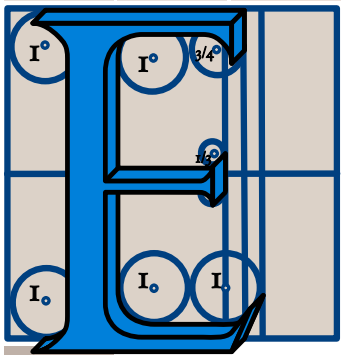
Euclides 4.0

The Language for Generative Modeling & Design Automation

Torsten Ullrich

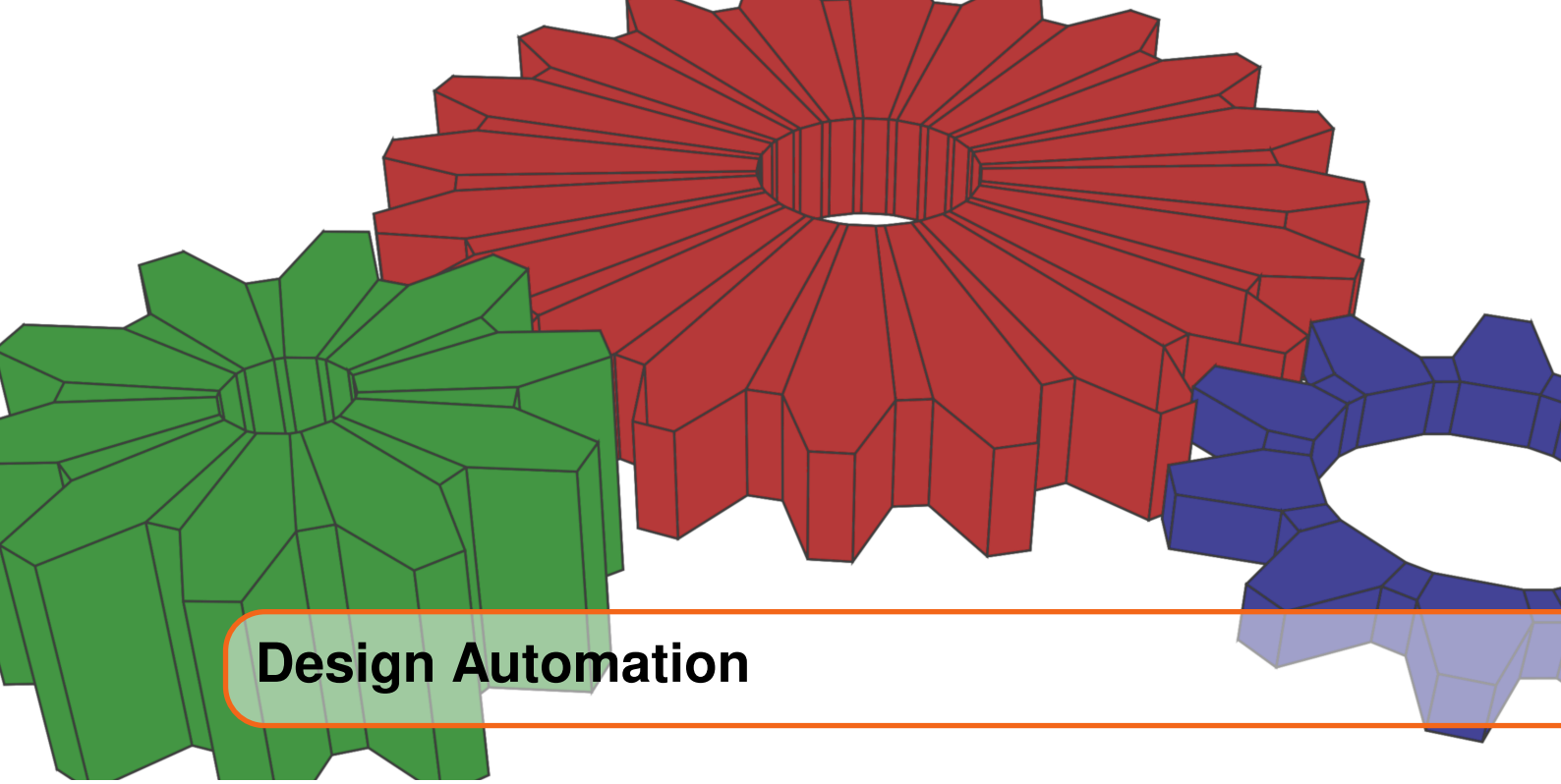


Fraunhofer Austria Research GmbH & Technische Universität Graz

www.fraunhofer.at

2nd Edition, August 2019

Contents



Design Automation

The idea of design automation is to allow the generation of highly complex objects based on a set of formal construction rules. Using these construction rules, an object is described by a sequence of processing steps, rather than just by the result of all applied operations: Shape design becomes rule design. Due to its very general nature, this approach can be applied to any domain and to any shape representation that provides a set of generating functions [KSU16].

In the context of computer-aided design (CAD) and shape description, the digital creation of a shape is called modeling. The most common representation of a shape is a composition of elementary objects. However, a shape can also be described by its generating process. In this case, the description is called a generative model. A generative model does not describe a shape by the parts it consists of, but by the operations and steps needed to be performed in order to create it; i.e., a generative model is an algorithm. Its implementation is an algorithmic description written in a programming language. Depending on the used software engineering paradigm, a generative model may also be called a procedural model or a functional model, if the algorithm is implemented in a procedural way, respectively functionally. For many purposes in CAD, the mightiness of a Turing-complete programming language may lead to potential problems, such as the halting problem. In order to avoid these problems, CAD frameworks often offer a language that is not Turing-complete; i.e., the set of language features is reduced to parametric modeling.

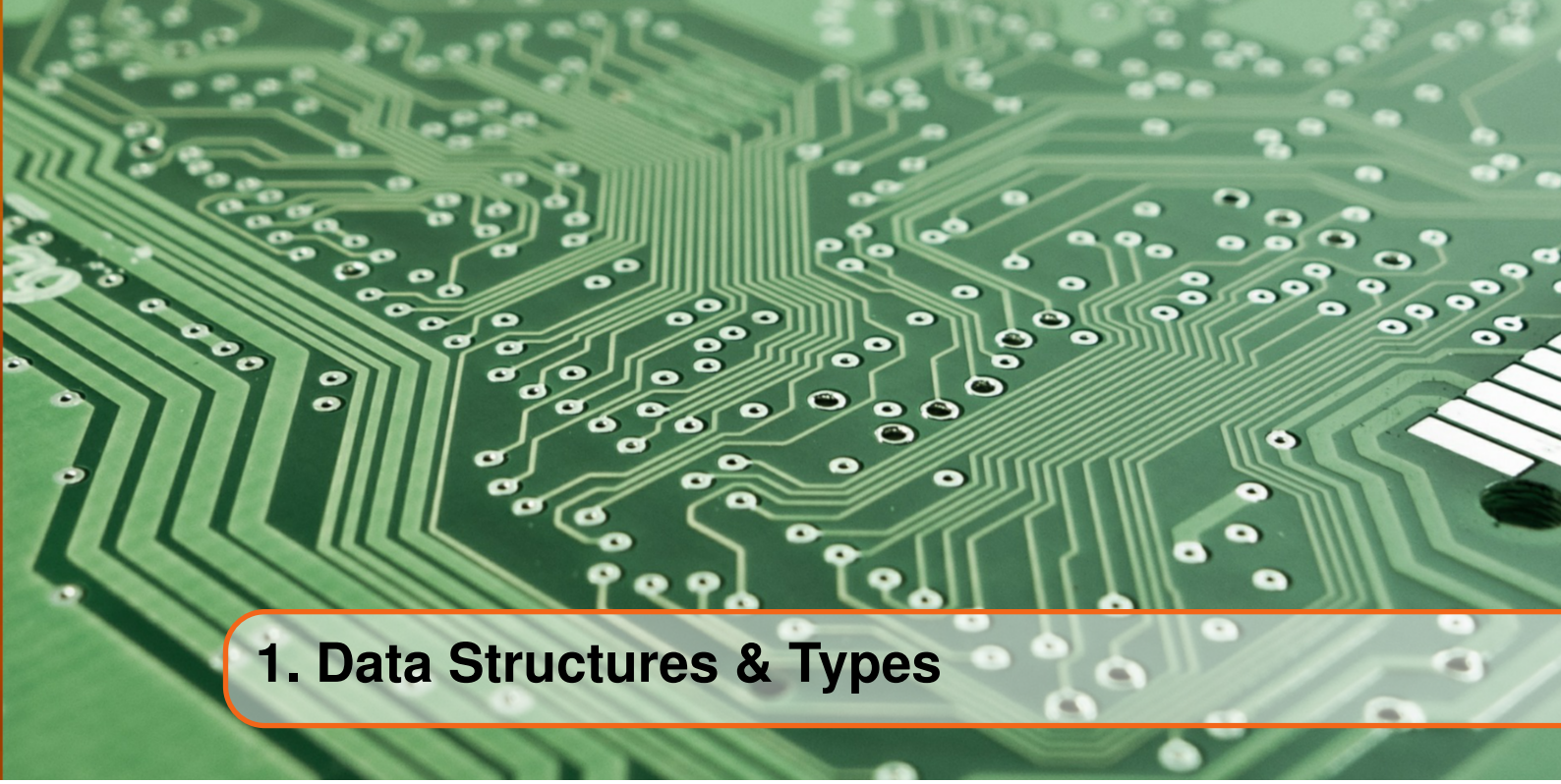
The description of a construction process has a long history. Geometry from the days of the ancient Greeks placed great emphasis on problems of constructing various geometric figures using only a ruler without markings (to draw lines) and a compass (to draw circles). Ruler-and-compass constructions are based on Euclid's axioms [HF07] using points, lines and circles that have already been constructed. The resulting geometric primitives together with the ruler-and-compass constructions are the first algorithmic descriptions of generative models. Euclid's Elements is probably the most successful textbook ever written. The long history of geometric constructions is also reflected in the history of civil

engineering and architecture. Their complexity is achieved by combining only a few basic geometric patterns. In this way it is possible that complex configurations can be obtained from elementary constructions. The different combinations of specific parametric features can be grouped together, leading to the concept of styles. As a consequence, a generative description can be an extremely compact representation for a whole class of shapes. In these cases generative modeling is superior to conventional approaches. Its compact description does not depend on the counter of primitives but on the model's complexity itself. Particularly large scale models and scenes – such as plants, buildings, cities, and landscapes – can be described efficiently. Therefore, generative descriptions make complex models manageable as they allow identifying a shape's high-level parameters. Another advantage of procedural modeling techniques is the included expert knowledge within an object description; e.g., classification schemes used in architecture, archaeology, civil engineering, etc. can be mapped to procedures. For a specific object only its type and its construction parameters have to be identified.

A disadvantage of many generative modeling systems is the dependency on a particular platform. A generative description usually depends on the programming language it is implemented in and on the used geometry representation. EUCLIDES overcomes this problem: EUCLIDES is a meta-modeler for procedural modeling based on the programming language JavaScript [SSUF10a]. It differs from other modeling environments in a very important aspect: target independence. Usually, a generative modeling environment consists of a script interpreter and a 3D rendering engine. A generative model (3D data structures with functionality) is interpreted directly to generate geometry, which is then visualized by the rendering engine. In our system a model's source code is not interpreted but parsed into an intermediate representation, an abstract syntax tree. The consistent intermediate representation serves as a basis for back-end exporters to different languages, different target platforms and for different purposes [SSUF10b].

This manual gives an introduction to the geometric language EUCLIDES (Version 4.0) and its included libraries.

Euclides Language



1. Data Structures & Types

The first example in almost every programming book since Brian Kernighan's and Dennis Ritchie's "The C Programming Language" [KR78] is a version of the famous "hello world"-program. Following this tradition, the first Euclides program implements "Hello World!":

Source Code 1.1 — Hello World.

```
1  import 'io'
2
3  print("Hello World!");
```

As expected, the "hello world"-program listed in source code 1.1 prints the corresponding message to the standard output stream as shown in output 1.2.

Output 1.2 — Hello World.

```
Hello World!
```

This example shows how source codes and their execution results are set in this book.

1.1 Variables and Constants

Variables and constants are called identifiers. Each identifier has a name which starts with a letter or an underscore followed by an arbitrary number of letters, digits, or underscores:

```
IDENTIFIER
        : ('_' | 'a'..'z' | 'A'..'Z') ('_' | 'a'..'z' | 'A'..'Z' | '0'..'9')*
        ;
```

Variables and constants are declared using the keywords **var** and **const**, respectively. Please note that Euclides is not case sensitive; i.e. keywords and identifiers can be written with arbitrary upper and lower case. As a consequence the Euclides programs listed in source code 1.3 and 1.4 are equivalent and return the same output 1.5.

Source Code 1.3 — Case Insensitive (A).

```
1  import 'io'
2
3  var xxx = 42;
4  print(xxx);                    // output: >> 42 <<
5  print(xxx);                    // output: >> 42 <<
```

Source Code 1.4 — Case Insensitive (B).

```
1  ImPoRt 'io'
2
3  VAR xxx = 42;
4  print(xxx);                    // output: >> 42 <<
5  print(XxX);                    // output: >> 42 <<
```

Output 1.5 — Case Insensitive (A & B).

```
42
42
```

Each variable and constant has a dynamic type depending on its value. Within Euclides the following types are supported:

Undefined The undefined value (the default value of each variable at declaration) is represented by the singleton value and keyword **undefined**. This value corresponds to `null`, `nil`, etc. in other languages.

Boolean The boolean values are represented by the keywords **true** and **false**, which have the type **boolean**.

Integer Integer values are stored in variables and constants of type integer. In Euclides, integers have arbitrary precision.

Float Floating point numbers are stored using a type according to the IEEE 754 standard for double precision floating point numbers.

String Strings are represented by a type with the same name. A string literal is composed of characters and escape sequences.

```
STRING_LITERAL
        : '"' (ESCAPE_SEQUENCE | CHARACTER)* '"'
        ;
```

with

```
ESCAPE_SEQUENCE
        : '\' ('n' | 'r' | 't' | '"' | ''' | '\' | ('u' HEX HEX HEX HEX))
        ;
```

and

```
CHARACTER
        : 'a'..'z' | 'A'..'Z' | '0'..'9' | '_' | ' '
        | '(' | ')' | '[' | ']' | '{' | '}' | '<' | '>' | '|'
        | '=' | '+' | '-' | '*' | '/' | '.' | ',' | ';' | ':'
        | '!' | '?' | '#' | '^' | '@' | '%' | '&' | '$' | '§'
        ;
```

and

```
HEX
        : '0'..'9' | 'a'..'f' | 'A'..'F'
        ;
```

These data types (undefined, boolean, integer, float, and string) are called *primitive* in contrast to the following *composed* data types. The primitive data types as well as the vector and the matrix data types are value-based, whereas the array, object, and function data types are reference-based.

