Type: Function (Complex)
Syntax: `CABS z&`
Description: Returns the absolute value (magnitude) of complex number `z` .
Example:

```
LET z& = 3+4i
LET magnitude# = CABS z&
PRINT magnitude#    ' Prints: 5.0
```

CALL

Type: Command (Control Flow)

Syntax: `CALL subroutineName arg1, arg2, ...`

Description: Calls a `SUB` procedure with arguments. The `CALL` keyword is optional. Parentheses are not permitted.

Example:

```
CALL DrawBox 10, 5, "*"
DrawBox 20, 3, "#"     ' Same as CALL DrawBox 20, 3, "#"
```

CARG

Type: Function (Complex)

Syntax: `CARG z&`

Description: Returns the argument (phase angle) of complex number `z` in radians.

Example:

```
LET z& = 1+1i
LET angle# = CARG z&
PRINT angle#    ' Prints: 0.785... (π/4 radians)
```

CATCH

Type: Command (Control Flow)

Syntax: `CATCH errorVariable$`

Description: Catches errors raised in a `TRY` block. The error message (a string) is assigned to `errorVariable$`. Used within `TRY...CATCH...FINALLY...END TRY` blocks.

Example:

```
TRY
    OPEN "data.txt" FOR READ AS file%
CATCH error$
    PRINT "Error: ", error$
END TRY
```

CASE

Type: Command (Control Flow)

Syntax: `CASE value` or `CASE value1, value2, ...` or `CASE IS operator value` or `CASE value1 TO value2`

Description: Defines a case within a `SELECT CASE` statement.

Example:

```
SELECT CASE grade%
    CASE 90 TO 100
        PRINT "A"
    CASE 80 TO 89
        PRINT "B"
    CASE ELSE
        PRINT "Other"
END SELECT
```

CBRT

Type: Function (Math)

Syntax: `CBRT x#`

Description: Returns the cube root of `x` .

Example:

```
LET result# = CBRT 27
PRINT result#    ' Prints: 3.0
```

CHR

Type: Operator (String)

Syntax: `CHR code%`

Description: Returns the character corresponding to the ASCII code.

Example:

```
LET char$ = CHR 65          ' "A"
LET newline$ = CHR 10       ' Newline character
```

CIRCLE

Type: Command (Graphics)

Syntax: `CIRCLE AT (x%, y%) RADIUS radius#` or `CIRCLE AT (x%, y%) RADIUS radius# WITH color%` or `CIRCLE AT (x%, y%) RADIUS radius# FILLED` or `CIRCLE AT (x%, y%) RADIUS radius# WITH color% FILLED`

Description: Convenience alias for `OVAL` when width equals height (a circle). Uses global foreground color by default, or `WITH color%` to override. Default is outline only, use `FILLED` for filled circle. Coordinates use bottom-left origin (0,0).

Example:

```
COLOR &HFFFF00FF    ' Set global color to yellow
CIRCLE AT (320, 240) RADIUS 50    ' Yellow circle outline (uses global color)
CIRCLE AT (100, 100) RADIUS 30 WITH &H00FF00FF FILLED    ' Filled green circle
CIRCLE AT (200, 200) RADIUS 40 FILLED    ' Filled circle (uses global color)
```

CEIL

Type: Operator (Rounding)

Syntax: `CEIL x#`

Description: Rounds `x` toward positive infinity (+∞). Returns the smallest integer ≥ `x` .

Example:

```
PRINT CEIL 3.2     ' Prints: 4
PRINT CEIL -3.7    ' Prints: -3
```

CONJ

Type: Function (Complex)

Syntax: `CONJ z&`

Description: Returns the complex conjugate of `z` (negates the imaginary part).

Example:

```
LET z& = 3+4i
LET conjugate& = CONJ z&
PRINT conjugate&    ' Prints: 3-4i
```

CONTINUE

Type: Command (Control Flow)

Syntax: `CONTINUE FOR` or `CONTINUE WHILE` or `CONTINUE DO`

Description: Skips the rest of the current loop iteration and continues with the next iteration.

Example:

```
FOR i% = 1 TO 10
    IF i% MOD 2 = 0 THEN CONTINUE FOR     ' Skip even numbers
    PRINT i%    ' Only prints odd numbers
NEXT i%
```

COS

Type: Function (Trigonometric)

Syntax: `COS x#`

Description: Returns the cosine of `x` (where `x` is in radians).

Example:

```
LET result# = COS 0
PRINT result#    ' Prints: 1.0
```

COSH

Type: Function (Hyperbolic)

Syntax: `COSH x#`

Description: Returns the hyperbolic cosine of `x` .

Example:

```
LET result# = COSH 0
PRINT result#    ' Prints: 1.0
```

CLS

Type: Command (Graphics)

Syntax: `CLS` or `CLS WITH backgroundColor%`

Description: Clears the graphics screen. Without a color, clears to black (&H000000FF). With a color, clears to the specified background color. Does not affect the text display.

Example:

```
CLS    ' Clear to black
CLS WITH &H000033FF    ' Clear to dark blue
```

CLOSE

Type: Command (File I/O)

Syntax: `CLOSE fileHandle%`

Description: Closes an open file.

Example:

```
OPEN "data.txt" FOR READ AS file%
' ... read operations ...
CLOSE file%
```

COLOR

Type: Command (Text/Graphics)

Syntax: `COLOR foregroundColor%` or `COLOR foregroundColor%, backgroundColor%` or `COLOR , backgroundColor%`

Description: Sets the global foreground and/or background color for both text and graphics operations. Colors are specified as 32-bit RGBA integers in `&HRRGGBBAA` format. These colors are used by default, but can be overridden in individual statements. For text, both foreground and background can be set. With `COLOR , backgroundColor%` , only the background color is set (foreground unchanged). For graphics, only the foreground color is used (background parameter is ignored).

Example:

```
COLOR &HFF0000FF        ' Set global foreground to red
PRINT "This is red text"
COLOR &HFFFFFF00, &H000000FF  ' Transparent text on black background
COLOR , &H000000FF      ' Set only background to black (foreground unchanged)
COLOR &H00FF00FF    ' Set global foreground to green for graphics
LINE FROM (0, 0) TO (100, 100)    ' Green line (uses global color)
LINE FROM (50, 50) TO (150, 150) WITH &HFF0000FF    ' Red line (overrides
global)
```

COPY

Type: Command (File I/O)
Syntax: `COPY "source" TO "destination"`
Description: Copies a file from source to destination.
Example:

```
COPY "data.txt" TO "backup.txt"
COPY "/source/file.bin" TO "/dest/file.bin"
```

CSQRT

Type: Function (Complex)
Syntax: `CSQRT z&`
Description: Returns the complex square root of `z` .
Example:

```
LET z& = -1+0i
LET root& = CSQRT z&
PRINT root&     ' Prints: 0+1i
```

DATE$

Type: Operator (Date/Time)
Syntax: `DATE$`
Description: Returns the current date as a string in `YYYY-MM-DD` format (ISO 8601 date format).
Example:

```
LET today$ = DATE$
PRINT "Today is: ", today$    ' Prints: "Today is: 2025-01-15"

IF DATE$ = "2025-12-25" THEN
    PRINT "Merry Christmas!"
END IF
```

DEG

Type: Operator (Unit Conversion)
Syntax: `expression DEG`
Description: Postfix operator that converts radians to degrees.
Example:

```
LET degrees# = (3.14159 / 2) DEG
PRINT degrees#    ' Prints: 90.0
```

DELETE

Type: Command (File I/O)
Syntax: `DELETE "filename"`
Description: Deletes a file.
Example:

```
DELETE "temp.txt"
DELETE "/Users/name/old_data.bin"
```

DIM

Type: Statement (Array Resizing)
Syntax: `DIM arrayName[size]` or `DIM arrayName[start TO end]`
Description: Resizes an array. Arrays are one-based by default. Undeclared arrays are presumed to have size 0, so you can use `DIM` to resize them even if they haven't been assigned yet. `DIM` is **only** used for resizing arrays—no other variables need to be declared.
Example:

```
DIM numbers%[10]             ' Resize to indices 1 to 10 (creates array if it doesn't exist)
LET numbers%[] = [1, 2, 3]
DIM numbers%[20]             ' Resize existing array to indices 1 to 20
DIM studentNames$[0 TO 11]   ' Resize to indices 0 to 11
DIM matrix#[5, 10]           ' Resize to 5×10 two-dimensional array
```

DO

Type: Command (Control Flow)
Syntax: `DO WHILE condition` or `DO UNTIL condition` or `DO` with `LOOP WHILE` or `LOOP UNTIL`
Description: Begins a `DO` loop with optional condition at top or bottom.
Example:

```
DO WHILE count% < 10
    PRINT count%
    LET count% += 1
LOOP

DO
    INPUT "Enter value: ", value%
LOOP UNTIL value% = 0
```

ELSE

Type: Command (Control Flow)
Syntax: `ELSE`
Description: Defines the alternative branch in an `IF` or `UNLESS` statement.
Example:

```
IF score% >= 60 THEN
    PRINT "Pass"
ELSE
    PRINT "Fail"
END IF
```

ELSEIF

Type: Command (Control Flow)
Syntax: `ELSEIF condition THEN`
Description: Adds additional conditional branches to an `IF` statement.
Example:

```
IF age% < 13 THEN
    PRINT "Child"
ELSEIF age% < 20 THEN
    PRINT "Teenager"
ELSE
```

```
    PRINT "Adult"
END IF
```

END

Type: Command (Control Flow)
Syntax: `END`
Description: Terminates program execution immediately.
Example:

```
IF criticalError% THEN
    PRINT "Fatal error!"
    END
END IF
```

END IF

Type: Command (Control Flow)
Syntax: `END IF`
Description: Terminates an `IF` statement block.
Example:

```
IF x% > 0 THEN
    PRINT "Positive"
END IF
```

END SELECT

Type: Command (Control Flow)
Syntax: `END SELECT`
Description: Terminates a `SELECT CASE` statement block.
Example:

```
SELECT CASE grade%
    CASE 90 TO 100
```

```
        PRINT "A"
END SELECT
```

END SUB

Type: Command (Control Flow)
Syntax: `END SUB`
Description: Terminates a `SUB` procedure definition.
Example:

```
SUB PrintMessage (msg$)
    PRINT msg$
END SUB
```

END UNLESS

Type: Command (Control Flow)
Syntax: `END UNLESS`
Description: Terminates an `UNLESS` statement block.
Example:

```
UNLESS valid% THEN
    PRINT "Invalid input"
END UNLESS
```

END TRY

Type: Command (Control Flow)
Syntax: `END TRY`
Description: Terminates a `TRY...CATCH...FINALLY` block.
Example:

```
TRY
    OPEN "file.txt" FOR READ AS file%
CATCH error$
    PRINT "Error: ", error$
FINALLY
```

```
    CLOSE file%
END TRY
```

EOF

Type: Operator (File I/O)
Syntax: `EOF fileHandle%`
Description: Unary operator that checks if the file pointer is at the end of the file. Returns integer: 0 = false, -1 = true.
Example:

```
OPEN "data.txt" FOR READ AS file%
WHILE NOT EOF file%
    LINE INPUT line$ FROM #file%
    PRINT line$
WEND
CLOSE file%
```

EXIT

Type: Command (Control Flow)
Syntax: `EXIT FOR` or `EXIT WHILE` or `EXIT DO` or `EXIT SUB`
Description: Immediately exits from the specified loop or procedure.
Example:

```
FOR i% = 1 TO 100
    IF found% THEN EXIT FOR
NEXT i%

DO
    IF quit% THEN EXIT DO
LOOP
```

EXP

Type: Function (Math)
Syntax: `EXP x#`

Description: Returns e raised to the power `x` (e^x).

