# Contents

[Contents 1](#_Toc403749228)

[1 Preliminaries 2](#_Toc403749229)

[2 Tokens 2](#_Toc403749230)

[3 Data Types and Variables 3](#_Toc403749231)

[3.1 Variable Declaration 3](#_Toc403749232)

[3.2 Primitive Types 4](#_Toc403749233)

[3.3 Pointers 5](#_Toc403749234)

[3.4 Composite Types 5](#_Toc403749235)

[3.4.1 Arrays 6](#_Toc403749236)

[3.4.2 Hash (Experimental) 7](#_Toc403749237)

[3.4.3 Enumeration 7](#_Toc403749238)

[3.4.4 Record 7](#_Toc403749239)

[3.5 Type Casting and Aliasing 8](#_Toc403749240)

[3.6 Scope 8](#_Toc403749241)

[4 Expressions, Operators and Statements 9](#_Toc403749242)

[4.1 Expressions 9](#_Toc403749243)

[4.2 Operators 9](#_Toc403749244)

[4.3 Statement 11](#_Toc403749245)

[5 Control Flow 11](#_Toc403749246)

[5.1 Selection 11](#_Toc403749247)

[5.1.1 The Two-Way Selection Statement 11](#_Toc403749248)

[5.1.2 Multiple-Selection Statement 11](#_Toc403749249)

[5.2 Iteration 12](#_Toc403749250)

[5.2.1 The while Loop 12](#_Toc403749251)

[5.2.2 The do-while Loop 12](#_Toc403749252)

[5.2.3 The for Loop 12](#_Toc403749253)

[5.3 Exception Handling 12](#_Toc403749254)

[6 Procedures 12](#_Toc403749255)

# Preliminaries

This manual contains the formal syntax and semantics description of the Epsilon language and is intended for the compiler writers of the language.

The syntax description is presented inside blue boxes and is typed in Consolas font. It uses the Extended Backus-Naur Form or EBNF format. The following are the guidelines on how EBNF is used in this manual:

* All nonterminal symbols are enclosed with <> symbols. Example: <number>, <letter>
* All terminal symbols are in boldface. Example: **if**, **then**, **for**
* The symbol **=>** is used for production rules and is read as “derives” or “is defined as.” The left side of the symbol is the nonterminal being defined while the right side is the definition.
* The OR symbol | is used for separating alternatives. Example: <id> => **A** | **B** means the nonterminal symbol <id> may produce **A** or **B**.
* Parenthesis is used to group symbols.
* Square bracket is used to denote optional part of the definition.
* The symbol \* is called the star operator. It denotes that its operand can be repeated zero or more times.
* The symbol ⁺ is called the plus operator. It denotes that its operand can be repeated one or more times.

Example code snippets in the manual are presented in Courier New font.

# Tokens

The following are the valid words in the Epsilon language presented in flex compatible regular expression.

|  |  |  |
| --- | --- | --- |
| Regular Expression | Token Name | Description |
| “\n” | NEWLINE | Statement terminator |
| [Tt][Rr][Uu][Ee]|[Ff][Aa][Ll][Ss][Ee] | BOOLEAN | Boolean |
| “\’”(.|”\\n”|”\\t”)”\’” | CHARACTER | Character value |
| “\””.\*”\”” | STRING | String value |
| [-+]?[0-9]+(.[0-9]+)? | NUMBER | Number value |
| [Nn][Uu][Ll][Ll] | NULL | Null value |
| [A-Za-z][A-Za-z0-9\_]{0,30} | ID | ID |
| [Bb][Oo][Oo][Ll][Ee][Aa][Nn] | BOOLEAN | Boolean data type |
| [Cc][Hh][Aa][Rr] | CHAR | Character data type |
| [Ii][Nn][Tt] | INT | Integer data type |
| [Ff][Ll][Oo][Aa][Tt] | FLOAT | Float data type |
| [Dd][Oo][Uu][Bb][Ll][Ee] | DOUBLE | Double data type |
| [Ll][Oo][Nn][Gg] | LONG | Long type qualifier |
| [Ss][Hh][Oo][Rr][Tt] | SHORT | Short type qualifier |
| [Ss][Ii][Gg][Nn][Ee][Dd] | SIGNED | Signed type qualifier |
| [Uu][Nn][Ss][Ii][Gg][Nn][Ee][Dd] | UNSIGNED | Unsigned type qualifier |
| “[“ | LEFTBRACKET | Left square bracket |
| “]” | RIGHTBRACKET | Right square bracket |
| “{“ | LEFTBRACE | Left curly brace |
| “}” | RIGHTBRACE | Right curly brace |
| “(” | LEFTPAREN | Left parenthesis |
| “)” | RIGHTPAREN | Right parenthesis |
| “,” | COMMA | Comma operator |
| “=” | ASSIGNMENT | Assignment operator |
| “+” | ADD | Addition operator |
| “-” | SUB | Subtraction operator |
| “\*” | MUL | Multiplication operator |
| “/” | DIV | Division operator |
| “%” | MOD | Modulo operator |
|  |  |  |
|  |  |  |

# Data Types and Variables

**Data types** or **types** is a classification that determines an object’s size in memory and the types of operations that can be used with it. A **variable** is a storage location associated with a symbolic name called ID which contains a value.