Array An array is a collection of arbitrary data type. Each array has a dynamic size (expanding as needed). All elements of an array must have the same type or must be **undefined**.

Vector A vector is similar to a one-dimensional array with floating point numbers stored in it. The differences between arrays and vectors are a different set of supported operators.

Matrix A matrix is similar to a two-dimensional array with floating point numbers stored in it. The differences between arrays, vectors and matrices are a different set of supported operators.

Function Each function in EUCLIDES is represented by a function data type.

Object An EUCLIDES object is a map with case-insensitive, string-based keys and arbitrary values.

In contrast to an identifier declared to be a variable, a declared constant cannot change its assigned value. In detail, a constant has to be defined with an assignment using an identifier on the left hand side and an expression on the right hand side. Once a value or reference has been assigned, a constant cannot be changed. All attempts to change a constant will result in compile-time errors (in cases of direct access) or run-time errors (in cases of indirect access). The initialization of a constant is different to the initialization of variables. The constants evaluate the initially assigned expression and copy the result deeply; i.e. a passed reference (e.g. pointing to an array) will be copied including all sub-references and dependencies.

The differences between constants and variables are illustrated in the following two source code examples (1.6 and 1.7).

Source Code 1.6 — Constants (A).

```
import 'io'

//
// Constants are declared with a value or a reference:
//
const pi = 3.1415926;

//
// It is not possible to change the value afterwards; i.e. the value
// of a constant can only be set during its definition. Any attempt to
// change the assigned value will result in an error; but, the copy of
// a constant is not constant ($ is the copy operator).
//
var almostPi = $pi;
almostPi = 3.0;

//
// Constant objects cannot be modified as well. Therefore, the
// following implementation of a counter cannot be used directly.
//
const counter = {
    "value"     : 42,
    "reset"     : function() { this@"value" = 0;                },
    "increment" : function() { this@"value" = this@"value" + 1; }
};

//
// Although the statement
//
//   counter@"reset"();
//
// will lead to a runtime error (due to the attempt to modify the
// object member "value"), the object can be copied and used as a
// protoype.
//
var cInstance = $counter;
cInstance@"increment"();
cInstance@"increment"();
cInstance@"increment"();

//
// As a result the following statement will print "45" to the
// standard output stream.
//
print(cInstance@"value");     // output: >> 45 <<
```

The second example (source code 1.7) illustrates the initialization of constants. The assigned value or reference is evaluated and the evaluation result is copied deeply; i.e. a deep copy duplicates everything. A deep copy of an array or an object is another array, respectively object, with all of the elements in the original collection duplicated recursively. Especially in combination with reference-based data types, the implicit deep copy makes a big difference.

Source Code 1.7 — Constants (B).

```
1  import 'io'
2
3  var array1 = [1, 2, 3];
4  var array2 = [0, 1, 42];
5  var varArray = [array1, array2];
6  //
7  print(array1);                // output: >> [1, 2, 3] <<
8  print(array2);                // output: >> [0, 1, 42] <<
9  print(varArray);              // output: >> [[1, 2, 3], [0, 1, 42]] <<
10
11 varArray @ 0 @ 2 = 42;
12 //
13 print(array1);                // output: >> [1, 2, 42] <<
14 print(array2);                // output: >> [0, 1, 42] <<
15 print(varArray);              // output: >> [[1, 2, 42], [0, 1, 42]] <<
16
17 const constArray = [array1, array2];
18 //
19 print(array1);                // output: >> [1, 2, 42] <<
20 print(array2);                // output: >> [0, 1, 42] <<
21 print(varArray);              // output: >> [[1, 2, 42], [0, 1, 42]] <<
22 print(constArray);            // output: >> [[1, 2, 42], [0, 1, 42]] <<
23
24 //
25 // The statement
26 //
27 //   constArray @ 0 @ 2 = 43;
28 //
29 // is an error and changing the reference used at declaration
30 // has no impact, because each newly generated constant copies
31 // it values and references deeply:
32 //
33 array1 @ 2 = -1;
34 //
35 print(array1);                // output: >> [1, 2, -1] <<
36 print(array2);                // output: >> [0, 1, 42] <<
37 print(varArray);              // output: >> [[1, 2, -1], [0, 1, 42]] <<
38 print(constArray);            // output: >> [[1, 2, 42], [0, 1, 42]] <<
```

The only way to modify a constant (at least partly) is to copy it deeply (using the copy operator) and to assign the copy to a variable. Then, the resulting copy is not constant.

1.2 Scoping and Visibility

Having parsed an EUCLIDES script all variables and constants are internally represented by unique symbols in a symbol table. The table has a strict hierarchical order that defines the visibility of a symbol. The following example code demonstrates the visibility of symbols. The parsed code has the symbol table shown in Table 1.1. Within a scope only the already (previously) defined symbols of the same scope or of a higher scope are visible. Block statements and statements of control flow introduce new, nested scopes. As a consequence, the variables `x` defined in lines #3 and #11 (see the listing 1.8) have the nested scopes `[ROOT:1:GLOBAL]` and `[ROOT:1:GLOBAL].[BLCK:10015:...]` (see Table 1.1).

Source Code 1.8 — Scope.

```
1  import 'io'
2
3  var x = 42;
4  print("start");                // output: >> start <<
5  print(x);                      // output: >> 42 <<
6
7  //
8  // A new statement block { } defines a new nested scope.
9  //
10 {
11     var x = 43;                // compiler warning!
12     print(x);                  // output: >> 43 <<
13 }
14
15 var f = function() {
16     print(x);
17     var x = 44;                // compiler warning!
18     print(x);
19 };
20 print("f:");                   // output: >> f: <<
21 f();                           // output: >> 42 <<
22                                // output: >> 44 <<
23
24 var g = function() {
25     var x = 45;                // compiler warning!
26     var h = function() {
27         print("h:");
28         print(x);
29     };
30     print(x);
31     h();
32 };
33
34 print("g:");                   // output: >> g: <<
35 g();                           // output: >> 45 <<
36                                // output: >> h: <<
37                                // output: >> 42 <<
38 print("end");                  // output: >> end <<
```

A function definition creates a new scope, which is always a child of the root scope `[ROOT:1:GLOBAL]`. Consequently, within a function only those variables and constants are visible, which are

- globally defined (as their scope (the root scope) is higher than the function's scope),
- function parameters (as they are in the scope of the function's body), or
- accessible via the **`this`** reference.

Table 1.1 shows an excerpt of the internal symbol table generated by the EUCLIDES compiler during parsing.

If a library, i.e. an EUCLIDES source imported by the **`import`** command, defines a symbol, and if the symbol's name starts with an underscore, then the symbol is anonymized after the parsing of the library. In this way, libraries can define global variables and constants, which are not visible from "outside". This technique is used to avoid global namespace pollution.

ID[1]	scope and name	definition (line number)	usages (line numbers)
10021	`[ROOT:1:GLOBAL].` `x`	#3	#3, #5, #16 (→ #15), #28 (→ 26, → 24)

10024	`[ROOT:1:GLOBAL].`
	`[BLCK:10015:...].`
	`x`
		#11	#11, #12

10028	`[ROOT:1:GLOBAL].`
	`[FUDE:10016:...].`
	`[BLCK:10017:...].`
	`x`
		#17 (→ 15)	#17 (→ 15), #18 (→ 15)

10034	`[ROOT:1:GLOBAL].`
	`[FUDE:10018:...].`
	`[BLCK:10019:...].`
	`x`
		#25 (→ 24)	#25 (→ 24), #30 (→ 24)

Table 1.1: The symbol table represents all variables and constants (and intermediate, temporary variables of subexpressions). Each symbol has a unique ID and lists all its usages (line number → enclosing statement). This excerpt only lists the variables named `x` in source code 1.8.

[1] Please note that a symbol's ID and the name of the scope should not be considered stable and may be subject to future changes.

1.3 Operators

Within expressions the EUCLIDES language supports a predefined set of operators. The complete list of operators is shown in the Tables 1.2 and 1.3.

code	operation	comment
`=`	assignment	defined for all types
`@`	member access	defined for strings, vectors, matrices, arrays, objects (first operand) with corresponding indices (second operand)
`::`	object-function binding	defined for objects (first operand) and strings (second operand) with function as a member access result
`$`	deep copy	defined for all types
`||`	logical OR	defined for booleans only
`&&`	logical AND	defined for booleans only
`!`	logical, unary NOT	defined for booleans only
`==`	relational equality	checks type and value/reference
`!=`	relational inequality	checks type and value/reference
`<`	relational LESS-THAN	defined for floats, integers, strings
`>`	relational GREATER-THAN	defined for floats, integers, strings
`<=`	relational LESS-OR-EQUAL	the code `x <= y` is mapped to the code `(x < y)||(x == y)`
`>=`	relational GREATER-OR-EQUAL	the code `x >= y` is mapped to the code `(x > y)||(x == y)`
`in`	relational IN	`X in Y` is defined for `X` being an arbitrary type and for `Y` being a string, an array, or an object
`typeof`	relational TYPE-OF	`X typeof Y` is defined for `X` being an arbitrary type and for `Y` being a string

Table 1.2: The operators of the EUCLIDES language (part 1).

code	operation	comment
`++`	concatation	defined for arrays, strings, and vectors
`+`	arithmetic, binary addition	defined for all arithmetic . . .
`-`	arithmetic, binary subtraction	. . . types (i.e. integers, floats, . . .
`*`	standard multiplication	. . . vectors, matrices)
`#`	element-wise multiplication	– " –
`/`	arithmetic division	– " –
`%`	arithmetic modulo	– " –
`^`	arithmetic power	– " –
`+`	arithmetic, unary plus	defined for all arithmetic types
`-`	arithmetic, unary minus	defined for all arithmetic types
`~`	arithmetic transpose	defined for vectors, matrices
`"`	String conversion (the operator consists of two single quotes)	defined for all types
`{}`	JSON conversion	defined for all types
`<>`	XML conversion	defined for all types
`boolean`	cast to boolean	defined for stings
`integer`	cast to integer	defined for stings
`float`	cast to float	defined for stings
`eval`	evaluation	expects a valid JSON-formated string

Table 1.3: The operators of the EUCLIDES language (part 2).

1.4 Data Types

EUCLIDES comprehends the following built-in data types:

Undefined The undefined value is represented by the singleton value and keyword `undefined`.

Boolean The boolean values are represented by the keywords `true` and `false`.

Integer Integer values in EUCLIDES may have arbitrary precision.

Float Floating point numbers are stored using a type according to the IEEE 754 standard for double precision floating point numbers.

String Strings are represented by a type with the same name.

These data types (undefined, boolean, integer, float, and string) are called *primitive* in contrast to the following *composed* data types. The primitive data types as well as the vector and the matrix data types are value-based, whereas the array, object, and function data types are reference-based.

Array An array is a collection of arbitrary data type.

Vector A vector is similar to a one-dimensional array with floating point numbers stored in it.

Matrix A matrix is similar to a two-dimensional array with floating point numbers stored in it.

Function Each function in EUCLIDES is represented by a function data type.

Object An EUCLIDES object is a map with case-insensitive, string-based keys and arbitrary values.

1.5 Type: Undefined

The data type `undefined` is an immutable singleton. For undefined data only a small subset of operators and operations is defined (see Listing 1.9).

Source Code 1.9 — Data Type UNDEFINED.

```
1  import 'io'
2
3  //
4  // Variables are undefined by default ...
5  // ..., but they can also be set explicitly to undefined.
6  //
7  var x;
8  var y = undefined;
9
10 print(x == y);                // output: >> TRUE <<
11 print(y typeof "UNDEFINED");  // output: >> TRUE <<
12
13 //
14 // The toString-, toJSON-, and toXML-operators are
15 // defined for all Euclides data types:
16 //
17 print();
18 print("toString conversion operator:");
19 print(''x);
20
21 print();
22 print("toJSON conversion operator: ");
23 print({}x);
24
```

```
25  print();
26  print("toXML conversion operator: ");
27  print(<>x);
```

The result of Listing 1.9 is shown in Output 1.10.

Output 1.10 — UNDEFINED data.

```
TRUE
TRUE

toString conversion operator:
UNDEFINED

toJSON conversion operator:
null

toXML conversion operator:
<?xml version="1.0" encoding="UTF-8"?>
<undefined/>
```

1.6 Type: Boolean

The following example illustrates the boolean data type.

Source Code 1.11 — Data Type BOOLEAN.

```
import 'io'

//
// Boolean data type:
//
var x = true;
var y = false;

print(x == y);                  // output: >> FALSE <<
print(y typeof "BOOLEAN");      // output: >> TRUE <<

//
// The toString-, toJSON-, and toXML-operators are
// defined for all Euclides data types:
//
print();
print("toString conversion operator:");
print(''x);

print();
print("toJSON conversion operator: ");
print({}x);

print();
print("toXML conversion operator: ");
print(<>x);
```

The result of Listing 1.11 is shown in Output 1.12.

Output 1.12 — BOOLEAN data.

```
FALSE
TRUE

toString conversion operator:
TRUE

toJSON conversion operator:
true

toXML conversion operator:
<?xml version="1.0" encoding="UTF-8"?>
<boolean>TRUE<boolean/>
```

Its usage and the corresponding operators are shown in the next example:

Source Code 1.13 — Data Type BOOLEAN.

```
1  import 'io'
2
3  //
4  // Boolean operators:
5  //
6  var x = true;
7  var y = false;
8
9  //
10 // Logical operations:
11 //
12 print(x || y);                 // output: >> TRUE <<
13 print(x && y);                 // output: >> FALSE <<
14 print(!x);                     // output: >> FALSE <<
15
16 //
17 // Casting:
18 //
19 var ecs = "true";
20 print(ecs typeof "STRING");    // output: >> TRUE <<
21
22 var t = boolean(ecs);
23 print(t typeof "BOOLEAN");     // output: >> TRUE <<
24 print(t);                      // output: >> TRUE <<
25
26 //
27 // Evaluation:
28 //
29 var json = "false";
30 print(json typeof "STRING");   // output: >> TRUE <<
31
32 var f = eval(json);
33 print(f typeof "BOOLEAN");     // output: >> TRUE <<
34 print(f);                      // output: >> FALSE <<
```

1.7 Type: Integer

Euclides has two built-in data types to represent numbers: an integer data type for arbitrary precision integer values and a floating point data type according to the IEEE 754 standard. The integer data type is illustrated in the source codes 1.14 and 1.16.

