The uC25 Language Specification

Contents

https://eecs390.github.io/uc-language/

1 The uC25 Language Specification

This document was generated on 2024-12-26.

2 Lexical Structure

Excepting comments and string literals, programs must be written in the subset of ASCII consisting of the following 93 characters:

- the 62 alphanumeric characters: `0-9`, `a-z`, `A-Z`
- the 25 symbols: `+ - * / % | & ! < > = # ( ) [ ] { } , . ; :  " \ _`
- the 6 whitespace characters: space, horizontal tab, carriage return, new line, vertical tab, form feed

A source program may consist of whitespace, comments, and tokens.

```
ProgramText:
    TextElements

TextElements:
    TextElement
    TextElememts TextElement

TextElement:
    Whitespace
    Comment
    Token
```

Tokens consist of keywords, identifiers, literals, operators, and delimiters.

```
Token:
    Keyword
    Identifier
    Literal
```

(continues on next page)

(continued from previous page)

```
    Operator
    Delimiter
```

2.1 Whitespace

Tokens may be separated by any of the 6 whitespace characters.

```
Whitespace:
    space
    horizontal tab
    carriage return
    vertical tab
    form feed
    NewLine

NewLine: the new-line character
```

2.2 Comments

There are two kinds of comments, a *delimited* comment and an *end-of-line comment*.

```
Comment:
    DelimitedComment
    EndOfLineComment

DelimitedComment:
    / * CommentChars * /
    / * * /

CommentChars:
    *
    * CommentCharsNoSlash
    CommentCharNoStar CommentChars
    CommentCharNoStar

CommentCharNoStar: any ASCII character except *

CommentCharsNoSlash:
    CommentCharNoSlash CommentChars
    CommentCharNoSlash

CommentCharNoSlash: any ASCII character except /

EndOfLineComment:
    / / CharactersNoNewLine NewLine
    / / NewLine

CharactersNoNewLine:
    CharacterNoNewLine
    CharactersNoNewLine CharacterNoNewLine
```

(continues on next page)

(continued from previous page)

```
CharacterNoNewLine: any ASCII character except the new line
```

2.3 Keywords

The following character sequences are reserved keywords and may not be used as identifiers.

```
Keyword: one of
    if else while for struct break continue return assert new
```

2.4 Identifiers

Identifiers consist of a sequence of characters begininning with a lower-case letter (`a-z`) or upper-case letter (`A-Z`). The remaining characters may consist of underscores, lower-case letters, upper-case letters, and digits (`0-9`).

```
Identifier: except Keyword, BooleanLiteral, and NullLiteral
    IdentifierStartCharacter
    IdentifierStartCharacter IdentifierCharacters

IdentifierStartCharacter:
    LowerCaseLetter
    UpperCaseLetter

IdentifierCharacters:
    IdentifierCharacter
    IdentifierCharacters IdentifierCharacter

IdentifierCharacter:
    IdentifierStartCharacter
    _
    Digit

LowerCaseLetter: one of
    a b c d e f g h i j k l m n o p q r s t u v w x y z

UpperCaseLetter: one of
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Digit: one of
    0 1 2 3 4 5 6 7 8 9
```

2.5 Literals

Literals include integer, floating-point, string, boolean literals, and the null literal.

```
Literal:
    IntegerLiteral
    FloatingLiteral
    StringLiteral
    BooleanLiteral
    NullLiteral
```

Integer Literals

Integer literals must be expressed in decimal format. They consist of a sequence of one or more decimal digits, followed by an optional lower- or upper-case `L`. Integer literals with an `l` or `L` suffix are of type `long`, while those without are of type `int`.

```
IntegerLiteral:
    Digits IntegerSuffix
    Digits

Digits:
    Digit
    Digits Digit

IntegerSuffix: one of
    l L
```

It is a compile-time error if an integer literal represents a value that is too large for its type. For an `int`, the largest valid literal is `2147483647` ($2^{31} - 1$), and for a `long`, the largest valid literal is `9223372036854775807L` ($2^{63} - 1$).