## Variable Declaration

Before a variable can be used, it must be declared first. Declaration is the process of announcing the existence of an object, therefore the compiler will start creating by binding an ID to it and allocating a storage space for it. The variable’s lifetime depends on the level of scope it has acquired. The grammar rules for declaring variables, except for those with pointer data types, follows the following pattern:

var\_declaration: type ID var\_list;

var\_list: COMMA ID var\_list

| NEWLINE

;

type: BOOLEAN | CHAR | INT | FLOAT | DOUBLE ;

The optional global keyword declares a variable that is in global scope (See [Section 2.3](#_Scope)). This kind of variables, called **global variables**, must be declared outside any function. The nonterminal symbol <type> is the data type of the variable. The comma (,) operator separates individual variables that are declared in one statement. Example:

int x // An integer x

float a, b, c // Three float variables declared in one statement

It is also allowed to initialize the value of a variable in its declaration but the syntax to do this varies in according to data type classification. Epsilon has 5 different classifications of data types: primitive, composite, pointers and abstract data types. Each of these classifications will be discussed in the subsequent sections.

## Primitive Types

Epsilon has the following primitive data types:

|  |  |
| --- | --- |
| Primitive Type | Description |
| boolean | A single-byte representing true or false. |
| char | A single byte data type capable of holding one character |
| int | An integer with the natural size of integer in the local machine |
| float | Single-prescision floating point |
| double | Double-precision floating point |

To initialize the value of the variables in its declaration with these types, use the following syntax:

var\_declaration: type ID var\_list;

var\_list: COMMA ID var\_list

| init

| NEWLINE

;

type: BOOLEAN | CHAR | INT | FLOAT | DOUBLE ;

init: ASSIGNMENT expr NEWLINE;

Example code:

int x = 0

int h, i, j = 0 // variable h, i, j are all equal to zer0

In the grammar above, the nonterminal value may be a literal value or a previously declared variable. This value must have the same data type with the variable(s) being declared or else an implicit casting will be performed by the compiler (See [Section 3.6](#_Type_Casting_and)).

There are some **type qualifiers** that can modify some primitive types. The type qualifiers short and long apply to integers. Integer variables declared in with these qualifiers may omit the keyword int in the declaration statement. The goal of this qualifiers is to resize the natural size of int which is 32 bits. The size of short is at least 16 bits but no greater than int while the size of long is at least 32 but not less than int. In summary, short ≤ int ≤ long.

The qualifier signed and unsigned apply to char and int. unsigned numbers are always positive or zero. By default, int is signed while char is unsigned. char can have negative values but only the positive values are printable.

In the family of floating data types, the long double is added which specifies the extended precision floating-point.

In summary, the syntax for declaring variables with primitive types is:

var-declaration: type ID var-list

| type-qual-list type ID var-list

;

type-qual-list: (LONG | SHORT)

| (LONG | SHORT) (SIGNED | UNSIGNED)

| (SIGNED | UNSIGNED)

| (SIGNED | UNSIGNED) (LONG | SHORT)

;

var-list: COMMA ID var-list

| init

| NEWLINE

;

type: BOOLEAN | CHAR | INT | FLOAT | DOUBLE ;

init: ASSIGNMENT expr NEWLINE;

## Pointers

A pointer is a variable whose value is a memory address. A pointer is always associated with a valid data type to determine the address range it can refer. The operator # is used to denote that a variable is a pointer. To declare a pointer:

int #ptr

The operator # only applies to the variable it is attached to. This means in the code

int num, #numptr, num2

the only pointer variable is the numptr.