Example:

```
LET result# = EXP 1
PRINT result#    ' Prints: 2.718... (e)
```

EXPAND

Type: Operator (Rounding)

Syntax: `EXPAND x#`

Description: Rounds `x` away from zero.

Example:

```
PRINT EXPAND 3.1     ' Prints: 4
PRINT EXPAND -3.1    ' Prints: -4
```

E#

Type: Operator (Math Constant)

Syntax: `E#`

Description: Returns the mathematical constant e (Euler's number) as a real number (approximately 2.718281828459045).

Example:

```
LET eValue# = E#
PRINT eValue#    ' Prints: 2.718281828459045

LET exponential# = E# ^ 2    ' e^2
LET naturalLog# = LOG E#     ' Should be approximately 1.0
```

EXISTS

Type: Operator (File I/O)

Syntax: `EXISTS "filename"`

Description: Unary operator that checks if a file or directory exists. Returns integer: 0 = false,

-1 = true.

Example:

```
IF EXISTS "data.txt" THEN
    READFILE "data.txt" INTO content$
ELSE
    PRINT "File not found"
END IF
```

FINALLY

Type: Command (Control Flow)

Syntax: `FINALLY`

Description: Marks the cleanup section of a `TRY...CATCH...FINALLY...END TRY` block. The `FINALLY` block always executes, whether an error occurred or not. Optional—you can use `TRY...CATCH...END TRY` without `FINALLY` .

Example:

```
TRY
    OPEN "data.txt" FOR READ AS file%
    READFILE "data.txt" INTO content$
CATCH error$
    PRINT "Error: ", error$
FINALLY
    CLOSE file%
END TRY
```

FLOOR

Type: Operator (Rounding)

Syntax: `FLOOR x#`

Description: Rounds `x` toward negative infinity (-∞). Returns the largest integer ≤ `x` .

Example:

```
PRINT FLOOR 3.7       ' Prints: 3
PRINT FLOOR -3.2      ' Prints: -4
```

FIND

Type: Operator (Array Search)
Syntax: `array[] FIND value`
Description: Finds the first element in an array that matches a value. Returns the element value if found, or 0 (for numeric arrays) or "" (for string arrays) if not found.
Example:

```
LET found% = numbers%[] FIND 5
IF found% <> 0 THEN PRINT "Found: ", found%

LET found$ = names$[] FIND "Bob"
IF found$ <> "" THEN PRINT "Found: ", found$
```

FOR

Type: Command (Control Flow)
Syntax: `FOR variable = start TO end [STEP increment]`
Description: Begins a `FOR` loop that iterates a variable through a range.
Example:

```
FOR i% = 1 TO 10
    PRINT i%
NEXT i%

FOR x# = 10 TO 0 STEP -0.5
    PRINT x#
NEXT x#
```

GET

Type: Command (Graphics)
Syntax: `GET spriteArray%[] FROM (x1%, y1%) TO (x2%, y2%)`
Description: Captures a rectangular region of the screen into an integer array (sprite). Array format: `[width, height, pixel1, pixel2, ...]` where first integer is width, second is height, and remaining integers are pixel colors (32-bit RGBA, row by row). Coordinates use bottom-left origin (0,0).
Example:

```
DIM sprite%[32 * 32 + 2]
GET sprite%[] FROM (100, 100) TO (131, 131)
```

GOSUB

Type: Command (Control Flow)
Syntax: `GOSUB labelName`
Description: Calls a subroutine at the specified label. Use `RETURN` to return.
Example:

```
GOSUB PrintHeader
PRINT "Main program"
END

LABEL PrintHeader
    PRINT "=========="
RETURN
```

GOTO

Type: Command (Control Flow)
Syntax: `GOTO labelName`
Description: Transfers control unconditionally to the specified label.
Example:

```
LET x% = 0
LABEL Loop
    PRINT x%
    LET x% += 1
    IF x% < 5 THEN GOTO Loop
```

IF

Type: Command (Control Flow)
Syntax: `IF condition THEN statement` or `IF condition THEN ... END IF`
Description: Executes code conditionally based on a boolean expression.
Example:

```
IF score% >= 90 THEN PRINT "A"

IF temperature# > 100 THEN
    PRINT "Boiling!"
END IF
```

IMAG

Type: Function (Complex)
Syntax: `IMAG z&`
Description: Returns the imaginary part of complex number `z`.
Example:

```
LET z& = 3+4i
LET imagPart# = IMAG z&
PRINT imagPart#    ' Prints: 4.0
```

HEX

Type: Operator (String)
Syntax: `HEX integer%`
Description: Converts an integer to its hexadecimal string representation (uppercase, no prefix). Negative numbers are represented using two's complement notation (8 hex digits for 32-bit integers).
Example:

```
LET hexStr$ = HEX 255    ' "FF"
LET hexStr$ = HEX 10     ' "A"
LET hexStr$ = HEX -1     ' "FFFFFFFF"
LET hexStr$ = HEX 4095    ' "FFF"
```

IMP

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 IMP expression2`
Description: Bitwise implication. Output bit is 1 when first bit is 0 or second bit is 1.
Example:

```
LET result% = 5 IMP 3    ' Binary implication
```

INKEY$

Type: Operator (Input)
Syntax: `INKEY$`
Description: Returns a string containing the key currently pressed, or empty string if no key is pressed. Non-blocking keyboard input. Returns the character or special key name.
Return Values:

- **Empty string (`""`):** No key is currently pressed
- **Printable characters:** Returns the character as a string (e.g., `"a"` , `"A"` , `"1"` , `" "`)
- **Special keys:** Returns special key names (see table below)

Special Key Names:

Key	Return Value
Escape	`"ESC"`
Enter/Return	`"ENTER"`
Backspace	`"BACKSPACE"`
Tab	`"TAB"`
Space	`" "` (space character)
Arrow Up	`"ARROWUP"`
Arrow Down	`"ARROWDOWN"`
Arrow Left	`"ARROWLEFT"`
Arrow Right	`"ARROWRIGHT"`
Home	`"HOME"`
End	`"END"`
Page Up	`"PAGEUP"`
Page Down	`"PAGEDOWN"`
Insert	`"INSERT"`

Key	Return Value
Delete	`"DELETE"`
F1-F12	`"F1"` through `"F12"`
Shift	`"SHIFT"`
Control	`"CTRL"`
Alt	`"ALT"`
Caps Lock	`"CAPSLOCK"`

Note: Special key names are case-sensitive.

Examples:

```
' Basic usage - get character or special key name
DO
    LET key$ = INKEY$
    IF key$ <> "" THEN
        PRINT "Key pressed: ", key$
        IF key$ = "ESC" THEN EXIT DO
    END IF
LOOP

' Detect arrow keys for game movement
LET key$ = INKEY
IF key$ = "ARROWUP" THEN
    LET y% += 1
ELSEIF key$ = "ARROWDOWN" THEN
    LET y% -= 1
ELSEIF key$ = "ARROWLEFT" THEN
    LET x% -= 1
ELSEIF key$ = "ARROWRIGHT" THEN
    LET x% += 1
END IF

' Check for function keys
LET key$ = INKEY
IF key$ = "F1" THEN
    GOSUB ShowHelp
ELSEIF key$ = "F2" THEN
    GOSUB SaveGame
END IF
```

INPUT

Type: Command (Text I/O)
Syntax: `INPUT variable` or `INPUT array[]`
Description: Reads a value from the user and assigns it to a variable. The variable's type sigil determines what type of value is expected. The prompt is displayed separately using `PRINT` before the `INPUT` statement. For arrays, reads multiple comma-separated values, filling the array from index 1 onwards.
Example:

```
PRINT "Enter your age: ";
INPUT age%
PRINT "Enter your name: ";
INPUT name$

DIM scores%[5]
PRINT "Enter 5 scores (comma-separated): ";
INPUT scores%[]
```

INSTR

Type: Operator (String Search)
Syntax: `string$ INSTR substring$` or `string$ INSTR substring$ FROM start%`
Description: Finds the position of the first occurrence of `substring$` in `string$`. With `FROM`, search begins at that position. Returns 0 if not found.
Example:

```
LET pos% = "Hello world" INSTR "world"    ' 7
LET pos% = "Hello world" INSTR "o" FROM 5    ' 5 (start search at position 5)
```

INDEXOF

Type: Operator (Array Search)
Syntax: `array[] INDEXOF value`
Description: Finds the index of the first occurrence of a value in an array. Returns 0 if not found.
Example:

```
LET index% = numbers%[] INDEXOF 5
IF index% > 0 THEN PRINT "Found at index: ", index%

LET index% = names$[] INDEXOF "Bob"
IF index% > 0 THEN PRINT "Found at index: ", index%
```

INCLUDES

Type: Operator (Array Search)
Syntax: `array[] INCLUDES value`
Description: Checks if an array includes a value. Returns TRUE% (-1) if found, FALSE% (0) if not found.
Example:

```
IF numbers%[] INCLUDES 5 THEN PRINT "Found"
IF names$[] INCLUDES "Bob" THEN PRINT "Found"
```