Integer literals are always nonnegative: a sequence such as `-42` consists of the unary `-` operator applied to the subsequent literal value. The minimal `int` value can be expressed by `(-2147483647-1)`, and the minimal `long` value by `(-9223372036854775807L-1)`.

Floating-Point Literals

A floating-point literal consists of a digit sequence consisting of a *significand* and optional *exponent*. The significand is a sequence of digits containing a single period (`.`). If an exponent is provided, the period (`.`) may be elided from the significand. The exponent consists of the character `e` followed by an optional sign and a non-empty digit sequence. Floating-point literals are of type `double`.

```
FloatingLiteral:
    SignificandWithPeriod
    SignificandWithPeriod Exponent
    Digits Exponent

SignificandWithPeriod:
    Digits .
    . Digits
    Digits . Digits
```

(continues on next page)

(continued from previous page)

```
Exponent:
    e Sign Digits
    e Digits

Sign: one of
    + -
```

String Literals

A string literal consists of a sequence of characters surrounded by double quotes ("). Any ASCII character except for the new-line character may appear in a string literal. The backslash (\) and double-quote (") characters may only appear as part of an escape sequence. A string literal has type `string`.

```
StringLiteral:
    " StringCharacters "
    " "

StringCharacters:
    StringCharacter
    StringCharacters StringCharacter

StringCharacter:
    UnescapedCharacter
    EscapedCharacter

UnescapedCharacter: any ASCII character except \, ", or new line

EscapedCharacter: any of the escape sequences below
```

The following escape sequences represent special characters:

Escape Sequence	Character
\"	double quote
\\	backslash
\a	audible bell
\b	backspace
\n	new line
\t	horizontal tab
\f	form feed
\r	carriage return

Boolean Literals

The character sequences `true` and `false` represent boolean literals, which have type `boolean`.

```
BooleanLiteral: one of
    true false
```

Null Literal

The character sequence `null` represents a null literal, which has the `null` type.

```
NullLiteral:
    null
```

2.6 Operators

The following 20 characters and character sequences represent operators.

```
Operator: one of
    + - * / % || && ! < > <= >= == != = ++ -- # << >>
```

2.7 Delimiters

The following characters are delimiters.

```
Delimiter: one of
    ( ) [ ] { } , . ; :
```

3 Program Structure

A uC program consists of a single source file, conventionally with extension `.uc`. A program consists of a sequence of type and function declarations.

```
Program:
    Declarations

Declarations:
    Declarations Declaration
    Empty

Declaration:
    FunctionDeclaration
    TypeDeclaration

Empty: empty
```

A program must have exactly one function with the following declaration:

```
void main(string[] args) {
  ...
}
```

The `main()` function is the entry point of a uC program. The `args` array contains the *command-line arguments* provided to the program.

3.1 Type Declarations

A type declaration consists of the `struct` keyword, followed by the name of the type and an open curly brace, followed by a list of field declarations, terminated by a closing curly brace and semicolon.

```
TypeDeclaration:
    struct Identifier { FieldDeclarations } ;

FieldDeclarations:
    FieldDeclarations FieldDeclaration
    Empty

FieldDeclaration:
    Type Identifier ;
```

As an example, the following declares a type containing an integer and a floating-point field:

```
struct foo {
  int x;
  double y;
};
```

It is a compile-time error for multiple fields to be given the same name. It is a compile-time error for a type declaration to declare a type with the same name as an existing built-in or user-defined type. It is *not* an error for a field to have the same name as a built-in or user-defined type or function.

A type can be used anywhere in the source program, including prior to its declaration.

3.2 Function Declarations

A function declaration consists of a return type, followed by the name of the function, a parameter list, and a body consisting of a block of statements.

```
FunctionDeclaration:
    Type Identifier ( ParametersOpt ) Block

ParametersOpt:
    Parameters
    Empty

Parameters:
    Parameter
    Parameters , Parameter
```

(continues on next page)

(continued from previous page)

```
Parameter:
    Type Identifier
```

As an example, the following declares a function that takes an integer argument and returns a string:

```
string bar(int i) {
  ...
}
```

A function can be used anywhere in the source program, including prior to its declaration.