Source Code 1.14 — Data Type INTEGER.

```
1  import 'io'
2
3  //
4  // Integer data type:
5  //
6  var x = -123;
7  var y = 456;
8
9  print(x);                    // output: >> -123 <<
10 print(x == y);               // output: >> FALSE <<
11 print(x == -123);            // output: >> TRUE <<
12 print(y typeof "INTEGER");   // output: >> TRUE <<
13
14 //
15 // The toString-, toJSON-, and toXML-operators are
16 // defined for all Euclides data types:
17 //
18 print();
19 print("toString conversion operator:");
20 print(''x);
21
22 print();
23 print("toJSON conversion operator: ");
24 print({}x);
25
26 print();
27 print("toXML conversion operator: ");
28 print(<>x);
```

The result of the conversion operators (and the previous output of source codes 1.14 is:

Output 1.15 — INTEGER data.

```
-123
FALSE
TRUE
TRUE

toString conversion operator:
-123

toJSON conversion operator:
-123

toXML conversion operator:
<?xml version="1.0" encoding="UTF-8"?>
<integer>-123</integer>
```

The following source code sketches the power of arbitrary precision:

Source Code 1.16 — Data Type INTEGER.

```
1  import 'io'
2
3  //
4  // Operators on integers:
5  //
6  var x = 12345;
7  var y = 67890;
8
9  //
10 // Relational operations:
11 //
12 print(x == y);                 // output: >> FALSE <<
13 print(x != y);                 // output: >> TRUE <<
14 print(x <  y);                 // output: >> TRUE <<
15 print(x <= y);                 // output: >> TRUE <<
16 print(x >  y);                 // output: >> FALSE <<
17 print(x >= y);                 // output: >> FALSE <<
18
19 //
20 // Arithmetic operations:
21 //
22 print(+x);                     // output: >> 12345 <<
23 print(-x);                     // output: >> -12345 <<
24 print(x + y);                  // output: >> 80235 <<
25 print(x - y);                  // output: >> -55545 <<
26 print(x * y);                  // output: >> 838102050 <<
27 print(x / y);                  // output: >> 0 <<
28 print(x % y);                  // output: >> 12345 <<
29 print(x ^ y);                  // output: >> 213988[...]640625 <<
30                                // Numbers are not abbreviated!
31 print(x^3 + 3*x^2 - 4*x + 8); // output: >> 1881823111328 <<
32
33 //
34 // Casting:
35 //
36 var ecs = "12345678901234567890123456789O";
37 print(ecs typeof "STRING");    // output: >> TRUE <<
38
39 var i = integer(ecs);
40 print(i typeof "INTEGER");     // output: >> TRUE <<
41 print(i);                      // output: >> 1234567890123456789 ↩
42                                // 01234567890 <<
43
44 var f = boolean(0);            // zero is casted to "false"
45 print(f typeof "BOOLEAN");     // output: >> TRUE <<
46 print(f);                      // output: >> FALSE <<
47
48 var t = boolean(2);            // non-zero is casted to "true"
49 print(t typeof "BOOLEAN");     // output: >> TRUE <<
50 print(t);                      // output: >> TRUE <<
51
```

```
52  //
53  // Evaluation:
54  //
55  var json = "987654321";
56  print(json typeof "STRING");  // output: >> TRUE <<
57
58  var value = eval(json);
59  print(value);                 // output: >> 987654321 <<
```

1.8 Type: Float

The floating point numbers are based on the IEEE 754 standard. In contrast to many other programming and scripting languages, EUCLIDES does not convert between integers and floats automatically. Each conversion has to be stated explicitly as shown in the following example (see lines 11 and 12):

Source Code 1.17 — Data Type FLOAT.

```
1  import 'io'
2
3  //
4  // Floating point number data type:
5  //
6  var x = -123.0;
7  var y = 456.345;
8
9  print(x);                      // output: >> -123.0 <<
10 print(x == y);                 // output: >> FALSE <<
11 print(x == -123);              // output: >> FALSE <<
12 print(x == -123.0);            // output: >> TRUE <<
13 print(y typeof "FLOAT");       // output: >> TRUE <<
14
15 //
16 // The toString-, toJSON-, and toXML-operators are
17 // defined for all Euclides data types:
18 //
19 print();
20 print("toString conversion operator:");
21 print(''x);
22
23 print();
24 print("toJSON conversion operator: ");
25 print({}x);
26
27 print();
28 print("toXML conversion operator: ");
29 print(<>x);
```

The conversion results are:

Output 1.18 — FLOAT data.

```
-123.0
FALSE
FALSE
TRUE
TRUE

toString conversion operator:
-123.0

toJSON conversion operator:
-123.0

toXML conversion operator:
<?xml version="1.0" encoding="UTF-8"?>
<float>-123.0</float>
```

The defined operators do not perform any implicit conversion as well.

Source Code 1.19 — Data Type FLOAT.

```
1  import 'io'
2
3  //
4  // Operators on floats:
5  //
6  var x = 12345.0;
7  var y = 67890.0;
8
9  //
10 // Relational operations:
11 //
12 print(x == y);                 // output: >> FALSE <<
13 print(x != y);                 // output: >> TRUE <<
14 print(x <  y);                 // output: >> TRUE <<
15 print(x <= y);                 // output: >> TRUE <<
16 print(x >  y);                 // output: >> FALSE <<
17 print(x >= y);                 // output: >> FALSE <<
18
19 //
20 // Arithmetic operations:
21 //
22 print(+x);                     // output: >> 12345.0 <<
23 print(-x);                     // output: >> -12345.0 <<
24 print(x + y);                  // output: >> 80235.0 <<
25 print(x - y);                  // output: >> -55545.0 <<
26 print(x * y);                  // output: >> 8.3810205E8 <<
27 print(x / y);                  // output: >> 0.1818382677861246 <<
28 print(x % y);                  // output: >> 12345.0 <<
29 print(x ^ y);                  // output: >> Infinity <<
30 print(x^3 + 3.0*x^2 - 4.0*x); // output: >> 1.881823111328E12 <<
31
32 //
33 // Casting:
34 //
35 var ecs = "-123456.7890";
36 print(ecs typeof "STRING");    // output: >> TRUE <<
37
38 var flt = float(ecs);
39 print(flt typeof "FLOAT");     // output: >> TRUE <<
40 print(flt);                    // output: >> -123456.789 <<
41
42 var f = boolean(0.0);          // zero is casted to "false"
43 print(f typeof "BOOLEAN");     // output: >> TRUE <<
44 print(f);                      // output: >> FALSE <<
45
46 var t = boolean(3.14);         // non-zero is casted to "true"
47 print(t typeof "BOOLEAN");     // output: >> TRUE <<
48 print(t);                      // output: >> TRUE <<
49
```

```
50  //
51  // Evaluation:
52  //
53  var json = "-987654321.1234";
54  print(json typeof "STRING");  // output: >> TRUE <<
55
56  var value = eval(json);
57  print(value);                 // output: >> -9.876543211234E8 <<
```

In the evaluation of the polynomial (line 30 in source code 1.19), the coefficients of the polynomial have to be floating point numbers; otherwise, an exception will be thrown, as EUCLIDES does not perform any implicit conversion.

1.9 Type: String

Euclides has a built-in data type for string handling with corresponding operators for comparison, concatation, access, modification, etc.

Source Code 1.20 — Data Type STRING.

```
1  import 'io'
2
3  //
4  // String data type:
5  //
6  var x = "Hello World!";
7  var y = "Hello World!";
8
9  print(x);                      // output: >> Hello World! <<
10 print(x == y);                 // output: >> TRUE <<
11 print(y typeof "STRING");      // output: >> TRUE <<
12
13 //
14 // The toString-, toJSON-, and toXML-operators are
15 // defined for all Euclides data types:
16 //
17 print();
18 print("toString conversion operator:");
19 print(''x);
20
21 print();
22 print("toJSON conversion operator: ");
23 print({}x);
24
25 print();
26 print("toXML conversion operator: ");
27 print(<>x);
```

The source code 1.20 produces the output:

Output 1.21 — STRING data.

```
Hello World!
TRUE
TRUE

toString conversion operator:
Hello World!

toJSON conversion operator:
"\u0048\u0065\u006C\u006C\u006F\u0020\u0057\u006F\u0072\u006C\u0064\u0021"

toXML conversion operator:
<?xml version="1.0" encoding="UTF-8"?>
<string>Hello World!</string>
```

The supported operators for string-based data include relational operators based on alphabetic order (<, <=, >, >=), concatation (++), and access.

Source Code 1.22 — Data Type STRING.

```
1  import 'io'
2
3  //
4  // Operators on strings:
5  //
6  var x = "Hello";
7  var y = "World";
8
9  //
10 // Relational operations:
11 //
12 print(x == y);                // output: >> FALSE <<
13 print(x != y);                // output: >> TRUE <<
14 print(x <  y);                // output: >> TRUE <<
15 print(x <= y);                // output: >> TRUE <<
16 print(x >  y);                // output: >> FALSE <<
17 print(x >= y);                // output: >> FALSE <<
18
19 //
20 // Alphabetic order according to unicode normalization form C (NFC):
21 // canonical decomposition, followed by canonical composition
22 //
23 var rene0 = "Ren\u00e9";
24 // using Unicode Character 'LATIN SMALL LETTER E WITH ACUTE' (U+00E9)
25
26 var rene1 = "Ren\u0065\u0301";
27 // using Unicode Character 'LATIN SMALL LETTER E' (U+0065)
28 // with  Unicode Character 'COMBINING ACUTE ACCENT' (U+0301)
29
30 print(rene0);                  // output: >> René <<
31 print(rene1);                  // output: >> René <<
32 print(rene0 == rene1);         // output: >> TRUE <<
33
34 //
35 // Access and modification
36 //
37 print(rene0@"size"());         // output: >> 4 <<
38 print(rene1@"size"());         // output: >> 4 <<
39
40 var sizeFunction = x@"size";  // access the "size" member function
41 print(sizeFunction());         // output: >> 5 <<
42 print(x@1);                    // output: >> e <<
43
44 rene0@3 = "t";                 // rene0 = "Rent"
45 rene1@3 = "derings";           // rene1 = "Renderings"
46
```

```
47  //
48  // Concatation and IN-operation
49  //
50  var text = rene0 ++ " some " ++ rene1;
51  print(text);                    // output: >> Rent some Renderings <<
52  print("some" in text);          // output: >> TRUE <<
53  print("same" in text);          // output: >> FALSE <<
54
55  //
56  // The member function "index" can be used to find an element x.
57  // The returned index is either -1 or the lowest index of
58  // an element y with x == y.
59  //
60  print(text@"size"());           // output: >> 20 <<
61  print(text@"index"("R"));       // output: >> 0 <<
62  print(text@"index"("s"));       // output: >> 5 <<
63  print(text@"index"("om"));      // output: >> 6 <<
64  print(text@"index"("os"));      // output: >> -1 <<
```

1.10 Type: Array

The data types **undefined**, **boolean**, **integer**, **float**, and **string** are value-based (as well as the data types **vector** and **matrix** described later). The data type `array` is reference-based (as well as the data type `object` described later). An array always has a dynamic size (expanding as needed) as illustrated in the next source code (except the array declared to be constant):

Source Code 1.23 — Data Type ARRAY.

```
1  import 'io'
2
3  //
4  // Array data type:
5  //
6  var array1 = [ 1, 2, 3, 4 ];
7  var array2 = [ 1, 2, 3, 4 ];
8  var array3 = [ 5, 6, 7, 8, 9 ];
9  var array4 = [ ];
10
11 print(array1);                 // output: >> [1, 2, 3, 4] <<
12 print(array1 == array2);       // output: >> FALSE <<
13 print(array1 == array3);       // output: >> FALSE <<
14 print(array1 typeof "ARRAY"); // output: >> TRUE <<
15 print(array4);                 // output: >> [ ] <<
16
17 //
18 // The toString-, toJSON-, and toXML-operators are
19 // defined for all Euclides data types:
20 //
21 print();
22 print("toString conversion operator:");
23 print(''array1);
24
25 print();
26 print("toJSON conversion operator: ");
27 print({}array1);
28
29 print();
30 print("toXML conversion operator: ");
31 print(<>array1);
```

The resulting output is:

Output 1.24 — ARRAY data.

```
[1, 2, 3, 4]
FALSE
FALSE
TRUE
[UNDEFINED, UNDEFINED, UNDEFINED, UNDEFINED, UNDEFINED, UNDEFINED]

toString conversion operator:
[1, 2, 3, 4]

toJSON conversion operator:
[
  1,
  2,
  3,
  4
]

toXML conversion operator:
<?xml version="1.0" encoding="UTF-8"?>
<array id="id0">
 <integer>1</integer>
 <integer>2</integer>
 <integer>3</integer>
 <integer>4</integer>
</array>
```

Please note that the XML output may vary; in detail, the array-ID `id0` is generated at run-time to resolve cyclic references. The id-mechanism ensures that the same reference always gets the same id, but different execution environments may result in different ids at different run-times. As a naive conversion to strings, JSON, XML of an array or an object may result in endless loops due to cyclic references, each conversion operator has a strategy to avoid this problem (the problem is implemented in source code 1.25).