INT

Type: Operator (Math)
Syntax: `INT x#`
Description: Converts a real number to an integer by truncating the decimal part (rounds toward zero).
Example:

```
LET dice% = INT (RND# * 6) + 1    ' Random integer 1-6
PRINT INT 3.9    ' Prints: 3
PRINT INT -3.9   ' Prints: -3
```

JOIN

Type: Operator (Array)
Syntax: `array$[] JOIN separator$`
Description: Joins array elements into a string with a separator.
Example:

```
LET names$[] = ["Alice", "Bob", "Charlie"]
LET joined$ = names$[] JOIN ", "    ' "Alice, Bob, Charlie"
```

LABEL

Type: Command (Control Flow)
Syntax: `LABEL labelName`
Description: Defines a label that can be targeted by `GOTO` or `GOSUB` .
Example:

```
LABEL StartProgram
PRINT "Starting..."

LABEL MainLoop
PRINT "Looping..."
```

LCASE

Type: Operator (String)
Syntax: `LCASE string$`
Description: Returns a copy of the string converted to lowercase.
Example:

```
LET lower$ = LCASE "WORLD"    ' "world"
```

LINE

Type: Command (Graphics)
Syntax: `LINE FROM (x1%, y1%) TO (x2%, y2%) WITH color%` or `LINE FROM (x1%, y1%) TO (x2%, y2%) WITH color% FILLED`
Description: Draws a line between two points. With `FILLED` , draws a filled rectangle. Coordinates use bottom-left origin (0,0).
Example:

```
LINE FROM (10, 10) TO (100, 50) WITH &H00FF00FF    ' Green line
LINE FROM (200, 200) TO (300, 300) WITH &HFF0000FF FILLED    ' Filled rectangle
```

LEFT

Type: Operator (String)
Syntax: `string$ LEFT length%`
Description: Returns the leftmost `length%` characters of the string. If `length%` is greater than the string length, returns the entire string.
Example:

```
LET text$ = "Hello, world!"
LET firstWord$ = text$ LEFT 5    ' "Hello"
LET firstChar$ = text$ LEFT 1    ' "H"
```

LINE INPUT

Type: Command (File I/O)
Syntax: `LINE INPUT lineVariable$ FROM #fileHandle%`
Description: Reads a complete line of text from a file, including the newline character. At end of file, an error occurs (check with `EOF` first).
Example:

```
OPEN "data.txt" FOR READ AS file%
WHILE NOT EOF file%
    LINE INPUT line$ FROM #file%
    PRINT line$
WEND
CLOSE file%
```

LISTDIR

Type: Command (File I/O)
Syntax: `LISTDIR "path" INTO filesArray$[]`
Description: Lists files in a directory and stores filenames in an array. Array is 1-based and includes files and subdirectories.
Example:

```
LISTDIR "." INTO files$[]
FOR i% = 1 TO | files$[] |
    PRINT files$[i%]
NEXT i%
```

LOC

Type: Operator (File I/O)
Syntax: `LOC fileHandle%`
Description: Unary operator that returns the current byte position in the file (0-based).
Example:

```
OPEN "data.bin" FOR READ AS file%
LET startPos% = LOC file%
READ value% FROM file%
LET endPos% = LOC file%
LET bytesRead% = endPos% - startPos%
CLOSE file%
```

LOCATE

Type: Command (Text I/O)
Syntax: `LOCATE row%, column%`
Description: Positions the text cursor at a specific row and column in the text display. Row and column are 1-based (row 1, column 1 is the top-left corner).
Example:

```
LOCATE 10, 20
PRINT "This text appears at row 10, column 20"
```

LET

Type: Statement (Variable Assignment)
Syntax: `LET variable = expression` or `LET array[] = expression`
Description: Assigns a value to a variable. Creates module-level (global) variables. Can assign to scalars or arrays. For arrays, the expression can be an array literal, another array, or an array

slice.

Example:

```
LET count% = 10
LET name$ = "Alice"
LET result# = 3.14159
LET complex& = 3+4i

' Array assignments
LET numbers%[] = [1, 2, 3, 4, 5]
LET copy%[] = numbers%[]
LET slice%[] = numbers%[2 TO 4]
```

LOCAL

Type: Statement (Variable Assignment)

Syntax: `LOCAL variable = expression` or `LOCAL array[] = expression`

Description: Creates a variable local to the current stack frame. Inside a `SUB`, the variable is scoped to that procedure's stack frame and destroyed when the `SUB` exits. Outside any `SUB`, the variable is scoped to the top-level stack frame. Can create local scalars or arrays.

Example:

```
SUB Calculate ()
    LOCAL temp% = 10     ' Local to this SUB
    LOCAL result# = 0    ' Local to this SUB
    LOCAL buffer%[] = [0, 0, 0]    ' Local array
END SUB
```

LTRIM

Type: Operator (String)

Syntax: `LTRIM string$`

Description: Returns a copy of the string with leading spaces removed.

Example:

```
LET trimmed$ = LTRIM "  text"    ' "text"
```

MID

Type: Operator (String)
Syntax: `string$ MID start% TO end%`
Description: Returns a substring from position `start%` to position `end%` (inclusive). If `start%` is beyond the string length, returns an empty string. If `end%` extends beyond the string, returns characters up to the end of the string.
Example:

```
LET text$ = "Hello, world!"
LET world$ = text$ MID 8 TO 12    ' "world" (positions 8 to 12)
LET middle$ = text$ MID 3 TO 6    ' "llo," (positions 3 to 6)
LET char$ = text$ MID 1 TO 1    ' "H" (single character at position 1)
```

LOG

Type: Function (Math)
Syntax: `LOG x#`
Description: Returns the natural logarithm (base e) of `x` .
Example:

```
LET result# = LOG 2.718
PRINT result#    ' Prints: 1.0 (approximately)
```

LOG10

Type: Function (Math)
Syntax: `LOG10 x#`
Description: Returns the common logarithm (base 10) of `x` .
Example:

```
LET result# = LOG10 100
PRINT result#    ' Prints: 2.0
```

LOG2

Type: Function (Math)
Syntax: `LOG2 x#`
Description: Returns the binary logarithm (base 2) of `x` .
Example:

```
LET result# = LOG2 8
PRINT result#    ' Prints: 3.0
```

LOOP

Type: Command (Control Flow)
Syntax: `LOOP` or `LOOP WHILE condition` or `LOOP UNTIL condition`
Description: Marks the end of a `DO` loop, with optional condition at bottom.
Example:

```
DO
    PRINT "Hello"
LOOP WHILE continue%

DO
    INPUT "Value: ", val%
LOOP UNTIL val% = 0
```

MKDIR

Type: Command (File I/O)
Syntax: `MKDIR "path"`
Description: Creates a directory.
Example:

```
MKDIR "backups"
MKDIR "/Users/name/data"
```

MOD

Type: Operator (Arithmetic)
Syntax: `expression1 MOD expression2`

Description: Returns the remainder of division. Works with both Integer and Real operands. When both operands are integers, the result is an integer. When either operand is real, the result is real.

Example:

```
LET remainder% = 17 MOD 5      ' Integer MOD Integer → Integer (2)
LET remainder# = 17.5 MOD 5   ' Real MOD Integer → Real (2.5)
LET remainder# = 17 MOD 5.5    ' Integer MOD Real → Real (0.5)
LET remainder# = 17.5 MOD 5.5 ' Real MOD Real → Real (1.0)
PRINT remainder%    ' Prints: 2
```

MOVE

Type: Command (File I/O)

Syntax: `MOVE "source" TO "destination"`

Description: Moves or renames a file.

Example:

```
MOVE "oldname.txt" TO "newname.txt"
MOVE "/temp/file.txt" TO "/final/file.txt"
```

NAND

Type: Operator (Boolean/Bitwise)

Syntax: `expression1 NAND expression2`

Description: Bitwise NAND. Output bit is 1 when at least one input bit is 0.

Example:

```
LET result% = 12 NAND 10
```

NEXT

Type: Command (Control Flow)

Syntax: `NEXT [variable]`

Description: Marks the end of a `FOR` loop. Variable name is optional.

Example:

```
FOR i% = 1 TO 10
    PRINT i%
NEXT i%

FOR j% = 1 TO 5
    PRINT j%
NEXT
```

NOR

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 NOR expression2`
Description: Bitwise NOR. Output bit is 1 when both input bits are 0.
Example:

```
LET result% = 12 NOR 10
```

NOT

Type: Operator (Boolean/Bitwise)
Syntax: `NOT expression`
Description: Bitwise NOT. Output bit is 1 when the input bit is 0.
Example:

```
LET result% = NOT 5
IF NOT gameOver% THEN GOSUB UpdateGame
```

NOW%

Type: Operator (Date/Time)
Syntax: `NOW%`
Description: Returns the current Unix timestamp (seconds since January 1, 1970, 00:00:00 UTC) as an integer. Useful for calculating time differences and storing absolute time values.
Example:

```
LET timestamp% = NOW%
PRINT "Timestamp: ", timestamp%    ' Prints: "Timestamp: 1736968245"

' Calculate elapsed time
LET startTime% = NOW%
SLEEP 2000    ' Wait 2 seconds
LET endTime% = NOW%
LET elapsed% = endTime% - startTime%
PRINT "Elapsed: ", elapsed%, " seconds"
```

OPEN

Type: Command (File I/O)
Syntax: `OPEN "filename" FOR mode AS fileHandle%`
Description: Opens a file and assigns a handle to a variable. Modes: `READ` (reading only), `APPEND` (writing, appending to end), `OVERWRITE` (writing, replacing existing content). Attempting to open a non-existent file in `READ` mode causes an error. Opening a non-existent file in `APPEND` or `OVERWRITE` mode creates the file.
Example:

```
OPEN "data.txt" FOR READ AS inputFile%
OPEN "output.txt" FOR OVERWRITE AS outputFile%
OPEN "log.txt" FOR APPEND AS logFile%
```

OR

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 OR expression2`
Description: Bitwise OR. Output bit is 1 when at least one input bit is 1.
Example:

```
LET result% = 12 OR 10    ' Binary: 1100 OR 1010 = 1110 (14)
IF (x% > 0) OR (y% < 10) THEN PRINT "Valid"
```

POP

Type: Statement (Array)
Syntax: `POP array[] INTO variable`
Description: Removes the last element from an array and assigns it to a variable.
Example:

```
POP numbers%[] INTO value%
POP names$[] INTO name$
```