It is a compile-time error for a function declaration to declare a function with the same name as an existing built-in or user-defined function. It is *not* an error for a function to have the same name as a built-in or user-defined type.

The scope of a function parameter is the entire body of the function.

It is a compile-time error for two parameters of a function to have the same name. It is *not* an error for a parameter to have the same name as a built-in or user-defined type or function.

4 Types

The types in a uC program consist of primitive types, user-defined types, and array types.

```
Type:
    Identifier
    ArrayType
```

4.1 Primitive Types

The following primitive types are available to uC programs:

- `int`: a 32-bit integer, representing values in the range $[-2^{31}, 2^{31} - 1]$.
- `long`: a 64-bit integer, representing values in the range $[-2^{63}, 2^{63} - 1]$.
- `double`: an IEEE754-compliant, double-precision floating-point number.
- `boolean`: a truth value, either `true` or `false`.
- `string`: a sequence of zero or more characters.
- `void`: denotes the return type of a function that does not return a value. It is a compile-time error to use `void` as a type in any other context.
- `null`: the type of the `null` literal. This type is syntactically prevented from being used anywhere other than the `null` literal.

Primitive types have *value semantics*, meaning that they are stored directly in variables, parameters, fields, and array elements. It is a compile-time error to use the `new` keyword with a primitive type.

4.2 User-Defined Types

User-defined types are those introduced by *Type Declarations*. Such types have *reference semantics*, meaning that a variable, parameter, field, or array element of user-defined type is either an indirect reference (i.e. a pointer) to an object of the given type or contains the special `null` reference.

4.3 Arrays

An array is a homogeneous, sequential collection of values. An array type is denoted by an element type followed by an empty pair of square brackets.

```
ArrayType:
    Type [ ]
```

The element type may be any primitive, user-defined, or array type, excluding the `void` and `null` types.

Arrays have reference semantics, so a variable, parameter, field, or array element of array type holds either an indirect reference to an array object or the special `null` reference.

An array has a `length` field, which contains the number of elements in the array.

4.4 Type Compatibility

The following implicit conversions are allowed:

- conversion of a value of type `int` to `long` or `double` type
- conversion of a value of type `long` to type `double`
- conversion of the `null` literal to any *reference type* (user-defined and array types)

Except for the conversions above, the type of a value must exactly match the type expected by the context in which the value is used. Specifically:

- the type of a return value must match or be implicitly covertible to the function's return type
- the type of an argument passed to a function call must match or be implicitly convertible to the type of the corresponding function parameter
- the type of the value on the right-hand side of an assignment must match or be implicitly convertible to the type of the object on the left-hand side
- the type of the value in the initialization expression of a variable definition must match or be implicitly convertible to the type of the variable

5 Functions

The functions in a uC program consist of built-in functions, which are available to all uC programs, and user-defined functions.

5.1 Built-in Functions

The following functions convert between numeric types:

- `long int_to_long(int)`
- `double int_to_double(int)`
- `int long_to_int(long)`
- `double long_to_double(long)`
- `int double_to_int(double)`
- `long double_to_long(double)`

The following functions convert between `string`s and other primitive types:

- `string int_to_string(int)`
- `string long_to_string(long)`
- `string double_to_string(double)`
- `string boolean_to_string(boolean)`
- `int string_to_int(string)`
- `long string_to_long(string)`
- `double string_to_double(string)`
- `boolean string_to_boolean(string)`

It is a runtime error to pass a `string` that does not consist of a valid representation of the input type to the functions that convert to `string`.