Source Code 1.25 — Data Type ARRAY.

```
1  import 'io'
2
3  //
4  // The array data type is reference-based ...
5  //
6  var array = [ ];
7  array@0 = [0, 0, 0];
8  array@1 = [1, 1, 1];
9  array@2 = array@0;
10
11 //
12 // ... and references may be cyclic:
13 //
14 array@3 = array;
15
16 print(''array);
17 print(<>array);
18 //print({}array);
```

The different strategies can be seen in the output (see 1.26) of source code 1.25:

Output 1.26 — ARRAY data.

```
[[0, 0, 0], [1, 1, 1], [ ... ], [ ... ]]
<?xml version="1.0" encoding="UTF-8"?>
<array id="id0">
 <array id="id1">
  <integer>0</integer>
  <integer>0</integer>
  <integer>0</integer>
 </array>
 <array id="id2">
  <integer>1</integer>
  <integer>1</integer>
  <integer>1</integer>
 </array>
 <array idref="id1"/>
 <array idref="id0        "/>
</array>
```

The string conversion recognizes arrays and objects which have been converted already and inserts the unicode character "horizontal ellipsis" instead. The XML conversion uses the id-idref-mechanism to map cyclic references to an XML structure. As the JSON format is not able to represent cyclic references, the JSON operator throws an exception, if applied to a data structure that contains the same reference more than once.

An array has a dynamic size. The access to arrays via the `@` operator checks the array size and the elements' data type: The array expands, in case of an "out of range" access. The first element that is not **`undefined`** defines the array element type, which is used for type checking by all following elements. Please note that the type checking does not include the elements of arrays and objects; i.e. an array of arrays of integers has the same type as an array of integers.

Source Code 1.27 — Data Type ARRAY.

```
1  import 'io'
2
3  //
4  // Operators on arrays:
5  //
6  var array1 = [ 1, 2, 3, 4 ];
7  var array2 = [ 5, 6, 7, 8 ];
8  var array3 = [ ];
9
10 //
11 // Relational operations:
12 //
13 print(array1 == array2);       // output: >> FALSE <<
14 print(array1 != array2);       // output: >> TRUE <<
15 print(array1 typeof "ARRAY"); // output: >> TRUE <<
16 print(4 in array1);           // output: >> TRUE <<
17 print(4 in array2);           // output: >> FALSE <<
18
```

```
19  //
20  // Access and modification (using built-in functions)
21  //
22  print(array1 ++ array2);      // output: >> [1, 2, 3, 4, 5, 6, 7, 8] <<
23  print(array1@"size"());       // output: >> 4 <<
24  print(array1@1);              // output: >> 2 <<
25
26  var array4 = array1 ++ array2;
27  print(array4);                // output: >> [1, 2, 3, 4, 5, 6, 7, 8] <<
28
29  var first  = array4@"first";// remove first element and return it
30  print(first);                 // output: >> 1 <<
31  var last   = array4@"last"(); // remove last element and return it
32  print(last);                  // output: >> 8 <<
33  print(array4);                // output: >> [2, 3, 4, 5, 6, 7] <<
34
35  array4@"front"(9);            // insert element at front;
36                                // i.e. at position 0
37  array4@"back"(99);            // insert element at back;
38                                // i.e. at position "size"()-1
39  print(array4);                // output: >> [9, 2, 3, 4, 5, 6, 7, 99] <<
40
41  array4@"insert"(3, 17);       // insert element (17) at
42                                // arbitrary position (3)
43  array4@"remove"(1);           // remove element at
44                                // arbitrary position (1)
45  print(array4);                // output: >> [9, 3, 17, 4, 5, 6, 7, 99] <<
46
47  //
48  array3@0 = "a";
49  array3@1 = "b";
50  array3@2 = "c";
51  print(array3);                // output: >> [a, b, c] <<
52
53  //
54  // The element that has been inserted first defines the type of
55  // all array elements; therefore, the statement: array3@0 = 42;
56  // would throw an exception as all elements must have the same
57  // type and array3 is filled with strings.
58  //
59
60  //
61  // The member function "index" can be used to find an element x.
62  // The returned index is either -1 or the lowest index of
63  // an element y with x == y.
64  //
65  var array5 = [ "a", "b", "c", "d", "e", "f", "g", "h"];
66  print(array5@"size"());       // output: >> 8 <<
67  print(array5@"index"("a"));   // output: >> 0 <<
68  print(array5@"index"("f"));   // output: >> 5 <<
69  print(array5@"index"("z"));   // output: >> -1 <<
```

1.11 Type: Vector and Matrix

Vectors and matrices are similar to a one-dimensional, resp. two-dimensional, arrays with floating point numbers stored in it. As vectors and matrices are used very often in geometric modeling, EUCLIDES supports them natively. In contrast to arrays, the vector and matrix data types are not a reference-based data type, but value-based ones.

Source Code 1.28 — Data Type VECTOR & MATRIX.

```
1  import 'io'
2
3  //
4  // Matrix data type:
5  //
6  var matrix1 = < <  1.0,  2.0,  3.0,  4.0 >,
7                  <  6.0,  7.0,  8.0,  9.0 >,
8                  < 11.0, 12.0, 13.0, 14.0 > >;
9
10 print(matrix1);               // output: >> ↩
11                               //  <<1.0, 2.0, 3.0, 4.0>, ↩
12                               //   <6.0, 7.0, 8.0, 9.0>, ↩
13                               //   <11.0, 12.0, 13.0, 14.0>>> <<
14
15 var axis1 = < 1.0, 1.0, 1.0, 0.0>;
16 var axis2 = < 0.0, 2.0, 2.0, 0.0>;
17 var axis3 = < 0.0, 0.0, 3.0, 0.0>;
18 print(axis1);                  // output: >> < 1.0, 1.0, 1.0, 0.0> <<
19 print(axis2);                  // output: >> < 0.0, 2.0, 2.0, 0.0> <<
20 print(axis3);                  // output: >> < 0.0, 0.0, 3.0, 0.0> <<
21
22 //
23 // Short notation for matrices using some vectors to define it
24 //
25 var matrix2 = < axis1, axis2, axis3 >;
26 print(matrix2);               // output: >> ↩
27                               //  <<1.0, 1.0, 1.0, 0.0>, ↩
28                               //   <0.0, 2.0, 2.0, 0.0>, ↩
29                               //   <0.0, 0.0, 3.0, 0.0>> <<
30
31 var test = <<1.0,1.0,1.0,0.0>, <0.0,2.0,2.0,0.0>, <0.0,0.0,3.0,0.0>>;
32 print(matrix1 == matrix2);    // output: >> FALSE <<
33 print(test    == matrix2);    // output: >> TRUE <<
34 print(test  typeof "MATRIX"); // output: >> TRUE <<
35 print(axis1 typeof "VECTOR"); // output: >> TRUE <<
36 //
37 // Short notation for vectors / matrices using dimensions to define them;
38 // all elements are set to zero:
39 //
40 var matrix4 = <| 3, 4 |>;
41 print(matrix4);               // output: >> ↩
42                               //  <<0.0, 0.0, 0.0, 0.0>, ↩
43                               //   <0.0, 0.0, 0.0, 0.0>, ↩
44                               //   <0.0, 0.0, 0.0, 0.0>> <<
45
46 var vector1  = <| 2 |>;
47 print(vector1);               // output: >> <0.0, 0.0> <<
48
```

```
49  //
50  // The toString-, toJSON-, and toXML-operators are
51  // defined for all Euclides data types:
52  //
53  print("conversion of a vector:");
54  print(''axis1);
55  print({}axis1);
56  print(<>axis1);
57
58  print("conversion of a matrix:");
59  print(''matrix1);
60  print({}matrix1);
61  print(<>matrix1);
```

The output of source code 1.28 is:

Output 1.29 — VECTOR & MATRIX data.

```
<<1.0, 2.0, 3.0, 4.0>, <6.0, 7.0, 8.0, 9.0>, <11.0, 12.0, 13.0, 14.0>>
<1.0, 1.0, 1.0, 0.0>
<0.0, 2.0, 2.0, 0.0>
<0.0, 0.0, 3.0, 0.0>
<<1.0, 1.0, 1.0, 0.0>, <0.0, 2.0, 2.0, 0.0>, <0.0, 0.0, 3.0, 0.0>>
FALSE
TRUE
TRUE
TRUE
<<0.0, 0.0, 0.0, 0.0>, <0.0, 0.0, 0.0, 0.0>, <0.0, 0.0, 0.0, 0.0>>
<0.0, 0.0>
conversion of a vector:
<1.0, 1.0, 1.0, 0.0>
[
  1.0,
  1.0,
  1.0,
  0.0
]
<?xml version="1.0" encoding="UTF-8"?>
<vector>
 <coefficient>1.0</coefficient>
 <coefficient>1.0</coefficient>
 <coefficient>1.0</coefficient>
 <coefficient>0.0</coefficient>
</vector>
```

```
conversion of a matrix:
<<1.0, 2.0, 3.0, 4.0>, <6.0, 7.0, 8.0, 9.0>, <11.0, 12.0, 13.0, 14.0>>
[
  [
    1.0,
    2.0,
    3.0,
    4.0
  ],
  [
    6.0,
    7.0,
    8.0,
    9.0
  ],
  [
    11.0,
    12.0,
    13.0,
    14.0
  ]
]
<?xml version="1.0" encoding="UTF-8"?>
<matrix>
 <vector>
  <coefficient>1.0</coefficient>
  <coefficient>2.0</coefficient>
  <coefficient>3.0</coefficient>
  <coefficient>4.0</coefficient>
 </vector>
 <vector>
  <coefficient>6.0</coefficient>
  <coefficient>7.0</coefficient>
  <coefficient>8.0</coefficient>
  <coefficient>9.0</coefficient>
 </vector>
 <vector>
  <coefficient>11.0</coefficient>
  <coefficient>12.0</coefficient>
  <coefficient>13.0</coefficient>
  <coefficient>14.0</coefficient>
 </vector>
</matrix>
```

The getters and setters of vectors are shown in source code 1.30:

Source Code 1.30 — Data Type VECTOR.

```
1  import 'io'
2
3  //
4  // Access and built-in functions:
5  //
6  var axis1 = < 1.0, 1.0, 1.0, 0.0>;
7  var axis2 = < 0.0, 2.0, 2.0, 0.0>;
8  var axis3 = < 0.0, 0.0, 3.0, 0.0>;
9  print(axis1);                  // output: >> <1.0, 1.0, 1.0, 0.0> <<
10 print(axis2);                  // output: >> <0.0, 2.0, 2.0, 0.0> <<
11 print(axis3);                  // output: >> <0.0, 0.0, 3.0, 0.0> <<
12
13 //
14 // Using access operator with integer index:
15 //
16 axis1@0 = 3.14159;
17 print(axis1@0);                // output: >> 3.14159 <<
18 print(axis1);                  // output: >> <3.14159, 1.0, 1.0, 0.0> <<
19
20 //
21 // Using access operator with array index:
22 //
23 axis2@[1] = 2.71828;
24 print(axis2@[1]);              // output: >> 2.71828 <<
25 print(axis2);                  // output: >> <0.0, 2.71828, 2.0, 0.0> <<
26
27 //
28 // Using built-in get function (which expects an integer parameter),
29 // and built-in set function (which expects an integer index and
30 // a float value):
31 //
32 axis3@"set"(0, -1.0);
33 print(axis3@"get"(2));         // output: >> 3.0 <<
34 print(axis3);                  // output: >> <-1.0, 0.0, 3.0, 0.0> <<
35
36 var axes = axis1 ++ axis2 ++ axis3;
37 print(axes@"getDimension"()); // output: >> 12 <<
38 print(axes);                  // output: >> <3.14159, 1.0, 1.0, 0.0, ↩
39                               //  0.0, 2.71828, 2.0, 0.0, -1.0, 0.0, ↩
40                               //  3.0, 0.0> <<
```

The getters and setters of matrices are shown in source code 1.31:

Source Code 1.31 — Data Type MATRIX.

```
1  import 'io'
2
3  //
4  // Access and built-in functions:
5  //
6  var matrix = << 1.0, 2.0, 3.0, 0.0>,
7                < 0.0, 4.0, 5.0, 0.0>,
8                < 0.0, 0.0, 6.0, 7.0>>;
9  print(matrix);                 // output: >> ↩
10                                //   <<1.0, 2.0, 3.0, 0.0>, ↩
11                                //    <0.0, 4.0, 5.0, 0.0>, ↩
12                                //    <0.0, 0.0, 6.0, 7.0>> <<
13
14 //
15 // Using access operator with integer index:
16 //
17 print(matrix@1);               // output: >> <0.0, 4.0, 5.0, 0.0> <<
18 print(matrix@1@2);             // output: >> 5.0 <<
19
20 matrix@1 = <-1.0, -2.0, -3.0, -4.0>;
21 print(matrix);                 // output: >> ↩
22                                //   << 1.0,  2.0,  3.0,  0.0>, ↩
23                                //    <-1.0, -2.0, -3.0, -4.0>, ↩
24                                //    < 0.0,  0.0,  6.0,  7.0>> <<
25
26 matrix@1@2 = 17.0;             // This statement has no effect. The
27                                // vector returned by (matrix@1) is not a
28                                // reference, but a new value/vector,
29                                // which is afterwards modfied at (@)
30                                // position 2.
31
32 print(matrix);                 // output: >> ↩
33                                //   << 1.0,  2.0,  3.0,  0.0>, ↩
34                                //    <-1.0, -2.0, -3.0, -4.0>, ↩
35                                //    < 0.0,  0.0,  6.0,  7.0>> <<
36
37 //
38 // Using access operator with array index:
39 //
40 matrix@[1, 2] = 17.0;
41 print(matrix);                 // output: >> ↩
42                                //   << 1.0,  2.0,  3.0,  0.0>, ↩
43                                //    <-1.0, -2.0, 17.0, -4.0>, ↩
44                                //    < 0.0,  0.0,  6.0,  7.0>> <<
45
46 matrix@[0] = <1.1, 2.2, 3.3, 4.4>;
47 print(matrix);                 // output: >> ↩
48                                //   << 1.1,  2.2,  3.3,  4.4>, ↩
49                                //    <-1.0, -2.0, 17.0, -4.0>, ↩
50                                //    < 0.0,  0.0,  6.0,  7.0>> <<
51
```

```
52  //
53  // Using built-in get, getColumn, getRow, getDiagonal
54  // and built-in set, setColumn, setRow, setDiagonal:
55  //
56  print(matrix@"get"(1, 2));     // output: >> 17 <<
57  print(matrix@"getColumn"(1)); // output: >> <2.2,-2.0, 0.0> <<
58  print(matrix@"getRow"(2));     // output: >> <0.0, 0.0, 6.0, 7.0> <<
59  print(matrix@"getDiagonal"());// output: >> <1.1, -2.0, 6.0> <<
60
61  matrix@"set"(2, 1, 42.0);
62  matrix@"setColumn"(0, <0.0, 0.1, 0.2>);
63  matrix@"setRow"(0, <-0.0, -0.1, -0.2, -0.3>);
64  print(matrix);                 // output: >> ↩
65                                 //  <<-0.0, -0.1, -0.2, -0.3>, ↩
66                                 //   < 0.1, -2.0, 17.0, -4.0>, ↩
67                                 //   < 0.2, 42.0,  6.0,  7.0>> <<
68
69  matrix@"setDiagonal"(<20.0, 30.0, 40.0>);
70  print(matrix);                 // output: >> ↩
71                                 //  <<20.0, -0.1, -0.2, -0.3>, ↩
72                                 //   < 0.1, 30.0, 17.0, -4.0>, ↩
73                                 //   < 0.2, 42.0, 40.0,  7.0>> <<
74
75  //
76  // The size of a matrix is:
77  //
78  var rows = matrix@"getNumberOfRows"();
79  var cols = matrix@"getNumberOfColumns"();
80  print("size = " ++ ''rows ++ " times " ++ ''cols);
81                                 // output: >> size = 3 times 4 <<
```

EUCLIDES also defines appropriate arithmetical operators on matrices, of course. (Please note that for readability purposes the output in the comments has been modified using additional line breaks, spaces, and rounding of floating point values.)