A pointer variable must always be initialized before it can be used. To initialize a pointer variable, assign it with a memory location either by memory allocation or using the address operator @, which returns the address of its operand.

int num = 5, #numPtr, num2

numPtr = @num

To access variable that a pointer points, use the dereferencing operator #. An error of **UninitializedPointerException** will result if the dereferencing operator is used with uninitialized pointer.

num2 = #numPtr // num2 = 5

#numPtr = 10 // num = 10

## Composite Types

**Composite types** are data structures derived from more than one data type. Epsilon has the following composite type:

* Array
* Hash
* Enumeration
* Record

### Arrays

An **array** is a group of memory location all having the same name and data type. The syntax for declaring multiple arrays in one statement is:

array-declaration: type ID array-size var-list;

array-size: LEFTBRACKET NUMBER RIGHTBRACKET array-size

| LEFTBRACKET NUMBER RIGHTBRACKET

;

var-list: COMMA ID array-size var-list

| NEWLINE

;

type: BOOLEAN | CHAR | INT | FLOAT | DOUBLE ;

//NOT FINISH

Example codes:

int numbers[10]

int table[5][4]

float grades[10], students[10]

The individual location in an array is called **element**. To access an element an array, use the following syntax:

array-element: ID LEFTBRACKET NUMBER RIGHTBRACKET

| ID

;

The number between the square brackets is the **index** number of the element being accessed. If an array is declared to have 12 elements, the available indexes are 0 to 11. If the array is referenced using its ID only, the value of the first element is the one that will be returned. So,

int numbers[10]

numbers = 9 //The same as numbers[0] = 9

Index of array must be an integer number. Expressions that accesses an array element using index which is a negative integer or outside the range of possible elements will return an **OutofBoundArrayException**.

Programmers can also initialize an array in its declaration. The syntax for this is:

array-declaration: type ID array-id init

| GLOBAL type ID array-id init

;

array-size: LEFTBRACKET NUMBER RIGHTBRACKET array-size

| LEFTBRACKET RIGHTBRACKET array-size

| LEFTBRACKET NUMBER RIGHTBRACKET

| LEFTBRACKET RIGHTBRACKET

;

init: ASSIGNMENT LEFTBRACE init-list NEWLINE

init-list: LEFTBRACE init-list

|

//INITIALIZATION LIST FOR MULTIDIMENSIONAL

value-list: value value-list

| COMMA value

| RIGHTBRACE

;

value: BOOLEAN | CHARACTER | STRING | NUMBER | NULL ;

type: BOOLEAN | CHAR | INT | FLOAT | DOUBLE ;

//NOT FINISH

### Hash (Experimental)

Hash is like an array. It is also a collection of memory location with the same name, but unlike an array, it is dynamic and the location of its elements are not necessarily contagiously stored in the memory. It is dynamic because the size of its elements can change in runtime. Another big difference it has from an array is that its elements are not accessed using index numbers, but rather, they are accessed using **keys** which are strings of characters that identifies each element. A **pair** is the combination of a key and the value that it identifies. Every key in a hash must be unique.

### Enumeration

**Enumeration** is a set of integer constants represented by IDs. The values in the enumeration starts at 0, unless specified otherwise, and are incremented by 1.

enum cardsuit

Clubs

Diamonds

Hearts

Spades

end enum

### Record

A **record** is a collection of variables under one ID. These variables are called **fields** of the record and are accessed using the dot operator. To define a record type, use the syntax,

record date

int day

int month

int year

end record

record-define: RECORD ID NEWLINE fields-list

fields-list: var-declaration fields-list

| array declaration fields-list

| record-declaration fields-list

| END RECORD

To declare a variable of type record date:

record date birthday

To initialize a record variable in its declaration, use this syntax:

record date birthday = {25, 2, 1995}

record-var-declaration: RECORD ID ID = LEFTBRACE expr expr-list

expr-list: expr

[\\NOT](file:///\\NOT) FINISH

To access fields of a record variable, use the dot (.) operator

birthday.year

## Type Casting and Aliasing

Epsilon supports the mechanism of giving an alias to any type using the keyword alias. The grammar rule for type aliasing is:

type-aliasing → **alias** <type> <alias-name> **CR+LF**

<alias-name> → <letter>⁺

## Scope

Epsilon defines 3 levels of scope:

1. **Block scope** – objects declared in this level have scopes from the point of their creation until the end of the block they are created in. Example of this are variables declared inside a function or in the parameter list, inside control flow statements like if..then..end if, and inside the block…end block statements.
2. **File scope** – objects declared in this level have scopes from the point of their creation until the end of file they are created in.
3. **Global scope** – objects declared in this level live in the entire execution of the program. Objects with this scope level must be declared with the global keyword. (Removed)

At any part of the program, an object can either be local or nonlocal. An active object is said to be **local** in a defined region if it is declared there while. An active object is said to be **nonlocal** in a defined region if it is not declared there.