The following functions are defined on `string`s:

- `int length(string)`: returns the length of the input `string`.
- `string substr(string, int, int)`: produces a `string` that contains a subsequence of the input `string`. The second argument is the start position, which must be in the range `[0,length(string)-1]`. The third argument is the length of the subsequence, which must be non-negative. The length is truncated if the input string contains insufficient characters.
- `int ordinal(string)`: returns the ASCII value of the given single-character string. If the given string does not consist of exactly one character, -1 is returned.
- `string character(int)`: returns a string containing the character with the given ASCII value. If the given argument is not in the range $[1, 127]$, an empty string is returned.

The following numerical functions are defined:

- `double pow(double, double)`: computes the first argument raised to the power of the second.
- `double sqrt(double)`: computes the square root of the argument. The argument must be non-negative.
- `double ceil(double)`: computes a floating-point representation of the ceiling of the argument.
- `double floor(double)`: computes a floating-point representation of the floor of the argument.

The following printing functions are defined:

- `void print(string)`: prints the given string to standard output, without a trailing new-line character.
- `void println(string)`: prints the given string to standard output, with a trailing new-line character.

The following input functions are defined:

- `string peekchar()`: returns the next character that is in standard input, without removing it from the stream. Returns an empty string if the stream is at end-of-file.
- `string readchar()`: returns the next character that is in standard input, removing it from the stream. Returns an empty string if the stream is at end-of-file.
- `string readline()`: returns the line that is in standard input, removing it from the stream. The trailing newline character, if there is one, is included in the resulting string. Returns an empty string if end-of-file is reached before any characters are read.

The following exit function is defined:

- `void exit(int)`: immediately halts program execution and exits with the given return code.

5.2 User-Defined Functions

User-defined functions are introduced by *Function Declarations*. A user-defined function consists of a return type, a name, parameters, and a block of statements constituting the body. Calling a user-defined function binds the values of the argument expressions to the parameter names in a fresh environment and executes the body in the subsequent environment. Functions are lexically scoped and do not have access to the caller's environment.

It is a compile-time error (no diagnostic required) for control to reach the end of a function whose return type is not `void`. A function that returns `void` may have bare `return` statements without an expression, or `return` statements with expressions that have `void` type. A function that returns any other type must have `return` statements with an expression whose type matches or is implicitly convertible to the return type.

6 Statements

Statements in uC include blocks, variable definitions, conditionals, loops, control statements (break, continue, and return), assert statements, and expression statements.

```
Statement:
    Block
    VariableDefinitionStatement
    IfStatement
    WhileStatement
    ForStatement
    BreakStatement
    ContinueStatement
    ReturnStatement
    AssertStatement
    ExpressionStatement
```

6.1 Blocks

A block is a compound statement that itself consists of a sequence of zero or more statements, enclosed by curly braces. A block executes each of its statements in order. It terminates normally when control reaches the end of its last statement. A control statment within a block may cause *abrupt termination*, where execution of the block is immediately terminated.

```
Block:
    { Statements }
```

(continues on next page)

(continued from previous page)

```
Statements:
    Statements Statement
    Empty
```

6.2 Variable Definitions

A variable definition consists of a type, followed by an identifier, an equals sign, and an initialization expression.

```
VariableDefinitionStatement:
    VariableDefinition ;

VariableDefinition:
    Type Identifier = Expression
```

The scope of a variable introduced by a variable-definition statement encompasses the variable's initialization as well as all subsequent statements in the block in which the variable definition is located. It is a compile-time error if an existing parameter or variable of the same name is in scope when a variable is defined (i.e. if the variable *shadows* an existing parameter or variable). It is a compile-time error for the initialization expression to refer to the variable being defined.

A variable definition binds the given identifier in the current environment to the result of evaluating the initialization expression. The type of the initialization expression must match or be implicitly convertible to the type specified in the variable definition.

6.3 Conditionals

A conditional consists of the keyword `if`, a parenthesized expression, a block, and optionally, an `else` keyword followed by a another block.

```
IfStatement:
    if ( Expression ) Block
    if ( Expression ) Block else Block
    if ( Expression ) Block else IfStatement
```

The expression constitutes the test of the conditional, and its type must be `boolean`. If the expression evaluates to true, the block immediately following the test is executed. Otherwise, the block or conditional following the `else` keyword is executed, if present.