PAINT

Type: Command (Graphics)
Syntax: `PAINT (x%, y%) WITH color%` or `PAINT (x%, y%) WITH color% BORDER borderColor%`
Description: Fills a bounded area with a color starting at point (x%, y%). With `BORDER`, fills until it reaches pixels of the specified border color. Without `BORDER`, fills until it reaches pixels of a different color than the starting point. Coordinates use bottom-left origin (0,0).
Example:

```
PAINT (100, 100) WITH &HFF0000FF    ' Fill area with red
PAINT (200, 200) WITH &H00FF00FF BORDER &H0000FFFF    ' Fill bounded by blue
pixels
```

PLAY

Type: Command (Audio)
Syntax: `PLAY voiceNumber%, mmlString$`
Description: Plays music or sound on a specific voice using Music Macro Language (MML). **All `PLAY` statements play music in the background**—execution continues immediately without waiting for the music to finish. `voiceNumber%` must be a voice created with `VOICE` statement. `mmlString$` contains the MML sequence that controls frequency (notes), timing, and velocity. The voice's timbre is determined by the GRIT NoiseCode configured in `VOICE`, and amplitude shaping is controlled by the ADSR envelope. Multiple voices can play simultaneously. Use `NOTES` operator to check how many notes remain for a voice. Velocity in MML (`V` command) is relative to the global `VOLUME` setting.
Example:

```
VOICE 0 PRESET 5 ADSR 0.01 0.1 0.7 0.2
TEMPO 120
```

```
PLAY 0, "CDEFGAB C"    ' Play scale (continues in background)
PLAY 0, "V100 C L4 V64 D L4"    ' Play with velocity (continues in background)
PLAY 0, "N60 N64 N67 L2"    ' Play chord using MIDI note numbers (continues in
background)

' Check how many notes remain
LET notesLeft% = NOTES 0
IF notesLeft% > 0 THEN PRINT "Still playing: ", notesLeft%, " notes remaining"
```

PI#

Type: Operator (Math Constant)

Syntax: `PI#`

Description: Returns the mathematical constant π (pi) as a real number (approximately 3.141592653589793).

Example:

```
LET piValue# = PI#
PRINT piValue#    ' Prints: 3.141592653589793

LET circumference# = 2 * PI# * radius#
LET area# = PI# * radius# * radius#
LET halfCircle# = PI# * radius#
```

PRINT

Type: Command (Text I/O)

Syntax: `PRINT expression1, expression2, ...` or `PRINT expression1, expression2, ...;` or `PRINT array[]` or `PRINT array[];` or `PRINT`

Description: Outputs text and values to the text display. The `PRINT` statement accepts expressions of any type (Integer, Real, Complex, String, Array) separated by commas. Each expression is automatically converted to its string representation and concatenated with no spacing between items. Comma (`,`) separates items with no spacing (concatenated). Semicolon (`;`) at the end suppresses the newline. Empty `PRINT` outputs a blank line. When printing an array, all elements are printed in brackets with spaces after commas (e.g., `[1, 2, 3, 4, 5]`).

Example:

```
PRINT "Hello, world!"
PRINT "Name: ", name$, " Age: ", age%
PRINT "X: ", x%, " Y: ", y%

' Mixed types
PRINT "Item: ", name$, " Count: ", count%, " Price: ", price#, " Complex: ", z&
' Prints: Item: Widget Count: 10 Price: 19.99 Complex: 3+4i

PRINT numbers%[]    ' Prints: [1, 2, 3, 4, 5]
PRINT "Scores: ", scores%[]    ' Prints: Scores: [85, 90, 78]
PRINT "Enter name: ";    ' No newline, continues on same line
PRINT    ' Blank line

' String concatenation alternative
PRINT "Name: " + name$ + " Age: " + STR age%    ' With explicit spacing
```

PSET

Type: Command (Graphics)
Syntax: `PSET (x%, y%) WITH color%`
Description: Sets a single pixel to a specified color. Coordinates use bottom-left origin (0,0). X ranges from 0 to 639, Y ranges from 0 to 479. Color is a 32-bit RGBA integer.
Example:

```
PSET (100, 200) WITH &HFF0000FF    ' Red pixel
FOR i% = 0 TO 639
    PSET (i%, 240) WITH &HFFFFFFFF    ' White horizontal line
NEXT i%
```

PUT

Type: Command (Graphics)
Syntax: `PUT spriteArray%[] AT (x%, y%)`
Description: Draws a sprite (from a `GET` array) onto the screen. The sprite's bottom-left corner is positioned at (x%, y%). Pixels are alpha-blended automatically using the alpha channel from the sprite data. Transparent pixels (alpha = 0) are not drawn. Coordinates use bottom-left origin (0,0).
Example:

```
PUT sprite%[] AT (200, 150)    ' Draw sprite with automatic alpha blending
PUT player%[] AT (x%, y%)    ' Animate sprite
```

PUSH

Type: Statement (Array)
Syntax: `PUSH array[], value`
Description: Adds an element to the end of an array.
Example:

```
PUSH numbers%[], 10
PUSH names$[], "David"
```

RAD

Type: Operator (Unit Conversion)
Syntax: `expression RAD`
Description: Postfix operator that converts degrees to radians.
Example:

```
LET radians# = 90 RAD
PRINT radians#    ' Prints: 1.5708... (π/2 radians)
```

RANDOMIZE

Type: Command (Random)
Syntax: `RANDOMIZE` or `RANDOMIZE seed%`
Description: Seeds the random number generator. Without argument, uses `NOW%` .
Example:

```
RANDOMIZE              ' Seed with current timestamp (NOW%)
RANDOMIZE 12345        ' Seed with specific value
RANDOMIZE NOW%         ' Explicitly seed with current timestamp
```

NOTES

Type: Operator (Audio)
Syntax: `NOTES voiceNumber%`

Description: Returns the number of notes remaining in the playback queue for the specified voice. Returns 0 when the voice has finished playing all notes. Useful for checking if a voice is still playing music.

Example:

```
PLAY 0, "CDEFGAB C"
LET notesLeft% = NOTES 0
IF notesLeft% > 0 THEN PRINT "Voice 0 still playing: ", notesLeft%, " notes
remaining"

' Wait for voice to finish
DO
    LET notesLeft% = NOTES 0
LOOP WHILE notesLeft% > 0
PRINT "Voice 0 finished playing"
```

READ

Type: Command (File I/O)

Syntax: `READ variable FROM fileHandle%` or `READ array[] FROM fileHandle%`

Description: Reads binary data from a file based on the variable's type. Integer: reads 4 bytes (32-bit signed integer). Real: reads 8 bytes (64-bit floating-point). Complex: reads 16 bytes (128-bit complex number). String: reads the string's length prefix and data. For arrays, reads elements sequentially, filling the array from index 1 onwards. Reading advances the file position. At end of file, an error occurs (check with `EOF` first).

Example:

```
OPEN "data.bin" FOR READ AS file%
READ count% FROM file%
DIM numbers%[count%]
FOR i% = 1 TO count%
    READ numbers%[i%] FROM file%
NEXT i%
CLOSE file%

' Or read entire array at once
DIM data%[100]
READ data%[] FROM file%
```

READFILE

Type: Command (File I/O)

Syntax: `READFILE "filename" INTO contentVariable$`

Description: Reads an entire file into a string variable.

Example:

```
READFILE "config.txt" INTO config$
PRINT config$
READFILE "data.json" INTO jsonData$
```

REAL

Type: Function (Complex)

Syntax: `REAL z&`

Description: Returns the real part of complex number `z` .

Example:

```
LET z& = 3+4i
LET realPart# = REAL z&
PRINT realPart#    ' Prints: 3.0
```

REPLACE

Type: Operator (String)

Syntax: `string$ REPLACE oldSubstring$ WITH newSubstring$`

Description: Returns a copy of the string with all occurrences of `oldSubstring$` replaced with `newSubstring$` .

Example:

```
LET new$ = "Hello world" REPLACE "world" WITH "EduBASIC"    ' "Hello EduBASIC"
```

RECTANGLE

Type: Command (Graphics)

Syntax: `RECTANGLE FROM (x1%, y1%) TO (x2%, y2%)` or `RECTANGLE FROM (x1%, y1%) TO (x2%, y2%) WITH color%` or `RECTANGLE FROM (x1%, y1%) TO (x2%, y2%) FILLED` or `RECTANGLE FROM (x1%, y1%) TO (x2%, y2%) WITH color% FILLED`

Description: Draws a rectangle outline or filled rectangle. Uses global foreground color by default, or `WITH color%` to override. Default is outline only, use `FILLED` for filled rectangle. Coordinates use bottom-left origin (0,0).

Example:

```
COLOR &HFFFFFFFF    ' Set global color to white
RECTANGLE FROM (50, 50) TO (150, 150)    ' White outline (uses global color)
RECTANGLE FROM (200, 200) TO (300, 300) WITH &HFF0000FF FILLED    ' Filled red
rectangle
```

RETURN

Type: Command (Control Flow)

Syntax: `RETURN`

Description: Returns from a `GOSUB` subroutine to the statement after the `GOSUB` call.

Example:

```
GOSUB PrintMessage
PRINT "Done"
END

LABEL PrintMessage
    PRINT "Hello!"
RETURN
```

REVERSE

Type: Operator (Array)

Syntax: `REVERSE array[]`

Description: Returns a new array with elements in reverse order.

Example:

```
LET reversed%[] = REVERSE numbers%[]
```

RMDIR

Type: Command (File I/O)
Syntax: `RMDIR "path"`
Description: Removes an empty directory.
Example:

```
RMDIR "temp"
RMDIR "/Users/name/old_data"
```

RND#

Type: Operator (Random)
Syntax: `RND#`
Description: Returns a random real number in the range [0, 1).
Example:

```
LET random# = RND#
LET dice% = INT (RND# * 6) + 1    ' Random integer 1-6
```

ROUND

Type: Operator (Rounding)
Syntax: `ROUND x#`
Description: Rounds `x` to the nearest integer. Ties round up.
Example:

```
PRINT ROUND 3.5      ' Prints: 4
PRINT ROUND 3.4      ' Prints: 3
```

OVAL

Type: Command (Graphics)
Syntax: `OVAL AT (x%, y%) WIDTH width# HEIGHT height#` or `OVAL AT (x%, y%) WIDTH width# HEIGHT height# WITH color%` or `OVAL AT (x%, y%) WIDTH width# HEIGHT height# FILLED` or `OVAL AT (x%, y%) WIDTH width# HEIGHT height# WITH color% FILLED`
Description: Draws an oval (ellipse) outline or filled oval. Uses global foreground color by default, or `WITH color%` to override. Default is outline only, use `FILLED` for filled oval. A circle

is a special case where width equals height. Coordinates use bottom-left origin (0,0).