# Expressions, Operators and Statements

## Expressions

An expression is the combination of symbols (e.g. literals, operators, function calls), that when executed, yields a value. The following are the types of Epsilon expressions:

* Arithmetic
* Logical
* Relational
* Function call
* Constant expression

Universally, the grammar for expression is:

expr: const-expr | function-call | arith-expr | log-expr

| rel-expr ;

The grammar for constant expression:

const-expr: BOOLEAN | CHARACTER | STRING | NUMBER | NULL ;

The grammar for function call is:

function-call: ID LEFTPAREN expr-list;

expr-list: COMMA expr expr-list

| RIGHTPAREN

;

The grammar for arithmetic expression is:

arith-expr: term arith-op expr

| LEFTPAREN expr RIGHTPAREN

| term

;

term: const-expr

| function-call

| ID

| LEFTPAREN term RIGHTPAREN

| pre-binary-operator term

| term post-binary-operator

arith-op: ADD | SUB | MUL | DIV | MOD ;

There ar

## Operators

Epsilon operator follows these precedence rules:

|  |  |  |
| --- | --- | --- |
| Operators | Description | Associativity |
| () | Parentheses (function call operator) | Left to right |
| [] | Array subscript |  |
| . | Member selection via object |  |
| -> | Member selection via pointer |  |
| ++ | Unary post-increment |  |
| -- | Unary post-decrement |  |
| ++ | Unary pre-increment | Right to left |
| -- | Unary pre-decrement |  |
| + | Unary plus |  |
| - | Unary minus |  |
| NOT | Unary logical negation |  |
| ~ | Unary bitwise complement |  |
| ( *type* ) | Unary cast |  |
| # | Dereference |  |
| @ | Address |  |
| sizeof | Determine size in bytes |  |
| \* | Multiplication | Left to right |
| / | Division |  |
| % | Modulus |  |
| + | Addition | Left to right |
| - | Subtraction |  |
| << | Bitwise left shift | Left to right |
| >> | Bitwise right shift |  |
| < | Relational less than | Left to right |
| <= | Relational less than or equal to |  |
| > | Relational greater than |  |
| >= | Relational greater than or equal to |  |
| == | Relational is equal to |  |
| != | Relational is not equal to |  |
| & | Bitwise AND | Left to right |
| ^ | Bitwise exclusive OR | Left to right |
| | | Bitwise inclusive OR | Left to right |
| AND | Logical AND | Left to right |
| OR | Logical OR | Left to right |
| ?: | Ternary conditional | Right to left |
| = | Assignment | Right to left |
| += | Addition assignment |  |
| -= | Subtraction assignment |  |
| \*= | Multiplication assignment |  |
| /= | Division assignment |  |
| %= | Modulus assignment |  |
| &= | Bitwise AND assignment |  |
| ^= | Bitwise exclusive OR assignment |  |
| |= | Bitwise inclusive OR assignment |  |
| <<= | Bitwise left shift assignment |  |
| >>= | Bitwise right shift |  |
| , | Comma | Left to right |

## Statement

An expression is the smallest single standalone element in a program. It consists of expressions or other statements. Expressions and statements differs from each other because the former yields a value or values after its computation, while the latter expresses actions that must be carried out.

The Epsilon has the following types of statement:

* Assignment
* Procedure call
* Control flow
* Macro statement
* Variable declaration

All Epsilon statement must be terminated by a newline (CR+LF or \r\n) character.

<stmt> → <ass\_stmt> | <proc\_call> | <macro\_stmt> **CR+LF**

The assignment statement of Epsilon has the following grammar rules:

<ass\_stmt> → {<var\_id> := }⁺ <expr> **CR+LF**

# Control Flow

## Selection

The selection statement provides means of choosing between two or more execution paths.

### The Two-Way Selection Statement

The two-way selection statement of Epsilon has the following grammar rules:

<if\_stmt> → **if** <condition> **then** <stmt\_list> {**elseif** <logic\_expr> **then**

<stmt\_list>} [**else** <stmt\_list>] **end if CR+LF**

<condition> → [<var\_id> = ]<expr>

<stmt\_list> → {<stmt><stmt\_trmntr>}

### Multiple-Selection Statement

<switch\_stmt> → **switch** <var\_id> **CR+LF**

{**case** (<var\_id> | <literal>), {(<var\_id> | <literal>)}**:**

<stmt\_list> **break** **CR+LF** }

[**default:** <stmt\_list> **break CR+LF**]

**end switch CR+LF**

## Iteration

### The while Loop

<while\_loop> → **while** <condition> **CR+LF** <stmt\_list> **end while CR+LF**

### The do-while Loop

<do-while\_loop> → **do** <stmt\_list> **while** <condition> **CR+LF**

### The for Loop

<for\_loop> → **for** [type] <var\_id> [**= (**<literal> | <var\_id>)] **to** **(**<literal> |

<var\_id>) **CR+LF**

<stmt\_list>

**next** **CR+LF**

## Exception Handling

# Procedures