6.4 Loops

A while loop consists of the `while` keyword, a parenthesized expression, and a block.

```
WhileStatement:
    while ( Expression ) Block
```

The expression constitutes the test of the while loop, and its type must be `boolean`. If the test expression evaluates to true, the block is executed, after which control normally returns to the beginning of the loop. If the test expression evaluates to false, execution proceeds past the loop.

A for loop consists of the `for` keyword, followed by a parenthesized list consisting of an initialization, a test expression, and an update expression, separated by semicolons, followed by a block. Any of the expressions in the parenthesized list may be omitted, but the separating semicolons must still be present.

```
ForStatement:
    for ( ForInitialization ; ExpressionOpt ; ExpressionOpt ) Block

ForInitialization:
    VariableDefinition
    ExpressionOpt

ExpressionOpt:
    Expression
    Empty
```

The initialization can be a variable definition or an arbitrary expression. If it is a variable definition, the scope of the variable consists of the entirety of the for loop, including the variable's own initialization expression. As in variable-definition statements, it is a compile-time error if an existing parameter or variable of the same name is in scope when a variable is defined in a `for` initialization, or for the initialization expression of the variable to refer to the variable being defined.

If a for loop contains an initialization, the initialization is performed upon first reaching the loop.

The middle expression constitutes the test, and its type must be `boolean` if it is present. If the test expression is omitted or evaluates to true, the block is executed.

The last expression is the loop update. If present, it is evaluated after each normal execution of the block. Following normal termination of the block and evaluation of the update, control returns to the beginning of the for loop, prior to evaluating the test expression. If the test expression is present and evaluates to false, execution proceeds past the loop.

If execution reaches a return statement within a while or for loop, the loop is abruptly terminated.

Execution of a while or for loop can also be controlled with a break or continue statement, which can only appear within the body of a loop.

```
BreakStatement:
    break ;

ContinueStatement:
    continue ;
```

A break statement causes execution of the loop that immediately contains it to abruptly terminate. A continue statement causes execution of the loop that immediately contains it to abruptly return to the beginning of the loop, prior to evaluating the test expression. In a for loop, the update expression, if present, is evaluated before execution is returned to the test expression.

6.5 Return Statements

A return statement consists of the `return` keyword, followed by an optional expression, followed by a semicolon.

```
ReturnStatement:
    return Expression ;
    return ;
```

If execution reaches a return statement, the current function is abruptly terminated. If the return type of the current function is `void`, then the expression may be elided. If the return type is any other type, then the expression must be

present. In either case, if the expression is present, the type of the expression must match or be implicitly convertible to the return type.

6.6 Assert Statements

An assert statement consists of the `assert` keyword, followed by a test expression, optionally followed by a colon and an additional expression, followed by a semicolon.

```
AssertStatement:
    assert Expression ;
    assert Expression : Expression ;
```

The test expression must have type `boolean`. The second expression, if provided, must have type `string`.

An assert statement evaluates the test expression and generates a runtime error if the expression evaluates to false. If the second expression is provided, it is included as part of the error message.

6.7 Expression Statements

An expression statement consists of an expression followed by a semicolon.

```
ExpressionStatement:
     Expression ;
```

An expression statement evaluates the given expression and discards its value.

7 Expressions

An expression is evaluated to produce a value. All expressions in a uC program have a type, and the type must be valid for the context in which the expression is used.

```
Expression:
    ParenthesizedExpression
    Literal
    NameExpression
    CallExpression
    NewExpression
    FieldAccessExpression
    ArrayIndexExpression
    UnaryPrefixOperation
    BinaryOperation
```

Any expression can be parenthesized, which can be used to override the default associativity and precedence of a sequence of operations.

```
ParenthesizedExpression:
    ( Expression )
```

7.1 Simple Expressions

A literal is an expression that evaluates to the value represented by the literal.

```
Literal:
    IntegerLiteral
    FloatingLiteral
    StringLiteral
    BooleanLiteral
    NullLiteral
```

The type of an integer literal is `long` if it ends with the `l` or `L` suffix, otherwise it is `int`. The type of a floating-point literal is `double`, a string literal is `string`, a boolean literal is `boolean`, and the null literal is `null`.