Example:

```
COLOR &HFFFF00FF    ' Set global color to yellow
OVAL AT (320, 240) WIDTH 100 HEIGHT 100    ' Yellow circle outline (uses global
color)
OVAL AT (100, 100) WIDTH 60 HEIGHT 60 WITH &H00FF00FF FILLED    ' Filled green
circle
OVAL AT (200, 200) WIDTH 80 HEIGHT 40 WITH &HFF00FFFF FILLED    ' Filled ellipse
(2:1 width:height)
```

RTRIM

Type: Operator (String)

Syntax: `RTRIM string$`

Description: Returns a copy of the string with trailing spaces removed.

Example:

```
LET trimmed$ = RTRIM "text  "    ' "text"
```

RIGHT

Type: Operator (String)

Syntax: `string$ RIGHT length%`

Description: Returns the rightmost `length%` characters of the string. If `length%` is greater than the string length, returns the entire string.

Example:

```
LET text$ = "Hello, world!"
LET lastWord$ = text$ RIGHT 6    ' "world!"
LET lastChar$ = text$ RIGHT 1    ' "!"
```

SELECT CASE

Type: Command (Control Flow)

Syntax: `SELECT CASE expression ... END SELECT`

Description: Multi-way branching based on the value of an expression.
Example:

```
SELECT CASE grade%
    CASE 90 TO 100
        PRINT "A"
    CASE 80 TO 89
        PRINT "B"
    CASE ELSE
        PRINT "Other"
END SELECT
```

SEEK

Type: Command (File I/O)
Syntax: `SEEK position% IN #fileHandle%`
Description: Positions the file pointer at a specific byte position (0-based). Position 0 is the beginning of the file. For text files, positions refer to UTF-8 byte positions. Seeking past end of file is allowed (file will extend on write).
Example:

```
OPEN "data.bin" FOR READ AS file%
SEEK 100 IN #file%     ' Jump to byte 100
READ value% FROM file%
SEEK 0 IN #file%       ' Return to beginning
CLOSE file%
```

SET

Type: Command (System Settings)
Syntax: `SET LINE SPACING ON` / `SET LINE SPACING OFF` / `SET TEXT WRAP ON` / `SET TEXT WRAP OFF` / `SET AUDIO ON` / `SET AUDIO OFF` / `SET VOLUME volume#`
Description: Configures system-wide settings. **Line Spacing:** `SET LINE SPACING OFF` uses 80×30 character grid (default). `SET LINE SPACING ON` uses 80×24 character grid with 4 additional pixels of spacing after each line. **Text Wrapping:** `SET TEXT WRAP OFF` truncates long lines at 80 characters (default). `SET TEXT WRAP ON` automatically wraps long lines to the next line. **Audio:** `SET AUDIO ON` enables all audio output (default). `SET AUDIO OFF` disables all audio output (PLAY statements are ignored but produce no error). **Volume:** `SET VOLUME volume#` sets global audio volume as a real number between 0.0 and 1.0 (inclusive),

automatically clamped to [0, 1]. Volume 0.0 is silent, 1.0 is maximum. This is an alternative to the `VOLUME` statement (which uses integer values 0-127).

Example:

```
SET LINE SPACING OFF    ' Use 80×30 character grid (default)
SET LINE SPACING ON     ' Use 80×24 character grid with spacing

SET TEXT WRAP OFF       ' Truncate long lines (default)
SET TEXT WRAP ON        ' Wrap long lines automatically

SET AUDIO ON            ' Enable audio output (default)
SET AUDIO OFF           ' Disable audio output

SET VOLUME 1.0          ' Maximum volume
SET VOLUME 0.5          ' Half volume
SET VOLUME 0.0          ' Silent
```

SHIFT

Type: Statement (Array)

Syntax: `SHIFT array[] INTO variable`

Description: Removes the first element from an array and assigns it to a variable.

Example:

```
SHIFT numbers%[] INTO value%
SHIFT names$[] INTO name$
```

SGN

Type: Function (Math)

Syntax: `SGN x#`

Description: Returns the sign of `x` : -1 (negative), 0 (zero), or 1 (positive).

Example:

```
PRINT SGN 5        ' Prints: 1
PRINT SGN -3       ' Prints: -1
PRINT SGN 0        ' Prints: 0
```

SIN

Type: Function (Trigonometric)
Syntax: `SIN x#`
Description: Returns the sine of `x` (where `x` is in radians).
Example:

```
LET result# = SIN (90 RAD)
PRINT result#    ' Prints: 1.0
```

SINH

Type: Function (Hyperbolic)
Syntax: `SINH x#`
Description: Returns the hyperbolic sine of `x` .
Example:

```
LET result# = SINH 0
PRINT result#    ' Prints: 0.0
```

SQRT

Type: Function (Math)
Syntax: `SQRT x#`
Description: Returns the square root of `x` .
Example:

```
LET result# = SQRT 16
PRINT result#    ' Prints: 4.0
```

STARTSWITH

Type: Operator (String)
Syntax: `string$ STARTSWITH prefix$`
Description: Checks if a string starts with the specified prefix. Returns integer: 0 = false, -1 =

true. Case-sensitive.

Example:

```
LET filename$ = "document.txt"
IF filename$ STARTSWITH "doc" THEN PRINT "Starts with 'doc'"
IF filename$ STARTSWITH ".txt" THEN PRINT "Starts with '.txt'"    ' FALSE%
```

STR

Type: Operator (String)

Syntax: `STR number`

Description: Converts any numeric type (integer, real, or complex) to its decimal string representation. For complex numbers, the format is `"real+imaginaryi"` or `"real-imaginaryi"` .

Example:

```
LET num% = 42
LET numStr$ = STR num%    ' "42"
LET piStr$ = STR 3.14159  ' "3.14159"
LET complexStr$ = STR (3+4i)    ' "3+4i"
LET complexStr$ = STR (3-4i)    ' "3-4i"
```

ENDSWITH

Type: Operator (String)

Syntax: `string$ ENDSWITH suffix$`

Description: Checks if a string ends with the specified suffix. Returns integer: 0 = false, -1 = true. Case-sensitive.

Example:

```
LET filename$ = "document.txt"
IF filename$ ENDSWITH ".txt" THEN PRINT "Ends with '.txt'"
IF filename$ ENDSWITH "doc" THEN PRINT "Ends with 'doc'"    ' FALSE
```

STEP

Type: Keyword (Control Flow)

Syntax: Used within `FOR` loops

Description: Specifies the increment/decrement value for a `FOR` loop.

Example:

```
FOR i% = 0 TO 100 STEP 10
    PRINT i%
NEXT i%
```

SUB

Type: Command (Control Flow)

Syntax: `SUB name parameters ... END SUB`

Description: Defines a subroutine procedure that can accept parameters. Parentheses are not permitted.

Example:

```
SUB DrawBox width%, height%, char$
    FOR row% = 1 TO height%
        FOR col% = 1 TO width%
            PRINT char$;
        NEXT col%
        PRINT
    NEXT row%
END SUB
```

SWAP

Type: Command (Variable Operation)

Syntax: `SWAP variable1 WITH variable2` or `SWAP array1[] WITH array2[]`

Description: Exchanges the values of two variables of the same type. Can swap scalars or arrays. Both operands must be of the same type.

Example:

```
LET x% = 5
LET y% = 10
SWAP x% WITH y%
PRINT "x: ", x%, " y: ", y%    ' Prints: x: 10 y: 5
```

```
LET arr1%[] = [1, 2, 3]
LET arr2%[] = [4, 5, 6]
SWAP arr1%[] WITH arr2%[]
PRINT arr1%[]    ' Prints: 4, 5, 6
PRINT arr2%[]    ' Prints: 1, 2, 3
```

TAN

Type: Function (Trigonometric)
Syntax: `TAN x#`
Description: Returns the tangent of `x` (where `x` is in radians).
Example:

```
LET result# = TAN (45 RAD)
PRINT result#    ' Prints: 1.0
```

TANH

Type: Function (Hyperbolic)
Syntax: `TANH x#`
Description: Returns the hyperbolic tangent of `x` .
Example:

```
LET result# = TANH 0
PRINT result#    ' Prints: 0.0
```

TIME$

Type: Operator (Date/Time)
Syntax: `TIME$`
Description: Returns the current time as a string in `HH:MM:SS` format (24-hour format).
Example:

```
LET now$ = TIME$
PRINT "Current time: ", now$    ' Prints: "Current time: 14:30:45"
PRINT DATE$, " ", TIME$, " - Program started"
```

THEN

Type: Keyword (Control Flow)
Syntax: Used with `IF` and `UNLESS`
Description: Separates the condition from the action in conditional statements.
Example:

```
IF x% > 0 THEN PRINT "Positive"
```

THROW

Type: Command (Control Flow)
Syntax: `THROW errorMessage$`
Description: Throws an error. The error message is a string. Control immediately transfers to the nearest `CATCH` block, or the program terminates if no `CATCH` block exists.
Example:

```
IF divisor% = 0 THEN
    THROW "Division by zero"
END IF

IF NOT EXISTS filename$ THEN
    THROW "File not found: " + filename$
END IF
```

TRY

Type: Command (Control Flow)
Syntax: `TRY ... CATCH errorVariable$ ... FINALLY ... END TRY`
Description: Begins a structured error handling block. The `TRY` block contains code that might raise an error. Errors are represented as strings. `CATCH` and `FINALLY` are optional.
Example:

```
TRY
    OPEN "data.txt" FOR READ AS file%
    READFILE "data.txt" INTO content$
CATCH error$
    PRINT "Error: ", error$
FINALLY
    CLOSE file%
END TRY
```

TO

Type: Keyword (Control Flow)
Syntax: Used within `FOR` loops and `CASE` clauses
Description: Specifies a range in `FOR` loops or `CASE` statements.
Example:

```
FOR i% = 1 TO 10
NEXT i%

CASE 90 TO 100
    PRINT "A"
```

TRIM

Type: Operator (String)
Syntax: `TRIM string$`
Description: Returns a copy of the string with leading and trailing spaces removed.
Example:

```
LET trimmed$ = TRIM "   text   "    ' "text"
```