Source Code 1.32 — Data Type VECTOR & MATRIX.

```
1  import 'io'
2
3  //
4  // Operators on vectors and matrices:
5  //
6  var vec1 = <1.0, 2.0, 3.0, 4.0>;
7  var vec2 = <5.0, 6.0, 7.0, 8.0>;
8  var mat1 = <vec1, vec2>;
9  var mat2 = <vec1, vec2, <1.1, 2.2, 3.3, 4.4>, <5.5, 6.6, 7.7, 8.8>>;
10
11 //
12 // Relational operations:
13 //
14 print(vec1 == vec2);          // output: >> FALSE <<
15 print(mat1 != mat2);          // output: >> TRUE <<
16 print(vec1 typeof "VECTOR");  // output: >> TRUE <<
17 print(vec1 typeof "MATRIX");  // output: >> FALSE <<
18
19 //
20 // Arithmetic operations:
21 //
22 print(vec1 + vec2);           // output: >> ↩
23                               //  <6.0, 8.0, 10.0, 12.0> <<
24 print(mat1 - mat1);           // output: >> ↩
25                               //  <<0.0, 0.0, 0.0, 0.0>, ↩
26                               //   <0.0, 0.0, 0.0, 0.0>> <<
27
28 //
29 // Multiplication is supported on floats, vectors and matrices.
30 // Integer values need to be converted first.
31 // For vector-vector (dot product), vector-matrix and matrix-matrix
32 // multiplications the dimensions have to match.
33 //
34 print(3.14159 * vec1);        // output: >> ↩
35                               //  <3.14159,  6.28318, ↩
36                               //   9.42477, 12.56636> <<
37
38 print(vec2 * 2.71828);        // output: >> ↩
39                               //  <13.5914,  16.30968, ↩
40                               //   19.02796, 21.74624> <<
41
42 print( 1.0 * mat2);          // output: >> ↩
43                               //  <<1.0, 2.0, 3.0, 4.0>, ↩
44                               //   <5.0, 6.0, 7.0, 8.0>, ↩
45                               //   <1.1, 2.2, 3.3, 4.4>, ↩
46                               //   <5.5, 6.6, 7.7, 8.8>> <<
47
48 print(mat2 * -1.0);          // output: >> ↩
49                               //  <<-1.0, -2.0, -3.0, -4.0>, ↩
50                               //   <-5.0, -6.0, -7.0, -8.0>, ↩
51                               //   <-1.1, -2.2, -3.3, -4.4>, ↩
52                               //   <-5.5, -6.6, -7.7, -8.8>> <<
53
```

```
54  print(vec1 * vec2);           // output: >> 70.0 <<
55
56  print(mat1 * vec1);           // output: >> <30.0, 70.0> <<
57
58  print(mat2 * vec2);           // output: >> <70.0, 174.0, 77.0, 191.4> <<
59
60  print(mat2 * mat2);           // output: >> ↩
61                                //  <<36.3,   47.0,   57.7,   68.4>, ↩
62                                //   <86.7,  114.2,  141.7,  169.2>, ↩
63                                //   <39.93,  51.7,   63.47,  75.24>, ↩
64                                //   <95.37, 125.62, 155.87, 186.12>>
65
66  //
67  // Euclides also supports elementwise multiplication
68  // with operands of same type.
69  //
70  print(vec1 # vec1);           // output: >> <1.0, 4.0, 9.0, 16.0> <<
71
72  print(mat1 # mat1);           // output: >> ↩
73                                //  <<1.0,  4.0,  9.0, 16.0>, ↩
74                                //  <25.0, 36.0, 49.0, 64.0>>
75
76  print(mat2 # mat2);           // output: >> ↩
77                                //  << 1.0,   4.0,   9.0,  16.0>, ↩
78                                //   <25.0,  36.0,  49.0,  64.0>, ↩
79                                //   < 1.21,  4.84, 10.89, 19.36>, ↩
80                                //   <30.25, 43.56, 59.29, 77.44>> <<
81
82  //
83  // The power operation is defined on matrix-integer operands.
84  //
85  print(mat2 * mat2 * mat2);    // the same output twice:
86  print(mat2 ^ 3);              // output: >> ↩
87                    //  << 710.97,   932.98,  1154.99,  1377.0>, ↩
88                    //   <1744.17,  2287.06,  2829.95,  3372.84>, ↩
89                    //   < 782.067, 1026.278, 1270.489, 1514.7>, ↩
90                    //   <1918.587, 2515.766, 3112.945, 3710.124>> <<
91
92  //
93  // Besides the "usual" unary operators (+ and -)
94  // Euclides has a built-in transpose operator.
95  //
96  print(+vec1);                 // output: >> <1.0, 2.0, 3.0, 4.0> <<
97  print(-vec2);                 // output: >> <-5.0, -6.0, -7.0, -8.0> <<
98
99  print(   vec1);               // output: >>  <1.0, 2.0, 3.0, 4.0>  <<
100 print(  ~vec1);               // output: >> <<1.0, 2.0, 3.0, 4.0>> <<
101 print( ~~vec1);               // output: >> <<1.0>,<2.0>,<3.0>,<4.0>> <<
102 print(~~~vec1);               // output: >> <<1.0, 2.0, 3.0, 4.0>> <<
103
104 print( mat1);                 // output: >> ↩
105                               // <<1.0, 2.0, 3.0, 4.0>, ↩
106                               //  <5.0, 6.0, 7.0, 8.0>>
107 print(~mat1);                 // output: >> ↩
108                               // <<1.0, 5.0>, ↩
109                               //  <2.0, 6.0>, ↩
110                               //  <3.0, 7.0>, ↩
111                               //  <4.0, 8.0>>
```

1.12 Type: Object

Objects are defined with key-value-pairs. The keys are case-insensitive strings, the values are arbitrary data types.

Source Code 1.33 — Data Type OBJECT.

```
1  import 'io'
2
3  //
4  // Object data type:
5  //
6  var object1 = { };
7  var object2 = { "name": "Torsten Ullrich",
8                  "id":   123456 };
9  var object3 = { };
10
11 print(object1);                  // output: >> {} <<
12 print(object1 == object2);       // output: >> FALSE <<
13 print(object1 == object3);       // output: >> FALSE <<
14 print(object1 typeof "OBJECT"); // output: >> TRUE <<
15
16 //
17 // The toString-, toJSON-, and toXML-operators are
18 // defined for all Euclides data types:
19 //
20 print();
21 print("toString conversion operator:");
22 print(''object2);
23
24 print();
25 print("toJSON conversion operator: ");
26 print({}object2);
27
28 print();
29 print("toXML conversion operator: ");
30 print(<>object2);
31
32 //
33 // Objects (and arrays) in Euclides may contain cyclic references:
34 //
35 object2@"cyclic_reference" = object2;
36
37 //
38 // The toString operator avoids cyclic references and inserts \u2026
39 //
40 print();
41 print("toString conversion with cyclic references:");
42 print(''object2);
43
44 //
45 // The XML format uses the id-idref mechanism, which
46 // has been already shown in the array examples.
47 //
48 print();
49 print("toXML conversion with cyclic references:");
50 print(<>object2);
51
```

```
52  //
53  // The toJSON operator throws an exception,
54  // as the JSON format cannot handle cyclic references.
55  //
56  print();
57  print("toJSON conversion with cyclic references:");
58  var json;
59  try {
60     json = {} object2;
61  } catch (var message) {
62     print(json);               // output: >> UNDEFINED <<
63     print(message);            // output: >> cyclic dependency in ↩
64                                //  serialization cannot be resolved
65  }
```

The output of the compiled and executed source code 1.33 is:

Output 1.34 — OBJECT data.

```
{}
FALSE
FALSE
TRUE

toString conversion operator:
{id : 123456, name : Torsten Ullrich}

toJSON conversion operator:
{
  "\u0069\u0064" :
    123456,
  "\u006E\u0061\u006D\u0065" :
    "\u0054\u006F\u0072\u0073\u0074\u0065\u006E\u0020\u0055\u006C\u006C\u0072\u0069\u0063\u0068"
}

toXML conversion operator:
<?xml version="1.0" encoding="UTF-8"?>
<object id="id0">
 <element key="id">
  <integer>123456</integer>
  </element>
 <element key="name">
  <string>Torsten Ullrich</string>
  </element>
</object>


toString conversion with cyclic references:
{cyclic_reference : { ... }, id : 123456, name : Torsten Ullrich}

toXML conversion with cyclic references:
<?xml version="1.0" encoding="UTF-8"?>
<object id="id0">
 <element key="cyclic_reference">
  <object idref="id0"/>
 </element>
 <element key="id">
  <integer>123456</integer>
 </element>
```

```
 <element key="name">
  <string>Torsten Ullrich</string>
 </element>
</object>


toJSON conversion with cyclic references:
UNDEFINED
cyclic dependency in serialization cannot be resolved
```

EUCLIDES supports several operations on objects, and each object has predefined, built-in functions. The following source code example demonstrates their usage:

Source Code 1.35 — Data Type OBJECT.

```
1  import 'io'
2
3  //
4  // Operators on objects:
5  //
6  var obj1 = { };
7  var obj2 = { "name" : "Torsten Ullrich",
8               "id"   : 123456 };
9  var obj3 = { "name" : "Christoph Schinko",
10              "id"   : 23456 };
11
12 //
13 // Relational operations:
14 //
15 print(obj2 == obj3);          // output: >> FALSE <<
16 print(obj2 != obj3);          // output: >> TRUE <<
17 print(obj1 typeof "OBJECT"); // output: >> TRUE <<
18 print("id" in obj1);          // output: >> FALSE <<
19 print("id" in obj2);          // output: >> FALSE <<
20 print(123456 in obj2);        // output: >> TRUE <<
21
22 //
23 // Access and modification
24 //
25 obj3@"id" = 987654321;
26 print(obj3@"id");             // output: >> 987654321 <<
27
28 //
29 // Built-in functions
30 //
31 print(obj3@"size"());         // output: >> 2 <<
32 print(obj3@"keys"());         // output: >> [id, name] <<
33 print(obj3@"values"());       // output: >> [987654321, ↩
34                               //  Christoph Schinko] <<
35
36 //
37 // It is possible to add a key named "keys",
38 // but this is strongly discouraged!
39 //
40 obj3@"keys" = ["1", "2"];
41 print(obj3@"keys");           // output: >> [1, 2] <<
42 print(obj3);
43
```

```
//
// Deleting the key-value-pair named "keys" restores
// the previous default behavior.
//
obj3@"keys" = undefined;
print(obj3@"keys"());         // output: >> [id, name] <<

//
// Redefining the key-value-pair called "toString" is possible
// as well. The redefinition of "toString" results in an object
// with a key-value-pair ("tostring" : function). This pair is
// visible in JSON (mapped to null) and XML (mapped to <function/>).
//
obj2@"toString" = function() { return ":-)"; };
print(obj2);                  // output: >> :-) <<
print(''obj2);                // output: >> :-) <<
```

The **eval** operator can be used to initialize objects and other EUCLIDES data types:

Source Code 1.36 — Operator EVAL.

```
import 'io'

//
// The eval operator supports another possibility to create
// an object by a valid JSON string:
//
var json_1 = ""
    ++ "{ \"order\": 4711, \n"
    ++ "  \"items\": [ \n"
    ++ "    { \"name\": \"IC12635\", \n"
    ++ "      \"quantity\": 10 }, \n"
    ++ "    { \"name\": \"LM358N\", \n"
    ++ "      \"quantity\": 2 } \n"
    ++ "  ] \n"
    ++ "}";
//
// init object via string eval
//
var object_1 = eval(json_1);
print(object_1);

//
// json mapping:
//    "null"              is mapped to undefined
//    "true" and "false" are mapped to boolean values
//    numbers are mapped to integers (if possible) or to floats
//    arrays and objects are mapped to Euclides arrays and objects
//
print(eval("null"));          // output: >> UNDEFINED <<
print(eval("true"));          // output: >> TRUE <<
print(eval("123"));           // output: >> 123 <<
print(eval("[ 1, 2, -3]"));   // output: >> [1, 2, -3] <<
try {
   print(eval("[ 1, 2, 3.0]"));
} catch (var error) {         // EXCEPTION: Euclides does not allow to
   print(error);              // mix integers and floats in one array.
}
```

1.13 Type: Function

In EUCLIDES a function is defined by an expression including a sequence of statements.