An identifier is an expression, and it must name a variable or parameter in the local scope. It is an l-value and can be used in an l-value context. When used in a context that expects a value, it evaluates to the value to which the variable or parameter is bound.

```
NameExpression:
    Identifier
```

The type of an identifier expression is the type of the variable or parameter it names.

7.2 Function Calls

A function call consists of an identifier followed by a parenthesized list of arguments, separated by commas.

```
CallExpression:
    Identifier ( ArgumentsOpt )

ArgumentsOpt:
    Arguments
    Empty

Arguments:
    Expression
    Arguments , Expression
```

The identifier must name a built-in or user-defined function. The number of arguments must match the number of function parameters, and each argument type must match or be implicitly convertible to the corresponding parameter type.

A function call evaluates the arguments in an indeterminate order, binds the argument values to the function parameters in a fresh environment, and executes the body of the function in the given environment.

The type of a function call is the return type of the invoked function.

7.3 Allocation Expressions

An allocation expression allocates space for an object of user-defined type or an array and initializes the object or array. It consists of the keyword `new`, followed by a type, followed by a list of zero or more argument expressions enclosed by parentheses or curly braces and separated by commas.

```
NewExpression:
    new Type ( ArgumentsOpt )
    new Type { ArgumentsOpt }
```

The type must name a user-defined or array type.

If the given type is user-defined, the allocation constructs a new object of that type. The arguments may be empty, in which case the fields of the newly created object undergo *Default Initialization*. Otherwise, the number of arguments must match the number of fields in the given type, and the argument types must match or be implicitly convertible to the corresponding field types. The fields of the newly created object are then initialized with the argument values. The allocation expression evaluates to a reference to the new object, and the type of the expression is that named by the given type.

If the given type is an array type, the allocation creates an array of the given type. Any number of argument expressions may be provided. The type of each argument must match or be implicitly convertible to the element type of the array. The array is initialized with the argument values, and it has initial length equal to the number of arguments. The allocation expression evaluates to a reference to the new array, and the type of the expression is the given array type.

The order of evaluation of the arguments of an allocation expression is indeterminate.

7.4 Field Access

A field-access expression consists of an expression denoting the receiver, followed by a period, followed by an identifier.

```
FieldAccessExpression:
    Expression . Identifier
```

The receiver expression preceding the period must be of user-defined or array type. It is a runtime error if the receiver expression evaluates to a null reference.

If the receiver is of array type, only the identifier `length` is valid, in which case the field-access expression has type `int` and evaluates to the length of the array produced by the receiver expression. The field-access expression is not an l-value in this case.

If the receiver is of user-defined type, the identifier must name a field defined by the type. The field-access expression has type associated with the given field. It evaluates to the l-value associated with the field in the object produced by the receiver expression. If used in a context that requires a value, then it evaluates to the value bound to that field.

7.5 Array Indexing

An array-indexing expression consists of an expression denoting the receiver, following by an open square bracket, an index expression, and a close square bracket.

```
ArrayIndexExpression:
    Expression [ Expression ]
```

The receiver expression preceding the square brackets must be of array type. It is a runtime error if the receiver evaluates to a null reference.

The index expression enclosed by the square brackets must be of type `int`. It is a runtime error if it evaluates to a value that is negative or greater than or equal to the length of the receiver array.

The array-indexing expression has type corresponding to the element type of the receiver, and it evaluates to the l-value associated with the array element at the given index. If used in a context that requires a value, then it evaluates to the value of the given element.

The order of evaluation of the receiver and index expressions is indeterminate.