TRIANGLE

Type: Command (Graphics)
Syntax: `TRIANGLE (x1%, y1%) TO (x2%, y2%) TO (x3%, y3%) WITH color%` or `TRIANGLE (x1%, y1%) TO (x2%, y2%) TO (x3%, y3%) WITH color% FILLED`
Description: Draws a triangle with vertices at the three specified points. With `FILLED` , draws

a filled triangle. Without `FILLED`, draws only the triangle outline. Coordinates use bottom-left origin (0,0).
Example:

```
TRIANGLE (100, 100) TO (200, 200) TO (150, 250) WITH &HFFFFFFFF    ' Outline
TRIANGLE (50, 50) TO (150, 50) TO (100, 150) WITH &HFF0000FF FILLED    ' Filled
```

TRUNC

Type: Operator (Rounding)
Syntax: `TRUNC x#`
Description: Rounds `x` toward zero (truncates the decimal part).
Example:

```
PRINT TRUNC 3.9      ' Prints: 3
PRINT TRUNC -3.9     ' Prints: -3
```

TRUE%

Type: Operator (Boolean Constant)
Syntax: `TRUE%`
Description: Returns the canonical boolean true value as an integer (-1). All bits are set to 1 in two's complement representation, which makes bitwise operations behave correctly. For example, `TRUE% AND TRUE%` yields `TRUE%` (`-1 AND -1 = -1`).
Example:

```
LET flag% = TRUE%
IF condition% THEN
    LET result% = TRUE%
END IF

LET check% = (x% > 0) AND TRUE%    ' Bitwise AND with TRUE%
```

FALSE%

Type: Operator (Boolean Constant)
Syntax: `FALSE%`

Description: Returns the canonical boolean false value as an integer (0).

Example:

```
LET flag% = FALSE%
IF NOT condition% THEN
    LET result% = FALSE%
END IF

LET check% = (x% < 0) OR FALSE%    ' Bitwise OR with FALSE%
```

TURTLE

Type: Command (Graphics)

Syntax: `TURTLE turtleNumber%, commandString$`

Description: Executes Logo-style turtle graphics commands. Turtles draw on the 640×480 canvas using the current foreground color. Commands: `FD n` (forward), `BK n` (back), `RT n` (right turn), `LT n` (left turn), `PU` (pen up), `PD` (pen down), `HOME` (return to center).

Example:

```
TURTLE 0, "FD 100 RT 90 FD 100 RT 90 FD 100 RT 90 FD 100"    ' Square
TURTLE 1, "FD 100 RT 120 FD 100 RT 120 FD 100"              ' Triangle
```

TEMPO

Type: Command (Audio)

Syntax: `TEMPO beatsPerMinute%`

Description: Sets the playback tempo for MML music in beats per minute (BPM). Affects all subsequent `PLAY` statements using MML. Default tempo is typically 120 BPM.

Example:

```
TEMPO 120    ' Set to 120 BPM (moderate tempo)
TEMPO 60     ' Set to 60 BPM (slow)
TEMPO 180    ' Set to 180 BPM (fast)
```

UCASE

Type: Operator (String)
Syntax: `UCASE string$`
Description: Returns a copy of the string converted to uppercase.
Example:

```
LET upper$ = UCASE "hello"    ' "HELLO"
```

UNLESS

Type: Command (Control Flow)
Syntax: `UNLESS condition THEN statement` or `UNLESS condition THEN ... END UNLESS` or `UNLESS condition THEN ... ELSE ... END UNLESS`
Description: Syntactic sugar for `IF NOT` . Executes when condition is false.
Example:

```
UNLESS valid% THEN PRINT "Invalid"

UNLESS password$ = "secret" THEN
    PRINT "Access denied"
END UNLESS

UNLESS balance# >= price# THEN
    PRINT "Insufficient funds"
ELSE
    LET balance# -= price#
    PRINT "Purchase complete"
END UNLESS
```

UNTIL

Type: Command (Control Flow)
Syntax: `UNTIL condition ... UEND`
Description: Syntactic sugar for `WHILE NOT` . Repeats a block until a condition becomes true. Condition tested before each iteration.
Example:

```
LET count% = 0
UNTIL count% >= 10
    PRINT count%
    LET count% += 1
UEND

LET input$ = ""
UNTIL input$ = "quit"
    INPUT "Enter command (or 'quit'): ", input$
    PRINT "You entered: ", input$
UEND
```

UEND

Type: Command (Control Flow)
Syntax: `UEND`
Description: Marks the end of an `UNTIL` loop.
Example:

```
UNTIL count% >= 10
    PRINT count%
    LET count% += 1
UEND
```

UNSHIFT

Type: Statement (Array)
Syntax: `UNSHIFT array[], value`
Description: Adds an element to the beginning of an array.
Example:

```
UNSHIFT numbers%[], 0
UNSHIFT names$[], "Alice"
```

VAL

Type: Operator (String)
Syntax: `VAL string$`
Description: Converts a string to a real number. Supports decimal, hexadecimal (with `&H` prefix), binary (with `&B` prefix), and complex number formats. Hexadecimal and binary strings are parsed as integers (then converted to real). Complex numbers use the format

`"real+imaginaryi"` or `"real−imaginaryi"` . When assigned to integer or complex variables, type coercion automatically converts the result.

Example:

```
LET text$ = "123"
LET number% = VAL text$    ' 123 (coerced to integer)
LET decimal$ = "3.14"
LET value# = VAL decimal$  ' 3.14
LET hexValue% = VAL "&HFF"    ' 255 (hexadecimal, coerced to integer)
LET binValue% = VAL "&B1010"  ' 10 (binary, coerced to integer)
LET complexValue& = VAL "3+4i"    ' 3+4i (complex number, coerced to complex)
LET complexValue& = VAL "3-4i"    ' 3-4i (complex number, coerced to complex)
```

VOICE

Type: Command (Audio)

Syntax:

```
VOICE voiceNumber% PRESET noisePreset%
VOICE voiceNumber% WITH noiseCode%
VOICE voiceNumber% PRESET noisePreset% ADSR attack# decay# sustain# release#
VOICE voiceNumber% WITH noiseCode% ADSR attack# decay# sustain# release#
VOICE voiceNumber% PRESET noisePreset% ADSR PRESET adsrPreset%
VOICE voiceNumber% WITH noiseCode% ADSR PRESET adsrPreset%
```

Description: Configures the GRIT timbre (sound character) and ADSR envelope for a voice. All voices use MML (for notes), GRIT (for timbre), and ADSR (for amplitude shaping). `voiceNumber%` identifies the voice (0-63, for 64 total voices). **Initialization:** All 64 voices are initialized with GRIT NoiseCode preset 0 and ADSR preset 0 by default, and can be used immediately in `PLAY` statements. **NoiseCode:** `PRESET noisePreset%` uses a preset from the 128-entry GRIT preset table (0-127). `WITH noiseCode%` uses a custom NoiseCode value (32-bit unsigned integer). **ADSR:** `ADSR attack# decay# sustain# release#` configures the amplitude envelope with custom values (attack and decay in seconds, sustain 0.0-1.0, release in seconds). `ADSR PRESET adsrPreset%` uses a preset from the 16-entry ADSR preset table (0-15). If ADSR is not specified, the voice retains its current ADSR configuration (or uses ADSR preset 0 if never set). Voices retain their configuration until explicitly changed by another `VOICE` statement.

Example:

```
' Voices can be used immediately with defaults (NoiseCode preset 0, ADSR preset
0)
```

```
PLAY 0, "CDEFGAB C"

VOICE 0 PRESET 5    ' GRIT preset with default ADSR (preset 0)
VOICE 1 PRESET 48 ADSR PRESET 1    ' Drum sound with percussive ADSR preset
LET myNoise% = &H12345678
VOICE 2 WITH myNoise% ADSR PRESET 3    ' Custom NoiseCode with pad ADSR preset
VOICE 3 PRESET 10 ADSR 0.01 0.1 0.0 0.1    ' GRIT preset with custom ADSR
```

VOLUME

Type: Command (Audio)

Syntax: `VOLUME volumeLevel%`

Description: Sets the global volume for all audio output. Volume level is 0-127, where 0 = silent and 127 = maximum volume. Default volume is typically 64 (50%). Affects all voices regardless of their configuration. Individual voice velocity (V in MML) is relative to the global volume.

Example:

```
VOLUME 127    ' Maximum volume
VOLUME 64     ' Half volume (default)
VOLUME 32     ' Quarter volume
VOLUME 0      ' Silent
```

WEND

Type: Command (Control Flow)

Syntax: `WEND`

Description: Marks the end of a `WHILE` loop.

Example:

```
WHILE count% < 10
    PRINT count%
    LET count% += 1
WEND
```

WHILE

Type: Command (Control Flow)

Syntax: `WHILE condition ... WEND`

Description: Repeats a block while a condition is true. Condition tested before each iteration.
Example:

```
LET count% = 0
WHILE count% < 10
    PRINT count%
    LET count% += 1
WEND
```

WRITE

Type: Command (File I/O)
Syntax: `WRITE expression TO fileHandle%` or `WRITE array[] TO fileHandle%`
Description: Writes data to a file. For strings: writes the text followed by a newline. For numbers: writes the binary representation (not text). For arrays, writes all elements sequentially in binary format. Text and binary operations can be mixed in the same file.
Example:

```
OPEN "output.txt" FOR OVERWRITE AS file%
WRITE "Name: " TO file%
WRITE playerName$ TO file%
WRITE score% TO file%    ' Write binary integer
WRITE numbers%[] TO file%    ' Write entire array
CLOSE file%
```

WRITEFILE

Type: Command (File I/O)
Syntax: `WRITEFILE "filename" FROM contentVariable$` or `WRITEFILE contentVariable$ TO "filename"`
Description: Writes an entire string to a file.
Example:

```
LET report$ = "Sales Report" + CHR 10 + "Total: $1000"
WRITEFILE "report.txt" FROM report$
WRITEFILE output$ TO "results.txt"
```

XNOR

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 XNOR expression2`
Description: Bitwise XNOR. Output bit is 1 when both input bits are the same.
Example:

```
LET result% = 12 XNOR 10
```

XOR

Type: Operator (Boolean/Bitwise)
Syntax: `expression1 XOR expression2`
Description: Bitwise XOR. Output bit is 1 when the input bits are different.
Example:

```
LET result% = 12 XOR 10    ' Binary: 1100 XOR 1010 = 0110 (6)
```

Appendix: Complete Operator Reference

This appendix provides a comprehensive listing of all operators in EduBASIC, organized by category.