Source Code 1.37 — Data Type FUNCTION.

```
1  import 'io'
2
3  //
4  // Function data type:
5  //
6  var fun1 = function()     { return 0;       };
7  var fun2 = function(x)    { return x + 1.0; };
8  var fun3 = function(x, y) { return x + y;   };
9  var fun4 = function(f, array) {
10    var result = [];
11    var index  = 0;
12    while (index < array@"size"()) {
13       result = result ++ [f(array@index)];
14       index  = index + 1;
15    }
16    return result;
17 };
18
19 print(fun1 == fun2);          // output: >> FALSE <<
20 print(fun1 != fun3);          // output: >> TRUE <<
21 print(fun1 typeof "FUNCTION");// output: >> TRUE <<
22
23 print(fun1());                // output: >> 0 <<
24 print(fun2(17.0));            // output: >> 18.0 <<
25 print(fun3(17.0, -28.0));     // output: >> -11.0 <<
26
27 print(fun4(fun2,
28    [1.0, 2.0, 3.0]));         // output: >> [2.0, 3.0, 4.0] <<
29
30 //
31 // The toString-, toJSON-, and toXML-operators are
32 // defined for all Euclides data types:
33 //
34 print();
35 print("toString conversion operator:");
36 print(''fun1);
37
38 print();
39 print("toJSON conversion operator: ");
40 print({}fun1);
41
42 print();
43 print("toXML conversion operator: ");
44 print(<>fun1);
```

The result of source code 1.37 is shown in output 1.38:

Output 1.38 — FUNCTION data.

```
FALSE
TRUE
TRUE
0
```

```
18.0
-11.0
[2.0, 3.0, 4.0]

toString conversion operator:
FUNCTION

toJSON conversion operator:
null

toXML conversion operator:
<?xml version="1.0" encoding="UTF-8"?>
<function/>
```

If a function is part of an object, and if it is accessed via the object and executed within the same expression, the **this** reference points to the object the function belongs to; e.g.:

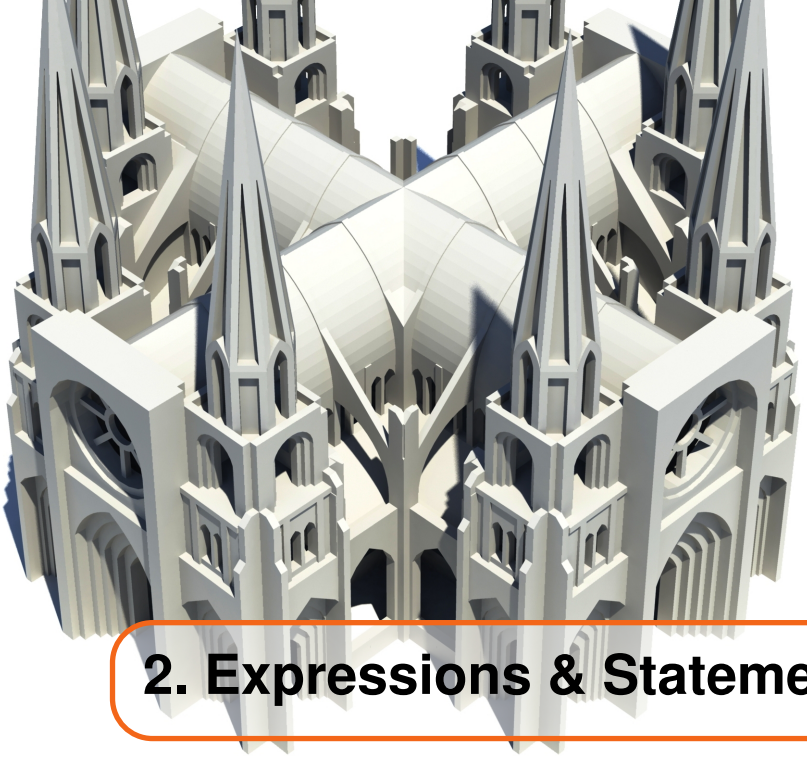
Source Code 1.39 — Data Type FUNCTION.

```
1  import 'io'
2
3  var object = {
4     "data"   : 42,
5     "method" : function() { print(this@"data"); }
6  };
7
8  object@"method"();            // output: >> 42 <<
9  var outOfObject = object@"method";
10
11 //
12 // Without the surrounding object, the this pointer points
13 // to the global environment, which does not define a "data"
14 // element. Therefore, this@"data" is UNDEFINED. As a consequence,
15 // evaluating outOfObject returns UNDEFINED:
16 //
17 outOfObject();                // output: >> UNDEFINED <<
18
19 //
20 // Now let's store the function in another object.
21 //
22 var another = {
23    "data"   : "Hello World",
24    "method" : outOfObject
25 };
26 another@"method"();           // output: >> Hello World <<
27
28 //
29 // The object-function bind operator creates a new function in
30 // which the this reference is bound to a fixed object.
31 // This functionality is often used for callbacks.
32 //
33
34 // non-bound function:
35 object@"method"();            // output: >> 42 <<
36 outOfObject();                // output: >> UNDEFINED <<
37
38 // bound function using bind (::) operator:
39 var boundToObject = object::"method";
40 boundToObject();              // output: >> 42 <<
```

A function may have optional parameters. Each optional parameter must have a default value or a default reference, which is used instead. In the list of parameters (at a function declaration), the non-optional parameters must be listed first.

Source Code 1.40 — Data Type FUNCTION.

```
import 'io'

var value = 42;

//
// Functions are usually declared as constants.
//
const fun = function(x, y = value, z = 3) {
   var xxyz = x * x + y * z;
   print(''x ++ "*" ++ ''x ++ "+" ++ ''y ++ "*" ++ ''z ++ "=" ++ ''xxyz);
};

fun(4);                         // output: >> 4*4+42*3=142 <<
fun(4, 1);                      // output: >> 4*4+1*3=19 <<
fun(4, 1, 2);                   // output: >> 4*4+1*2=18 <<

//
// Each function copys its default parameters at daclaration.
// As a consequence, modifying them afterwards has no effect.
//
value = 50;
fun(4);                         // output: >> 4*4+42*3=142 <<
fun(4, 1);                      // output: >> 4*4+1*3=19 <<
fun(4, 1, 2);                   // output: >> 4*4+1*2=18 <<

//
// The copied default parameter is not a deep copy:
// the object, whose reference is used as a default parameter,
// can be modified, and the function uses the same object.
//
var counter = {
    "value" : 0,
    "inc"   : function() {
        this@"value" = this@"value" + 1;
        print(this@"value");
    }
};

const count = function(x = counter) {
    x@"inc"();
};

count(counter);                 // output: >> 1 <<
counter@"inc"();                // output: >> 2 <<

count({
  "inc" : function () {
    print(":-)");
    }
  });                           // output: >> :-) <<

counter@"inc"();                // output: >> 3 <<
count(counter);                 // output: >> 4 <<
```



2. Expressions & Statements

The language Euclides is closely related to JavaScript, Java, and C++. Its source code is a list of import commands to include libraries followed by a sequence of statements. An import command is composed of the keyword **import** and a library name in single quotes. For example, the import command **import** 'io' is interpreted by the compiler to include the Euclides library io. Depending on compilation settings the compiler searches for the file io.ecl in the library path. The corresponding start rule is:

```
script
        : (importCommand)* (statement)* EOF
        ;
```

with

```
statement
        : statementBlock
        | statementConstantDeclaration
        | statementVariableDeclaration
        | statementExpression
        | statementIf
        | statementSwitch
        | statementFor
        | statementWhile
        | statementBreak
        | statementReturn
        | statementThrow
        | statementTryCatch
        | statementAnnotation
        | statementNativeCode
        ;
```

The language EUCLIDES has two different kind of comments, which can be at arbitrary position within source code: short comments and long comments. A short comment starts with `//` and ends at the end of the line in which it started. A long comment starts with `/**`, can include several line breaks and ends with `*/`. In contrast to other languages that support a similar functionality (such as C, C++, Java, etc.), EUCLIDES does not support comments starting with `/*` instead of `/**`.

2.1 Declaration Statements

The language EUCLIDES knows variables and constants. A variable has the default value **`undefined`** until a value has been set. A constant needs a value at declaration time and cannot be changed. The corresponding grammar rules are

```
statementConstantDeclaration
        : CONST IDENTIFIER '=' expression ';'
        ;
```

and

```
statementVariableDeclaration
        : VAR IDENTIFIER ('=' expression)? ';'
        ;
```

Each identifier has a name which starts with a letter or an underscore followed by an arbitrary number of letters, digits, or underscores:

```
IDENTIFIER
        : ('_' | 'a'..'z' | 'A'..'Z') ('_' | 'a'..'z' | 'A'..'Z' | '0'..'9')*
        ;
```

2.2 Expressions

An expression is a (left-)recursive structure and the operator precedence is just the order of definition listed below:

```
expression
        : expressionPrimary
        | expression '@' expression
        | expression '(' (expression (',' expression)*)? ')'
        | ('+' | '-') expression
        | ('!'  | '$' | '~' | ''' | '{}' | '<>') expression
        | expression '^' expression
        | expression ('*' | '#' | '/' | '%') expression
        | expression ('+' | '-') expression
        | expression '++' expression
        | expression ('<' | '>' | '<=' | '>=' | IN | TYPE_OF) expression
        | expression ('==' | '!=') expression
        | expression '&&' expression
        | expression '||' expression
        | expression '=' expression
        ;
```

The primary expression expressionPrimary is either

- **undefined**, **true**, **false**, **this** or
- a string literal, a numerical literal, an identifier, or
- a vector-, matrix-, array-, object-, function-definition, or
- an expression in parentheses '(' expression ')'.

Please note that valid grammar words are not necessarily valid programs; e.g. the left hand side of an assignment expression '=' expression must be assignable. As a consequence **true** = **false**; is not a valid statement accepted by the EUCLIDES compiler.

A single valid expression forms a valid statement:

```
statementExpression
        : expression
        ;
```

2.3 Blocks

Several statements can be combined to a block of statements:

```
statementBlock
        : '{' (statement)* '}'
        ;
```

2.4 Conditional Statements

The language EUCLIDES comprehends two conditional statements: if-statements and switch-statements. The corresponding grammar rules are

```
statementIf
        : IF '(' expression ')' statement (ELSE statement)?
        ;
```

and

```
statementSwitch
        : SWITCH '(' expression ')' (switchcase)+ (DEFAULT statement)?
        ;
```

with

```
switchcase
        : CASE '(' expression ')' statement
        ;
```

The semantics of a switch-statement in EUCLIDES is different to many other programming languages:

1. The first expression in the switch-statement is evaluated and stored in a temporary, internal identifier `tmp`.
2. Each case is identified by an expression in parentheses. The expression is evaluated and stored in another temporary, internal identifier `case0`. The type of `case0` is expected to be an array or an object.
3. The statement of the corresponding case is executed if the expression `tmp` **in** `case0` evaluates to **true**.
4. Similar to an if-statement, each case consists of one statement. If several statements should be executed, they have to be combined by a block statement.

5. Only one case is executed. The first case, whose expression contains the value/reference of `tmp`, terminates the switch-statement. The "Fall-Through" mechanism known from languages such as C, C++, Java, etc. is not supported in Euclides.

The following example illustrates the behavior.

Source Code 2.1 — Statement SWITCH.

```
1  import 'io'
2
3  var value = 42;
4  switch (value)
5      case ( [0]    ) print("zero");
6      case ( [1, 2] ) print("one or two");
7      default         print("many");
```

The Euclides compiler normalizes the source code during the parse tree construction and analysis step. In this phase a switch-statement is reduced to a sequence of nested if-else-statements. In detail, the source code 2.1 is equivalent to source code 2.2.

Source Code 2.2 — Statement IF-ELSE.

```
1  import 'io'
2
3  var value = 42;
4  if (value in [0]) {
5      print("zero");
6  } else {
7      if (value in [ 1 , 2 ]) {
8          print("one or two");
9      } else {
10         print("many");
11     }
12 }
```

2.5 Loop Statements

Loop statements can be used to iterate over arrays and objects. Furthermore, loop statements can be used to repeat a (block-)statement several times. For the first use-case Euclides provides a for-loop:

```
statementFor
        : FOR '(' VAR IDENTIFIER ':' IDENTIFIER IN expression ')'
            statement
        ;
```

The first identifier and the second one form a key-value pair. If the expression is an array, then the key contains the array index and the value contains the corresponding array elements. If the expression is an object, then the key-value pair contains the object's key-value elements.

A more generalized loop is EUCLIDES' while-statement. It is suitable for the second use-case; i.e. for repeating a (block-)statement several times. The grammar rule is:

```
statementWhile
        : WHILE '(' expression ')'
            statement
        ;
```

Both loops can be interrupted by a jump statement.

2.6 Jump Statements

The execution path in an EUCLIDES application can be changed via the jump statements: **break**, **return**, and **throw**. A **break** statement can be used to exit a loop (independent of the loop condition).

A **return** statement can be used to exit a function. An optional return value will be passed out of the function an can be used by the surrounding environment that has called the function.

A **throw** statement creates an exception, which should be caught by the surrounding, calling environment.

2.7 Exceptions

In an exceptional situation, an EUCLIDES application may throw an exception. The exception contains a passed value / reference:

```
statementThrow
        : THROW expression ';'
        ;
```

The thrown value / reference specified in the expression of the throw-statement is assigned to the identifier defined in the try-catch-statement:

```
statementTryCatch
        : TRY statement CATCH '(' VAR IDENTIFIER ')' statement
        ;
```

2.8 Annotations & Native Code

Two statements in EUCLIDES provide access to the internal data structures and control flows generated by the compiler. These statements are specific to a compiler version and should not be used in "normal" source code.

The compiler settings can be set within the source code using annotations. These annotation statements are

```
statementAnnotation
        : '/*@' ( annotation )* '*/'
        ;
```

with

```
annotation
        : IDENTIFIER ('.' IDENTIFIER)* ('=' (UNDEFINED|TRUE|FALSE
            |STRING_LITERAL|NUMERICAL_LITERAL))? ';'
        ;
```

The second type of statement to access to the internal data structures and control flows generated by the compiler are native-code-statements:

```
statementNativeCode
        : NATIVE_CODE
        ;
```

with

```
NATIVE_CODE
        : '/*%' .*? '*/'
        ;
```

The inner part of the native code (`.*?`) is interpreted by the compiler back-end. It should start with the back-end name, followed by a mapping of identifiers in Euclides to identifiers in the target platform code, followed by `%%` and platform specific code. In this way it is possible to reuse library functions already available at the target platform. The following example (source code 2.3) illustrates the reuse of Java's math library in Euclides.