7.6 Unary Operations

Unary prefix operations consist of sign operations, logical complement, increment, and decrement.

```
UnaryPrefixOperation:
    + Expression
    - Expression
    ! Expression
    ++ Expression
    -- Expression
    # Expression
```

The sign operations, `+` and `-`, require the given expression to be of numeric type (`int`, `long`, or `double`), and the type of the sign expression is the same as its subexpression. The `+` operator has no effect on its operand, while the `-` operator negates it.

The complement operator `!` requires its operand expression to be of `boolean` type, and it results in a `boolean`. The complement expression evaluates to the complement of the value produced by the operand.

The increment and decrement operators, `++` and `--`, require the given operand to be an l-value of numeric type. The increment and decrement expressions increment and decrement, respectively, the object denoted by the operand. The result is the new value of the operand object, and the type is the same as that of the operand.

The ID operator `#` requires the given operand to be of reference type, and it produces a value of type `long`. If the operand is not a null reference, the resulting value represents the identity of the referenced object, and it is guaranteed to be unique during the lifetime of the object. If the operand is a null reference, then the resulting value is zero, which is guaranteed to be distinct from the identity of any valid object.

7.7 Binary Operations

Binary infix operations consist of arithmetic operations, logical operations, arithmetic comparisons, equality comparisons, assignment, and array push and pop operations.

```
BinaryOperation:
    ArithmeticOperation
    LogicalOperation
    Comparison
    EqualityTest
    Assignment
    ArrayOperation
```

The order of evaluation of the operands in a binary operation is indeterminate, except in logical operations.

Arithmetic Operations

Arithmetic operations include addition, subtraction, multiplication, division, and modulo.

```
ArithmeticOperation:
    Expression + Expression
    Expression - Expression
    Expression * Expression
    Expression / Expression
    Expression % Expression
```

The operands to binary `-`, `*`, `/`, and `%` must be of numeric type. If the two operands have the same type, then the type of the overall expression also has that type. Otherwise, it is the case that the type of one operand is implicitly convertible to the type of the other, in which case the type of the overall expression is the latter.

The division operator `/`, when the result is of integer type (`int` or `long`), truncates the result. In other words, it rounds the result toward zero. The result value is undefined if the second operand evaluates to zero.

The operands of the modulo operator `%` must have integer type (`int` or `long`). The result is the remainder from dividing the first operand value by the second, preserving the sign of the first operand value. The result is undefined if the second operand evaluates to zero.

The operands to the + operator must be of primitive type, but cannot be of `null` or `void` type. If the operands are both of numeric type, the result type is as for the other arithmetic operators. If one operand is of `boolean` type, then the other operand must be of `string` type. If one or both operands is of `string` type, then the operator performs string concatenation, and the non-`string` operand, if there is one, is implicitly converted to a `string` using the appropriate built-in conversion function. The result of string concatentation has type `string`.

The result of an arithmetic operation on integer types is undefined if the computed value lies outside the range of the type of the overall expression.

Logical Operations

Logical operations consist of disjunction (`||`) and conjunction (`&&`).

```
LogicalOperation:
    Expression || Expression
    Expression && Expression
```

The operands must be of type `boolean`, and the result is also of type `boolean`.

The logical operations are short circuiting. The left-hand operand is evaluated first, and if it is true in the case of disjunction or false in conjunction, the right-hand operand is not evaluated, and the result is the value of the left-operand. Otherwise, the right-hand operand is evaluated, and its value is the result of the operation.

Comparisons

Comparisons consist of the operators <, <=, >, and >=.

```
Comparison:
    Expression < Expression
    Expression <= Expression
    Expression > Expression
    Expression >= Expression
```

The operands must both be of numeric type or both be of `string` type. The result is of type `boolean`. Numeric values are compared numerically, while strings are compared lexicographically. Comparison operations are prohibited from being chained together, both by associativity and by type rules.

Equality Tests

Equality comparisons include the == and != operators.

```
EqualityTest:
    Expression == Expression
    Expression != Expression
```

The operands must either have the same type, or the type of one operand must be implicitly convertible to the type of the other. The result is of type `boolean`.