Arithmetic Operators

Operator	Description	Example	Returns
`+`	Addition	`5 + 3`	Integer or Real
`-`	Subtraction	`10 - 4`	Integer or Real
`*`	Multiplication	`6 * 7`	Integer or Real
`/`	Division	`15 / 4`	Real
`MOD`	Modulo (remainder)	`17 MOD 5`	Integer
`^`	Exponentiation	`2 ^ 8`	Real
`**`	Exponentiation (alt)	`2 ** 8`	Real

Operator	Description	Example	Returns
`!`	Factorial	`5!`	Integer
`+`	Unary plus	`+x#`	Same as operand
`-`	Unary minus (negation)	`-x#`	Same as operand

Note: Exponentiation (`^` , `**`) is right-associative. All other operators are left-associative.

Assignment Operators

Operator	Description	Example	Equivalent To
`=`	Assignment	`LET x% = 5`	N/A
`+=`	Addition assignment	`LET x% += 5`	`LET x% = x% + 5`
`-=`	Subtraction assignment	`LET x% -= 3`	`LET x% = x% - 3`
`*=`	Multiplication assignment	`LET x% *= 2`	`LET x% = x% * 2`
`/=`	Division assignment	`LET x% /= 4`	`LET x% = x% / 4`
`^=`	Exponentiation assignment	`LET x% ^= 2`	`LET x% = x% ^ 2`

Note: The `LET` keyword is required for all assignments.

Comparison Operators

Operator	Description	Example	Returns
`=`	Equal to	`x% = y%`	Integer (0 or -1)
`<>`	Not equal to	`x% <> y%`	Integer (0 or -1)
`<`	Less than	`x% < y%`	Integer (0 or -1)
`>`	Greater than	`x% > y%`	Integer (0 or -1)
`<=`	Less than or equal to	`x% <= y%`	Integer (0 or -1)
`>=`	Greater than or equal to	`x% >= y%`	Integer (0 or -1)

Note: Comparison operators return `-1` for true, `0` for false. They work with all data types including strings (lexicographic comparison) and arrays (element-wise comparison).

Logical (Bitwise) Operators

Operator	Description	Example	Returns
`AND`	Bitwise AND	`x% AND y%`	Integer
`OR`	Bitwise OR	`x% OR y%`	Integer
`NOT`	Bitwise NOT (complement)	`NOT x%`	Integer
`XOR`	Bitwise XOR (exclusive OR)	`x% XOR y%`	Integer
`NAND`	Bitwise NAND (NOT AND)	`x% NAND y%`	Integer
`NOR`	Bitwise NOR (NOT OR)	`x% NOR y%`	Integer
`XNOR`	Bitwise XNOR (exclusive NOR)	`x% XNOR y%`	Integer
`IMP`	Bitwise implication	`x% IMP y%`	Integer

Note: All logical operators are bitwise. Boolean constants: `FALSE% = 0`, `TRUE% = -1`.

String Operators

Operator	Description	Example	Returns
`+`	String concatenation	`"Hello" + " " + "World"`	String
`LEFT`	Extract left portion	`text$ LEFT 5`	String
`RIGHT`	Extract right portion	`text$ RIGHT 6`	String
`MID`	Extract substring	`text$ MID 8 TO 12`	String
`INSTR`	Find substring position	`"Hello world" INSTR "world"`	Integer
`INSTR FROM`	Find substring from position	`"Hello world" INSTR "o" FROM 5`	Integer
`REPLACE WITH`	Replace substring	`"Hello" REPLACE "ll" WITH "y"`	String
`JOIN`	Join array elements	`names$[] JOIN ", "`	String
`STARTSWITH`	Check if starts with prefix	`filename$ STARTSWITH "doc"`	Integer (0 or -1)

Operator	Description	Example	Returns
`ENDSWITH`	Check if ends with suffix	`filename$ ENDSWITH ".txt"`	Integer (0 or -1)
`| |`	String length	`| text$ |`	Integer

Array Operators

Operator	Description	Example	Returns
`+`	Array concatenation	`arr1%[] + arr2%[]`	Array
`FIND`	Find element in array	`numbers%[] FIND 5`	Element value or 0
`INDEXOF`	Find index of element	`numbers%[] INDEXOF 5`	Integer (index or 0)
`INCLUDES`	Check if array includes element	`numbers%[] INCLUDES 5`	Integer (0 or -1)
`REVERSE`	Reverse array elements	`REVERSE numbers%[]`	Array
`JOIN`	Join array elements into string	`names$[] JOIN ", "`	String
`| |`	Array length	`| numbers%[] |`	Integer

Mathematical Operators

Trigonometric Operators

Operator	Description	Example	Operand Types	Returns
`SIN`	Sine (radians)	`SIN angle#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`COS`	Cosine (radians)	`COS angle#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`TAN`	Tangent (radians)	`TAN angle#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)

Operator	Description	Example	Operand Types	Returns
`ASIN`	Arcsine (returns radians)	`ASIN ratio#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`ACOS`	Arccosine (returns radians)	`ACOS ratio#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`ATAN`	Arctangent (returns radians)	`ATAN slope#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)

Hyperbolic Operators

Operator	Description	Example	Operand Types	Returns
`SINH`	Hyperbolic sine	`SINH value#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`COSH`	Hyperbolic cosine	`COSH value#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`TANH`	Hyperbolic tangent	`TANH value#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`ASINH`	Inverse hyperbolic sine	`ASINH value#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`ACOSH`	Inverse hyperbolic cosine	`ACOSH value#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`ATANH`	Inverse hyperbolic tangent	`ATANH value#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)

Exponential and Logarithmic Operators

Operator	Description	Example	Operand Types	Returns
`EXP`	e raised to power	`EXP power#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)

Operator	Description	Example	Operand Types	Returns
`LOG`	Natural logarithm (base e)	`LOG value#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`LOG10`	Common logarithm (base 10)	`LOG10 value#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`LOG2`	Binary logarithm (base 2)	`LOG2 value#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)

Root Operators

Operator	Description	Example	Operand Types	Returns
`SQRT`	Square root	`SQRT number#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)
`CBRT`	Cube root	`CBRT number#`	Integer, Real, Complex	Real (Integer/Real), Complex (Complex)

Rounding and Truncation Operators

Operator	Description	Example	Operand Types	Returns
`ROUND`	Round to nearest integer (ties up)	`ROUND 3.5`	Integer, Real	Real
`FLOOR`	Round toward negative infinity	`FLOOR 3.7`	Integer, Real	Real
`CEIL`	Round toward positive infinity	`CEIL 3.2`	Integer, Real	Real
`TRUNC`	Round toward zero	`TRUNC 3.9`	Integer, Real	Real
`EXPAND`	Round away from zero	`EXPAND 3.1`	Integer, Real	Real
`INT`	Integer conversion (floor for pos, ceil for neg)	`INT 3.7`	Integer, Real	Integer

Note: Rounding and truncation operators are not applicable to complex numbers.

Other Mathematical Operators

Operator	Description	Example	Operand Types	Returns
`SGN`	Sign of number (-1, 0, or 1)	`SGN value#`	Integer, Real	Integer
`| |`	Absolute value / norm	`| -5 |` or `| 3+4i |`	Integer, Real, Complex	Real

Note: The `SGN` operator is not applicable to complex numbers. Use `CABS` for complex number magnitude.

Complex Number Operators

Operator	Description	Example	Returns
`REAL`	Real part of complex number	`REAL z&`	Real
`IMAG`	Imaginary part of complex	`IMAG z&`	Real
`CONJ`	Complex conjugate	`CONJ z&`	Complex
`CABS`	Absolute value (magnitude)	`CABS z&`	Real
`CARG`	Argument (phase angle) in radians	`CARG z&`	Real
`CSQRT`	Complex square root	`CSQRT z&`	Complex

Type Conversion Operators

Operator	Description	Example	Returns
`INT`	Convert to integer	`INT 3.7`	Integer
`STR`	Convert to string	`STR 42`	String
`VAL`	Convert string to number	`VAL "123"`	Integer, Real, or Complex
`HEX`	Convert to hexadecimal string	`HEX 255`	String
`BIN`	Convert to binary string	`BIN 10`	String
`CHR`	Convert ASCII code to character	`CHR 65`	String

Operator	Description	Example	Returns
`ASC`	Convert character to ASCII code	`ASC "A"`	Integer

String Manipulation Operators

Operator	Description	Example	Returns
`UCASE`	Convert to uppercase	`UCASE "hello"`	String
`LCASE`	Convert to lowercase	`LCASE "WORLD"`	String
`LTRIM`	Remove leading spaces	`LTRIM "  text"`	String
`RTRIM`	Remove trailing spaces	`RTRIM "text  "`	String
`TRIM`	Remove leading/trailing spaces	`TRIM "  text  "`	String
`REVERSE`	Reverse string	`REVERSE "abc"`	String

Unit Conversion Operators

Operator	Description	Example	Returns
`DEG`	Convert radians to degrees	`(3.14159 / 2) DEG`	Real
`RAD`	Convert degrees to radians	`90 RAD`	Real

File I/O Operators

Operator	Description	Example	Returns
`EOF`	Check if at end of file	`EOF fileHandle%`	Integer (0 or -1)
`LOC`	Get current byte position	`LOC fileHandle%`	Integer
`EXISTS`	Check if file/directory exists	`EXISTS "data.txt"`	Integer (0 or -1)

Constants

These operators take no arguments and return a value. They use type sigils to indicate their return type.