Source Code 2.3 — Statement NATIVE.

```
1  import 'io'
2
3  const cos = function(x){
4      var result;
5
6      /*%java8%result%__result__%x%__x__%%
7      double input  = (__x__).toFloat(RUNTIME, ii);
8      double output = Math.cos(input);
9      __result__ = RUNTIME.initFloat(ii, output);
10     */
11
12     return result;
13 };
14
15 print(cos(1.0));
```

The source code 2.3 contains one native code statement in the lines 6–10.

line 6: `/*%java8` The native code statement starts with the name of the back-end (`java8`). All other back-ends will ignore this statement; only the `java8` target platform will include the corresponding native code in its generated code.

line 6: `%result%__result__` This declaration describes the mapping of an Euclides' identifier to the corresponding target platform. Without this declaration, identifiers have automatically generated, "unreadable" names.

line 6: `%x%__x__` Each variable used in the target platform code has to be declared. Of course, the variable / constant has to be visible and accessible within the current scope.

line 6: `%%` This token closes the declaration. The following code is copy-and-pasted into the target platform code. As a consequence, all following statements are Java-statements and not Euclides statements.

line7: `double input = (__x__).toFloat(RUNTIME, ii);` The Euclides identifier `x` is available in the Java context. It is mapped onto a Java type according to the back-end settings; namely `euclides.Var __x__;` The java8 back-end provides access functions such as `toFloat`. The parameter `RUNTIME` offers access to run-time functions (data type factory, operators, etc.); the parameter `ii` is an index into a look-up table

referencing the input source code. This parameter can be used to provide sensible run-time exceptions, warnings, and errors.

line 8: `double output = Math.cos(input);` This is a pure Java statement using the included mathematics library of the Java Runtime Environment.

line 9: `__result__ = RUNTIME.initFloat(ii, output);` This Java statement uses the Java implementation of the Euclides run-time to initialize a data type FLOAT. The result is an instance of the FLOAT data type implementing the already known `euclides.Var` interface mentioned above. This result is stored in `__result__`, which is available and visible in the Euclides context.

line 10: `*/` This token closes the native statement.

2.9 Examples

The following source codes shows some well-known algorithms implemented in Euclides. These examples shall demonstrate the usage of the Euclides language elements in relation to each other.

Sieve of Eratosthenes

From Wikipedia, the free encyclopedia:

> In mathematics, the sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to any given limit.
>
> It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2. The multiples of a given prime are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime. This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime.
>
> The earliest known reference to the sieve is in Nicomachus of Gerasa's *Introduction to Arithmetic*, which describes it and attributes it to Eratosthenes of Cyrene, a Greek mathematician.

The implementation in Euclides is shown in source code 2.4.

Source Code 2.4 — Sieve of Eratosthenes.

```
1  import 'io'
2
3  //
4  // init data structures and constant limits
5  //
6  const limit = 10000;
7  var isPrime = [ ];
8  var index = 0;
9  while (index <= limit) {
10     isPrime@index = true;
11     index = index + 1;
12 }
13 isPrime@0 = false;
14 isPrime@1 = false;
15 index = 2;
16
```

```
17  //
18  // sieve
19  //
20  while (index <= limit) {
21      //
22      // go to next prime
23      //
24      if (isPrime@index) {
25          //
26          // print it
27          //
28          print(index);
29          //
30          // mark its multiples as non-prime
31          //
32          var multiples = 2;
33          while (multiples * index <= limit) {
34              isPrime@(multiples * index) = false;
35              multiples = multiples + 1;
36          }
37      }
38      index = index + 1;
39  }
```

Euclidean Algorithm

From Wikipedia, the free encyclopedia:

> In mathematics, the Euclidean algorithm is an efficient method for computing the greatest common divisor (GCD) of two numbers, the largest number that divides both of them without leaving a remainder. It is named after the ancient Greek mathematician Euclid, who first described it in his Elements (c. 300 BC). It is an example of an algorithm, a step-by-step procedure for performing a calculation according to well-defined rules, and is one of the oldest algorithms in common use. It can be used to reduce fractions to their simplest form, and is a part of many other number-theoretic and cryptographic calculations.
>
> The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if the larger number is replaced by its difference with the smaller number. Since this replacement reduces the larger of the two numbers, repeating this process gives successively smaller pairs of numbers until the two numbers become equal. When that occurs, they are the GCD of the original two numbers.
>
> The version of the Euclidean algorithm described above (and by Euclid) can take many subtraction steps to find the GCD when one of the given numbers is much bigger than the other. A more efficient version of the algorithm shortcuts these steps, instead replacing the larger of the two numbers by its remainder when divided by the smaller of the two (with this version, the algorithm stops when reaching a zero remainder). With this improvement, the algorithm never requires more steps than five times the number of digits (base 10) of the smaller integer. This was proven by Gabriel Lamé in 1844, and marks the beginning of computational complexity theory. Additional methods for improving the algorithm's efficiency were developed in the 20th century.

Source Code 2.5 — Euclidean Algorithm.

```
import 'io'

const primes = [  2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
    53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127,
    131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197,
    199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277,
    281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367,
    373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449,
    457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547,
    557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631,
    641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727,
    733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823,
    827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919,
    929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997];

var v1 = primes@42 * primes@43 * primes@44 * primes@45;
var v2 =             primes@43 * primes@44 * primes@45 * primes@46;
var reference = primes@43 * primes@44 * primes@45;

print(v1);                    // output: >> 1445140189 <<
print(v2);                    // output: >> 1596463769 <<
print(reference);             // output: >> 7566179 <<

//
// Greatest Common Divisor aka Euclidean algorithm (version 1)
//
const gcd1 = function(a, b) {
    while (a != b)
        if (a > b)
            a = a - b;
        else
            b = b - a;
    return a;
};

var result1 = gcd1(v1, v2);
print(result1 == reference);  // output: >> TRUE <<

//
// Greatest Common Divisor aka Euclidean algorithm (version 2)
//
const gcd2 = function(a, b) {
    if (b == 0)
        return a;
    else
        return gcd2(b, a % b);
};

var result2 = gcd2(v1, v2);
print(result2 == reference);  // output: >> TRUE <<
```

Hexadecimal

From Wikipedia, the free encyclopedia:

> In mathematics and computing, hexadecimal (also base 16, or hex) is a positional numeral system with a radix, or base, of 16. It uses sixteen distinct symbols, most often the symbols $0-9$ to represent values zero to nine, and $A-F$ (or alternatively $a-f$) to represent values ten to fifteen.
>
> Hexadecimal numerals are widely used by computer system designers and programmers, as it provides a more human-friendly representation of binary-coded values. Each hexadecimal digit represents four binary digits, also known as a nibble, which is half a byte. For example, a single byte can have values ranging from 0000 0000 to 1111 1111 in binary form, which can be more conveniently represented as 00 to *FF* in hexadecimal.

The following algorithm (see source code 2.6) converts a hexadecimal representation into a decimal value.

Source Code 2.6 — Hexadecimal Code.

```
1  import 'io'
2
3  const hex2dec = function(hexcode) {
4      print("converting hexcode \'" ++ ''hexcode ++ "\':");
5      var value    = 0;
6      var position = 0;
7      var length   = hexcode@"size"();
8      //
9      while (position < length) {
10         var c = hexcode@position;
11         var v;
12         var power = 16^(length - 1 - position);
13         switch (c)
14             case (["1"]) v = 1 * power;
15             case (["2"]) v = 2 * power;
16             case (["3"]) v = 3 * power;
17             case (["4"]) v = 4 * power;
18             case (["5"]) v = 5 * power;
19             case (["6"]) v = 6 * power;
20             case (["7"]) v = 7 * power;
21             case (["8"]) v = 8 * power;
22             case (["9"]) v = 9 * power;
23             case (["a", "A"]) v = 10 * power;
24             case (["b", "B"]) v = 11 * power;
25             case (["c", "C"]) v = 12 * power;
26             case (["d", "D"]) v = 13 * power;
27             case (["e", "E"]) v = 14 * power;
28             case (["f", "F"]) v = 15 * power;
29             default v = 0;
30         print(c ++ " = " ++ ''v);
31         value    = value    + v;
32         position = position + 1;
33     }
34     print("result = " ++ ''value);
35 };
36 hex2dec("c8f3");
```

Merge Sort

From Wikipedia, the free encyclopedia:

> In computer science, merge sort is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945. A detailed description and analysis of bottom-up merge sort appeared in a report by Goldstine and von Neumann as early as 1948.

Source Code 2.7 — Merge Sort.

```
import 'io'

const mergesort = function(array) {

    //
    // merge step
    //
    const merge = function(sort, merge, a, b, low, pivot, high) {
        var h = low;
        var i = low;
        var j = pivot + 1;
        //
        while ((h<=pivot) && (j<=high)) {
            if (a@h <= a@j) {
                b@i = a@h;
                h = h + 1;
            } else {
                b@i = a@j;
                j = j + 1;
            }
            i = i + 1;
        }
        //
        if (h > pivot) {
            var k = j;
            while (k <= high) {
                b@i = a@k;
                i = i + 1;
                k = k + 1;
            }
        } else {
            var k = h;
            while (k <= pivot) {
                b@i = a@k;
                i = i + 1;
                k = k + 1;
            }
        }
        //
        var k = low;
        while (k <= high) {
            a@k = b@k;
            k = k + 1;
        }
    };
```

```
46
47      //
48      // divide-and-conquer step
49      //
50      const sort = function(sort, merge, a, b, low, high) {
51          var pivot;
52          if (low < high) {
53              pivot = (low + high) / 2;
54              sort(sort, merge, a, b, low, pivot);
55              sort(sort, merge, a, b, pivot+1, high);
56              merge(sort, merge, a, b, low, pivot, high);
57          }
58      };
59
60      //
61      // top level entry point
62      //
63      var size   = array@"size"();
64      var result = [ ];
65      sort(sort, merge, array, result, 0, size - 1);
66      return array;
67  };
68
69
70  //
71  // example 1:
72  //
73  var a = [12, 10, 43, 23, -78, 45, 123, 56, 98, 41, 90, 24];
74  mergesort(a);
75  print(a);
76
77  //
78  // example 2:
79  //
80  var b = a ++ a ++ a;
81  mergesort(b);
82  print(b);
```

3. Library Concepts

In order to simplify the development of EUCLIDES applications, commonly-used routines and functions are available via libraries.

3.1 Library Names & Files

The source code in EUCLIDES is a list of import commands to include libraries followed by a sequence of statements. An import command is composed of the keyword **`import`** and a library name in single quotes. For example, the import command **`import`** `'io'` is interpreted by the compiler to include the EUCLIDES library `io`. Depending on compilation settings the compiler searches for the file `io.ecl` in the library path. The corresponding start rule is:

```
script
        : (importCommand)* (statement)* EOF
        ;
```

with

```
importCommand
        : IMPORT MODULE_LITERAL
        ;
```

and

```
MODULE_LITERAL
        : '' ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')+ ''
        ;
```

The libraries are imported (and parsed, and analyzed) in the order they are listed in the sequence of import commands. Already imported libraries are skipped automatically. There is no need to prevent manually a library being loaded twice.

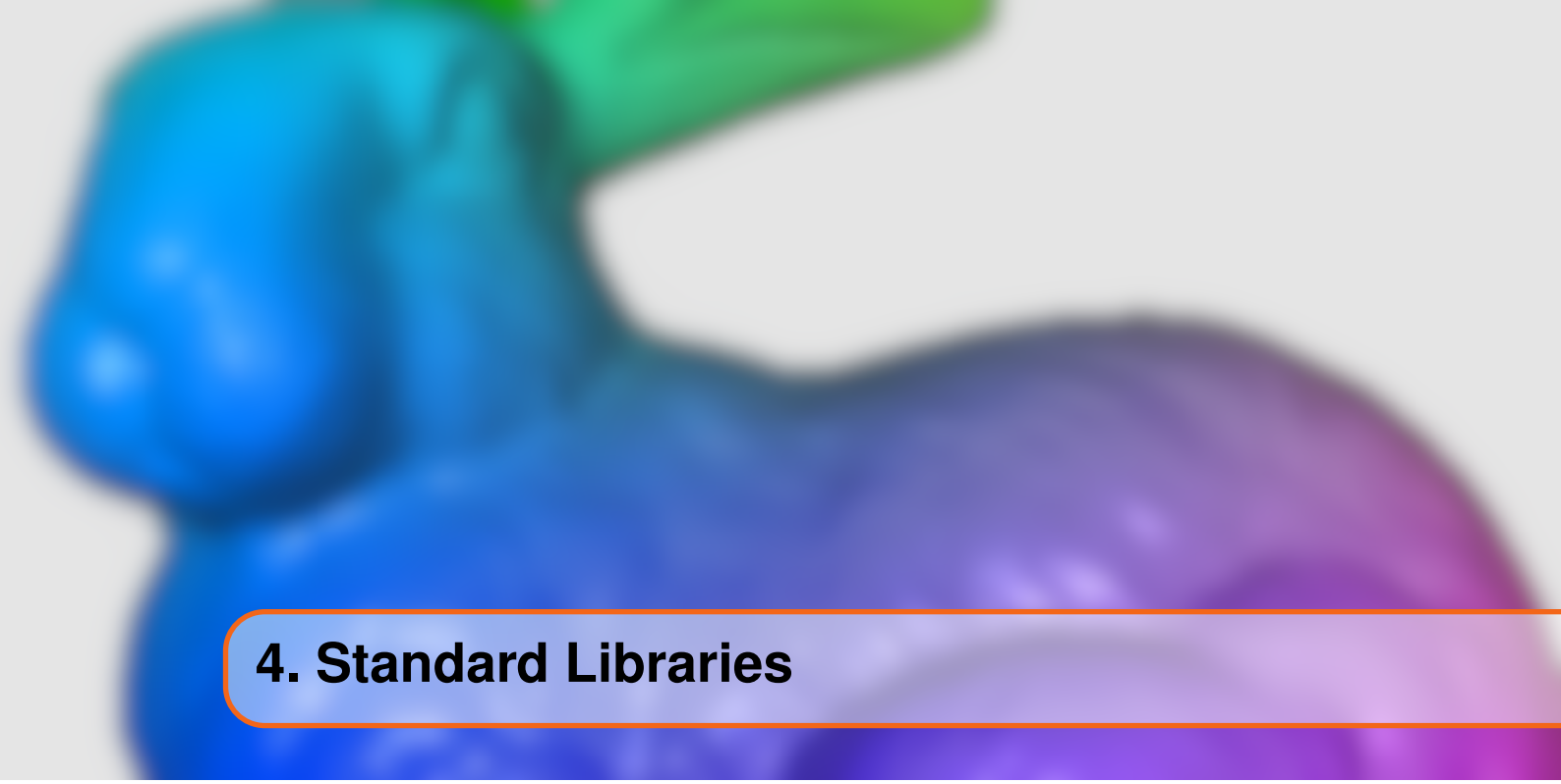
Circular dependencies, in which a library *A* depends on another library *B*, which depends on the previously mentioned library *A*, have to be avoided, as the compiler cannot resolve these circular references.