An equality comparison for a reference type compares the two objects by contents rather than by pointer equality. If one of the two operands evaluates to a null reference, the result is true if the other operand also evaluates to a null reference and false otherwise.

For user-defined types, objects are compared field by field, which are recursively compared. Two objects of a user-defined type with no fields always compare equal. Arrays are compared by length and then by the individual elements. Two empty arrays always compare equal.

The result of an equality comparison for a reference type is undefined if either of the operands contains a circular reference.

Assignment

The assignment operation is as follows:

```
Assignment:
    Expression = Expression
```

The type of the right-hand expression must match or be implicitly convertible to the type of the left-hand side. The left-hand expression must produce an l-value. Assignment replaces the value of the object produced by the left-hand operand with the value produced by the right-hand. The assignment operation as a whole has the same type as the left-hand operand, and it produces the new value of the left-hand operand.

Array Operations

Arrays support push (<<) and pop (>>) operations.

```
ArrayOperation:
    Expression << Expression
    Expression >> Expression
```

A push operation appends the value produced by the right-hand expression to the end of the array produced by the left-hand expression. The left-hand operand must be of array type, and the type of the right-hand operand must match or be implicitly convertible to the element type of the left-hand array type. The result of the push operation has the same type as the left-hand operand, and the value is the same array as that produced by the left-hand expression. It is a runtime error if the left-hand operand evaluates to null.

A pop operation removes the last item from the left-hand array and assigns it to the object produced by the right-hand operand, or discards it if the right-hand operand is the `null` literal. The left-hand expression must be of array type.

The right-hand operand must be either `null` or an l-value. In the latter case, the element type of the array must match or be implicitly convertible to the type of the right-hand expression. The result of the pop operation has the same type as the left-hand operand, and the value is the same array as that produced by the left-hand expression. It is a runtime error if the left-hand operand evaluates to null or an empty array.

7.8 Associativity and Precedence

Operators in uC have the following associativity and precedence, ordered from lowest precedence to highest:

Precedence Class	Operators	Associativity
1	<<, >>	left
2	=	right
3	||	left
4	&&	left
5	==, !=	non-associative
6	<, <=, >, >=	non-associative
7	infix +, -	left
8	*, /, %	left
9	prefix +, -, !, ++, --, #	right
10	., [ ]	left

Non-associative operators of a single precedence class cannot be chained together within a single expression.

8 Memory Management

Implementations of uC are required to perform automatic memory management, i.e. garbage collection, for objects of reference type.

8.1 Default Initialization

Objects of user-defined type constructed by the zero-argument default constructor are default initialized. In default initialization, an object's fields are set to default values. A field of primitive numeric type is set to zero, a field of `boolean` type is set to false, and a field of `string` type is set to an empty string. Fields of reference type, including array type, are set to null.

9 Revision History

- uC25 (December 2024)
  - Allow blocks as statements.
  - Allow return statements in `void` functions to return `void` expressions.
  - Add assert statements.
  - Clarify that arithmetic operations on integers results in an undefined value upon over/underflow.
  - Add `exit()` function.
- uC24 (December 2023)

  - Rename floating-point type from `float` to `double`.
  - Change syntax of type declarations to more closely match C-like languages.
  - Change syntax of local-variable declarations to more closely match C-like languages.
- uC23 (December 2022)
  - Change behavior of prefix increment/decrement and assignment so that they no longer produce l-values.
  - Specify valid range of integer literals.
- uC22 (January 2022)
  - Change syntax of allocations to unify user-defined and array allocations.
- uC18 (November 2018)
  - Add cascading `if`s.
  - Add `for` loops.
  - Add ID operator.
  - Explicitly permit variables to have the same name as a type or function.
  - Specify that equality comparisons are undefined in the presence of circular references.
- uC17 (November 2017)
  - Add input functions (`peekchar()`, `readchar()`, `readline()`).
  - Specify that the order of evaluation of the arguments of an allocation expression is indeterminate.
- uC16 (November 2016): original version