Operator	Description	Example	Returns
`RND#`	Random real number in [0, 1)	`RND#`	Real
`PI#`	Mathematical constant π	`PI#`	Real
`E#`	Mathematical constant e	`E#`	Real
`TRUE%`	Boolean constant (value: -1)	`TRUE%`	Integer
`FALSE%`	Boolean constant (value: 0)	`FALSE%`	Integer
`INKEY$`	Currently pressed key (non-blocking)	`INKEY$`	String
`DATE$`	Current date (YYYY-MM-DD format)	`DATE$`	String
`TIME$`	Current time (HH:MM:SS format)	`TIME$`	String
`NOW%`	Current Unix timestamp	`NOW%`	Integer

Audio Operators

Operator	Description	Example	Returns
`NOTES`	Number of notes remaining in voice	`NOTES 0`	Integer

Special Operators

Operator	Description	Example	Returns
`( )`	Grouping (override precedence)	`(2 + 3) * 4`	Varies
`[ ]`	Array subscript / structure member	`arr%[5]` or `obj[member$]`	Varies
`[1 TO 5]`	Array slicing	`arr%[1 TO 5]`	Array
`[...]`	Array slice to end	`arr%[5 TO ...]`	Array
`[ ]`	Empty array literal	`[ ]`	Array
`[1, 2, 3]`	Array literal	`[1, 2, 3]`	Array
`{ }`	Empty structure literal	`{ }`	Structure
`{ a$: "x" }`	Structure literal	`{ name$: "Bob", age%: 30 }`	Structure

Operator Precedence Summary

1. **Constants**: `RND#` , `INKEY$` , `PI#` , `E#` , `DATE$` , `TIME$` , `NOW%` , `TRUE%` , `FALSE%` (highest precedence)
2. **Parentheses** `( )`
3. **Mathematical operators**: `SIN` , `COS` , `TAN` , `ASIN` , `ACOS` , `ATAN` , `SINH` , `COSH` , `TANH` , `ASINH` , `ACOSH` , `ATANH` , `EXP` , `LOG` , `LOG10` , `LOG2` , `SQRT` , `CBRT` , `FLOOR` , `CEIL` , `ROUND` , `TRUNC` , `EXPAND` , `SGN`
4. **Complex operators**: `REAL` , `IMAG` , `CONJ` , `CABS` , `CARG` , `CSQRT`
5. **Type conversion operators**: `INT` , `STR` , `VAL` , `HEX` , `BIN`
6. **String manipulation operators**: `ASC` , `CHR` , `UCASE` , `LCASE` , `LTRIM` , `RTRIM` , `TRIM` , `REVERSE`
7. **File I/O operators**: `EOF` , `LOC` , `NOTES`
8. **Postfix operators**: `!` (factorial), `DEG` , `RAD`
9. **Absolute value / norm / length**: `| |`
10. **Unary**: `+` and `-`
11. **Exponentiation**: `^` or `**` (right-associative)
12. **Multiplicative**: `*` , `/` , `MOD`
13. **Additive**: `+` , `-`
14. **Array search**: `FIND` , `INDEXOF` , `INCLUDES`
15. **String/Array operators**: `INSTR` , `JOIN` , `REPLACE` , `LEFT` , `RIGHT` , `MID`
16. **Comparison**: `=` , `<>` , `<` , `>` , `<=` , `>=`
17. **Logical NOT**: `NOT`
18. **Logical AND**: `AND` , `NAND`
19. **Logical OR**: `OR` , `NOR`
20. **Logical XOR**: `XOR` , `XNOR`
21. **Logical IMP**: `IMP` (lowest precedence)

Notes:

- When operators have the same precedence, evaluation proceeds left to right, except for exponentiation which is right-associative
- Constants take zero arguments and can appear anywhere in an expression
- All constants use type sigils in their identifiers to indicate return types

Appendix: CSS Color Names

EduBASIC supports all standard CSS color names as preset colors. Color names are case-insensitive and can be used anywhere a 32-bit RGBA color value is accepted, including the `COLOR` statement and all graphics statements with `WITH color%` syntax.

Usage:

- `COLOR "red"` - Use color name directly (string expressions are automatically recognized as color names)
- `PSET (100, 100) WITH "blue"` - Use color name in graphics statements
- Color names resolve to their standard CSS color values in `&HRRGGBBAA` format with full opacity (alpha = 255)

Aliases:

- `gray` and `grey` are both supported (same color)
- `aqua` and `cyan` are both supported (same color)
- `fuchsia` and `magenta` are both supported (same color)
- `darkgray` / `darkgrey` , `dimgray` / `dimgrey` , `lightgray` / `lightgrey` , `lightslategray` / `lightslategrey` , `slategray` / `slategrey` , and `darkslategray` / `darkslategrey` are all supported

The following table lists all 147 standard CSS color names in alphabetical order:

Color Name	Value (`&HRRGGBBAA`)	Swatch
aliceblue	&HF0F8FFFF	□
antiquewhite	&HFAEBD7FF	□
aqua	&H00FFFFFF	□
aquamarine	&H7FFFD4FF	□
azure	&HF0FFFFFF	□
beige	&HF5F5DCFF	□
bisque	&HFFE4C4FF	□
black	&H000000FF	□
blanchedalmond	&HFFEBCDFF	□
blue	&H0000FFFF	□
blueviolet	&H8A2BE2FF	□

Color Name	Value (`&HRRGGBBAA`)	Swatch
brown	&HA52A2AFF	□
burlywood	&HDEB887FF	□
cadetblue	&H5F9EA0FF	□
chartreuse	&H7FFF00FF	□
chocolate	&HD2691EFF	□
coral	&HFF7F50FF	□
cornflowerblue	&H6495EDFF	□
cornsilk	&HFFF8DCFF	□
crimson	&HDC143CFF	□
cyan	&H00FFFFFF	□
darkblue	&H00008BFF	□
darkcyan	&H008B8BFF	□
darkgoldenrod	&HB8860BFF	□
darkgray	&HA9A9A9FF	□
darkgreen	&H006400FF	□
darkkhaki	&HBDB76BFF	□
darkmagenta	&H8B008BFF	□
darkolivegreen	&H556B2FFF	□
darkorange	&HFF8C00FF	□
darkorchid	&H9932CCFF	□
darkred	&H8B0000FF	□
darksalmon	&HE9967AFF	□
darkseagreen	&H8FBC8FFF	□
darkslateblue	&H483D8BFF	□
darkslategray	&H2F4F4FFF	□

Color Name	Value (`&HRRGGBBAA`)	Swatch
darkturquoise	&H00CED1FF	□
darkviolet	&H9400D3FF	□
deeppink	&HFF1493FF	□
deepskyblue	&H00BFFFFF	□
dimgray	&H696969FF	□
dodgerblue	&H1E90FFFF	□
firebrick	&HB22222FF	□
floralwhite	&HFFFAF0FF	□
forestgreen	&H228B22FF	□
fuchsia	&HFF00FFFF	□
gainsboro	&HDCDCDCFF	□
ghostwhite	&HF8F8FFFF	□
gold	&HFFD700FF	□
goldenrod	&HDAA520FF	□
gray	&H808080FF	□
green	&H008000FF	□
greenyellow	&HADFF2FFF	□
honeydew	&HF0FFF0FF	□
hotpink	&HFF69B4FF	□
indianred	&HCD5C5CFF	□
indigo	&H4B0082FF	□
ivory	&HFFFFF0FF	□
khaki	&HF0E68CFF	□
lavender	&HE6E6FAFF	□
lavenderblush	&HFFF0F5FF	□

Color Name	Value (`&HRRGGBBAA`)	Swatch
lawngreen	&H7CFC00FF	☐
lemonchiffon	&HFFFACDFF	☐
lightblue	&HADD8E6FF	☐
lightcoral	&HF08080FF	☐
lightcyan	&HE0FFFFFF	☐
lightgoldenrodyellow	&HFAFAD2FF	☐
lightgray	&HD3D3D3FF	☐
lightgreen	&H90EE90FF	☐
lightpink	&HFFB6C1FF	☐
lightsalmon	&HFFA07AFF	☐
lightseagreen	&H20B2AAFF	☐
lightskyblue	&H87CEFAFF	☐
lightslategray	&H778899FF	☐
lightsteelblue	&HB0C4DEFF	☐
lightyellow	&HFFFFE0FF	☐
lime	&H00FF00FF	☐
limegreen	&H32CD32FF	☐
linen	&HFAF0E6FF	☐
magenta	&HFF00FFFF	☐
maroon	&H800000FF	☐
mediumaquamarine	&H66CDAAFF	☐
mediumblue	&H0000CDFF	☐
mediumorchid	&HBA55D3FF	☐
mediumpurple	&H9370DBFF	☐
mediumseagreen	&H3CB371FF	☐

Color Name	Value (`&HRRGGBBAA`)	Swatch
mediumslateblue	&H7B68EEFF	□
mediumspringgreen	&H00FA9AFF	□
mediumturquoise	&H48D1CCFF	□
mediumvioletred	&HC71585FF	□
midnightblue	&H191970FF	□
mintcream	&HF5FFFAFF	□
mistyrose	&HFFE4E1FF	□
moccasin	&HFFE4B5FF	□
navajowhite	&HFFDEADFF	□
navy	&H000080FF	□
oldlace	&HFDF5E6FF	□
olive	&H808000FF	□
olivedrab	&H6B8E23FF	□
orange	&HFFA500FF	□
orangered	&HFF4500FF	□
orchid	&HDA70D6FF	□
palegoldenrod	&HEEE8AAFF	□
palegreen	&H98FB98FF	□
paleturquoise	&HAFEEEEFF	□
palevioletred	&HDB7093FF	□
papayawhip	&HFFEFD5FF	□
peachpuff	&HFFDAB9FF	□
peru	&HCD853FFF	□
pink	&HFFC0CBFF	□
plum	&HDDA0DDFF	□

Color Name	Value (`&HRRGGBBAA`)	Swatch
powderblue	&HB0E0E6FF	□
purple	&H800080FF	□
rebeccapurple	&H663399FF	□
red	&HFF0000FF	□
rosybrown	&HBC8F8FFF	□
royalblue	&H4169E1FF	□
saddlebrown	&H8B4513FF	□
salmon	&HFA8072FF	□
sandybrown	&HF4A460FF	□
seagreen	&H2E8B57FF	□
seashell	&HFFF5EEFF	□
sienna	&HA0522DFF	□
silver	&HC0C0C0FF	□
skyblue	&H87CEEBFF	□
slateblue	&H6A5ACDFF	□
slategray	&H708090FF	□
snow	&HFFFAFAFF	□
springgreen	&H00FF7FFF	□
steelblue	&H4682B4FF	□
tan	&HD2B48CFF	□
teal	&H008080FF	□
thistle	&HD8BFD8FF	□
tomato	&HFF6347FF	□
turquoise	&H40E0D0FF	□
violet	&HEE82EEFF	□

Color Name	Value (`&HRRGGBBAA`)	Swatch
wheat	&HF5DEB3FF	□
white	&HFFFFFFFF	□
whitesmoke	&HF5F5F5FF	□
yellow	&HFFFF00FF	□
yellowgreen	&H9ACD32FF	□

Note: The swatches in the table above are visual representations. In actual usage, color names are case-insensitive, so `"Red"` , `"red"` , and `"RED"` all refer to the same color.