3.2 Scoping and Visibility

Having parsed an EUCLIDES script all variables and constants are internally represented by unique symbols in a symbol table. The table has a strict hierarchical order that defines the visibility of a symbol.

The visibility of symbols in libraries (ecl files) and source code (ecs files) is the same with one important exception: if a library defines a symbol, and if the symbol's name starts with an underscore, then the symbol is anonymized after the parsing of the library. In detail, the symbol can be referenced within the library, but before loading another library or source code, the symbol is renamed with an internal, "anonymous" name. In this way, libraries can define global variables and constants, which are not visible from "outside". This technique is used to avoid global namespace pollution.

II Euclides Libraries



4. Standard Libraries

The EUCLIDES framework is distributed with a set of standard libraries:

io The *io* library provides functions for stream-based input and output routines (such as `print` commands), for file handling, and for web publishing.

text The *text* library offers string-based functions for text handling.

math Commonly used mathematical functions (abs, min, max, sin, cos, exp, ...) are implemented in the *math* library.

sequence Convenience functions for array handling and interpolation can be found in the *sequence* library.

blas The *blas* library is a collection of basic linear algebra subprograms including generators for 3d-translation and 3d-rotation 4×4 matrices.

color In EUCLIDES colors are represented by a 3d vector with red, green, blue values between 0.0 and 1.0. The *color* library provides often used color definitions (see `colorNames`) and color schemes (see `colorMaps`).

image Images can be handled in EUCLIDES using the *image* library.

material Images and colors can be combined to define simple materials. The *material* library provides a constructor and commonly used, predefined materials.

4.1 Library: IO

Euclides Standard Library for file- and stream-based input and ouput routines.

This library provides a convenient function to print text to the standard output stream named `print`. Furthermore, it provides additional functions for printing:

- `printOut`, `flushOut` for printing to standard output stream,
- `printErr`, `flushErr` for printing to standard error stream.

For file input and output operations the library provides the functions

- `textIn`, `textOut` for platform independent text files in UTF-8 encoding,
- `fileIn`, `fileOut` for handling binary files,
- `resourceBinary`, `resourceText` for accessing embedded files.

Furthermore, the library has a `input` function to read in console input.

Last but not least, the library has a `publish` function to start a web server, which forwards all requests to callback functions defined in the `publish` configuration.

---

print:

```
STRING="" → ( )
```

The function `print` writes a text message to standard out including a newline at the end. Its parameter will be converted to a string automatically, if necessary. If text shall not be written to the standard error stream instead of the standard output stream, or if no newline at the end of the message shall be printed, then the preferred method to use is `printOut` or `printErr`.
A typical example of the print function is the well known hello-world-program:

Source Code 4.1 — Example.

```
1  import 'io'
2
3  print("Hello World!");
```

---

input:

```
( ) → STRING
```

The function `input` reads a line from the system input stream and returns it as a string.

Source Code 4.2 — Example.

```
1  import 'io'
2
3  var text = input();
4  var index = text@"size"();
5
6  while (index > 0) {
7     index = index - 1;
8     printOut(text@index);
9  }
10 flushOut();
```

---

printout:

```
STRING → ( )
```

The function `printOut` writes a text message to the standard output stream without a newline at the end. Its parameter will be converted to a string, if necessary. Due to stream buffering the text message may not be printed at once. Use `flushOut` to ensure that the buffer content is printed synchronously.

---

flushout:

```
( ) → ( )
```

Depending on the target platform standard out uses caching and buffering mechanisms. This function ensures that the buffer contents are written to standard out.

---

printerr:

```
STRING → ( )
```

The function `printErr` writes a text message to standard error without a newline at the end. Its parameter will be converted to a string, if necessary. Due to stream buffering the text message may not be printed at once. Use `flushErr` to ensure that the buffer content is printed synchronously.

---

flusherr:

```
( ) → ( )
```

Depending on the target platform standard error uses caching and buffering mechanisms. This function ensures that the buffer contents are written to standard error.

---

textin:

```
STRING → ARRAY<STRING>
```

This `textIn` function takes a string parameter `filename`, reads the text file of the same name, parses its content, and returns an array of strings, which contains the file content line by line without the newline characters at the end of each line.

---

textout:

```
(STRING, ARRAY<TEMPLATE_A>) → ( )
```

The function `textOut` takes two parameters (`fileName` and `fileContent`) and writes the text file named `fileName`. If `fileContent` is an array, the resulting text file will be composed of the array elements converted to strings separarted by newline characters. If `fileContent` is not an array, it will be converted to a string that is written directly into the file.

---

filein:

```
STRING → OBJECT{"name" : STRING, "type" : STRING, "content" : ARRAY<INTEGER>}
```

The function `fileIn` reads the binary file, whose name is passed as parameter. It returns an object with the entries `name`, `type`, and `content`. `name` and `type` are strings whereas `content` is an array of integers representing the file content as a sequence of bytes.

---

fileout:

```
OBJECT{ "name" : STRING, "type" : STRING, "content" : ARRAY<INTEGER>} → ( )
```

The function `fileOut` takes one parameter representing a file. A file representation is an object with the entries `name`, `type`, and `content`. `name` and `type` are strings, whereas `content` is an array of integers representing the file content as a sequence of bytes.

---

resourcebinary:

`STRING → ARRAY<INTEGER>`

The function `resourceBinary` takes one parameter representing a resource; i.e. an embedded, compiled file. In contrast to `fileIN` a resource is resolved at compile time and its content is included in the resulting, compiled binary. Via `resourceBinary` the resource can be accessed. The function returns an array of integers representing the embedded file content as a sequence of bytes:

Source Code 4.3 — Example.

```
1  import 'io'
2
3  //
4  // The resource mechanism of Euclides can be used to
5  // deploy static files by embedding them into the compiled
6  // binary. each embedded file has to be located in the
7  // library path and has to be named by an identifier;
8  // e.g. to embed the file "euclides_64.png" with identifier
9  // 'euclides_logo' add a compiler annotation:
10 //
11
12 /*@
13   resource.euclides_logo = "euclides_64.png";
14  */
15
16 //
17 // Access the embedded file content (the file is now
18 // accessed from the compiled executable; the file
19 // "euclides_64.png" is not needed after compilation):
20 //
21 var data = resourceBinary("euclides_logo");
22
23 var file = {
24     "name"    : "euclides_test.png",
25     "type"    : "application/octet-stream",
26     "content" : data };
27
28 fileOut(file);
29 print(":-)");
```

---

resourcetext:

`STRING → ARRAY<INTEGER>`

The function `resourceText` takes one parameter representing a resource; see `resourceBinary` for further details on resources. The function returns an array of strings representing the embedded file content line by line without the newline characters at the end of each line.

---

publish:

```
OBJECT{ “port” : INTEGER, “log” : STRING → ( ), “entry” : ARRAY<OBJECT{ }>} → ( )
```

The function `publish` takes one parameter representing a web server configuration including its entry points and the corresponding callback functions.

A minimal example is:

Source Code 4.4 — Example.

```
1  import 'io'
2  //
3  // The minimal configuration to set up a web server in Euclides
4  // only needs an entry point, a mime type and a function:
5  //
6  var config = {
7     "entry" : [
8         // each entry point has to be specified
9         { "point"             : "/index.html",
10          "contenttype"       : "text/html",
11          "contentfunction"   : function(request) {
12                  print("called index.html");
13                  return "<html><body> :-) </body></html>";
14              }
15        }
16    ]
17 };
18
19 publish(config);
```

Having started the web server, the server prints log messages to standard out including its setup and all requests at a registered entry point.

Instead of a globally defined `"contentfunction"`, it is possible to pass a bound function using the bind operator.

Please note, the web server does not stop automatically and the `publish` function does not return, until the web server has been stopped.

The following examples demonstrate the usage of the built-in web server of EUCLIDES. Let's start with a simple web server that counts the number of accesses:

Source Code 4.5 — The Counting Web Server.

```
import 'io'
//
// The state of a web server can be stored in an object.
// If the same object provides a callback function, it
// can be used by the publish function as a valid entry point:
//
var server = {
    //
    // a simple counter
    //
    "countedCalls" : 0,
    //
    // callback function
    //
    "handleRequest" : function(request) {
        this@"countedCalls" = this@"countedCalls" + 1;
        return "<html>"
            ++ "<body>"
            ++ "<h1>Euclides Web Server</h1>"
            ++ "<p>accessed " ++ ''(this@"countedCalls") ++ " times</p>"
            ++ "</body>"
            ++ "</html>";
    }
};

var config = {
   "entry" : [
       // each entry point has to be specified
       { "point"             : "/index.html",
         "contentType"       : "text/html",
         "contentFunction"   : server::"handleRequest"
       }
   ]
};

publish(config);
```

A more advanced example illustrates the usage of several entry points, the handling of requests to pass parameters, returning byte data (instead of text) and the embedding of client side JavaScript:

Source Code 4.6 — The Calculating Web Server.

```
1   import 'io'
2   import 'text'
3
4   const EUCLIDES_ICON_32x32_PNG = [
5       -119, 80, 78, 71, 13, 10, 26, 10, 0, 0, 0, 13, 73, 72, 68, 82, 0,
6       0, 0, 32, 0, 0, 0, 32, 8, 6, 0, 0, 0, 115, 122, 122, -12, 0, 0, 0,
```

. . . more binary data . . .

```
124     -13, 127, 2, 76, -111, -108, -103, 105, 110, -48, -96, 0, 0, 0, 0,
125     73, 69, 78, 68, -82, 66, 96, -126];
126
127 var server = {
128     //
129     // a simple counter
130     //
131     "countedCalls" : 0,
132     //
133     // callback function for index.html
134     //
135     "index" : function(request) {
136         //
137         // count web access
138         //
139         this@"countedCalls" = this@"countedCalls" + 1;
140         //
141         // return a static html web page
142         //
143         return "<html>"
144             ++ "<head>"
145             ++ "<link rel=\"icon\""
146             ++ " href=\"http://localhost:8080/favicon.png\""
147             ++ " type=\"image/png\">"
148             ++ "</head>"
149             ++ "<body>"
150             ++ "<h1><img src=\"favicon.png\">"
151             ++ " Euclides Web Server</h1>"
152             ++ "<ul>"
153             //
154             ++ "<li>the server has been accessed "
155             ++ ''(this@"countedCalls") ++ " times</li>"
156             //
157             ++ "<li><pre>first summand: </pre>"
158             ++ "<textarea id=\"sum1\"></textarea></li>"
159             //
160             ++ "<li><pre>second summand: </pre>"
161             ++ "<textarea id=\"sum2\"></textarea></li>"
162             //
163             ++ "<li><pre>addition:</pre> <button onclick=\""
164             //
165             // start: embedded JavaScript
166             //
```

```
164                 //
165                 // start: embedded JavaScript
166                 //
167                 //   Clicking on the button opens a new window with
168                 //   a link which directs to the server's addition page
169                 //   with the two parameters stored in a URI-encoded,
170                 //   JSON-array that contain the values of the two text
171                 //   areas defined above.
172                 //
173                 ++ "  var param1 = document.getElementById('sum1').value;"
174                 ++ "  var param2 = document.getElementById('sum2').value;"
175                 ++ "  var json = '[ ' + param1 + ', ' + param2 + ']';"
176                 ++ "  var encodedURI = encodeURIComponent(json);"
177                 ++ "  var link = 'http://localhost:8080/addition.html?'"
178                 ++ "    + encodedURI;"
179                 ++ "  window.open(link);"
180                 //
181                 ++"\">caculate result</button></li>"
182                 ++ "</ul>"
183                 ++ "</body>"
184                 ++ "</html>";
185         },
186         //
187         // callback function for addition
188         //
189         "addition" : function(request) {
190             this@"countedCalls" = this@"countedCalls" + 1;
191             //
192             // The request contains the parameters in an url-encoded,
193             // json-style array.
194             //
195             var result = "undefined: parameters have to be integers";
196             try {
197                 var msg = urldecode(request);
198                 var parameters = eval(msg);
199                 result = ''(parameters@0 + parameters@1);
200             } catch (var error) {
201                // ups ...
202             }
203             //
204             return "<html>"
205                 ++ "<head>"
206                 ++ "<link rel=\"icon\""
207                 ++ " href=\"http://localhost:8080/favicon.png\""
208                 ++ " type=\"image/png\">"
209                 ++ "</head>"
210                 ++ "<body>"
211                 ++ "<h1><img src=\"favicon.png\">"
212                 ++ " Euclides Web Server</h1>"
213                 ++ "<p>The request is: " ++ request ++ "</p>"
214                 ++ "<p>The addition result is: " ++ ''result ++ ".</p>"
215                 ++ "<p>Return to "
216                 ++ "<a href=\"http://localhost:8080/index.html\">start</a>"
217                 ++ ".</p>"
218                 ++ "</body>"
219                 ++ "</html>";
220         }
221     };
```

```
222
223  var config = {
224     //
225     // The default port is 8080. To use another port
226     // set the corresponding attribute in the config:
227     //
228     // "port"  : 80,
229     //
230     // By default the web server prints some logging
231     // messages to standard out. To disable / modify
232     // this behavior simply set a new logging function:
233     //
234     // "log" : function(txt) { printErr(txt); printErr("\n"); },
235     //
236
237     "entry" : [
238         { "point"             : "/index.html",
239           "contentType"       : "text/html",
240           "contentFunction"   : server::"index"
241         },
242         { "point"             : "/addition.html",
243           "contentType"       : "text/html",
244           "contentFunction"   : server::"addition"
245         },
246         { "point"             : "/favicon.png",
247           "contentType"       : "image/png",
248           "contentFunction"   : function(request)
249              { return EUCLIDES_ICON_32x32_PNG; }
250         }
251     ]
252  };
253
254  //
255  // start web service:
256  //
257  publish(config);
```

The running web server is shown in Figure 4.1. The web server has been started on Microsoft Windows and the logging messages of the server can be seen in the Command Prompt Windows. The server is accessed via the Web browser "Opera". The "Opera" window in the background shows the "index.html" page. Setting two parameters on "index.html" and clicking on the "calculate result" button opens (via embedded JavaScript) a new window referencing the "addition.html" page and passing the query parameters set before. The server response can be seen in the "Opera" window in the foreground. It shows the calculation result.

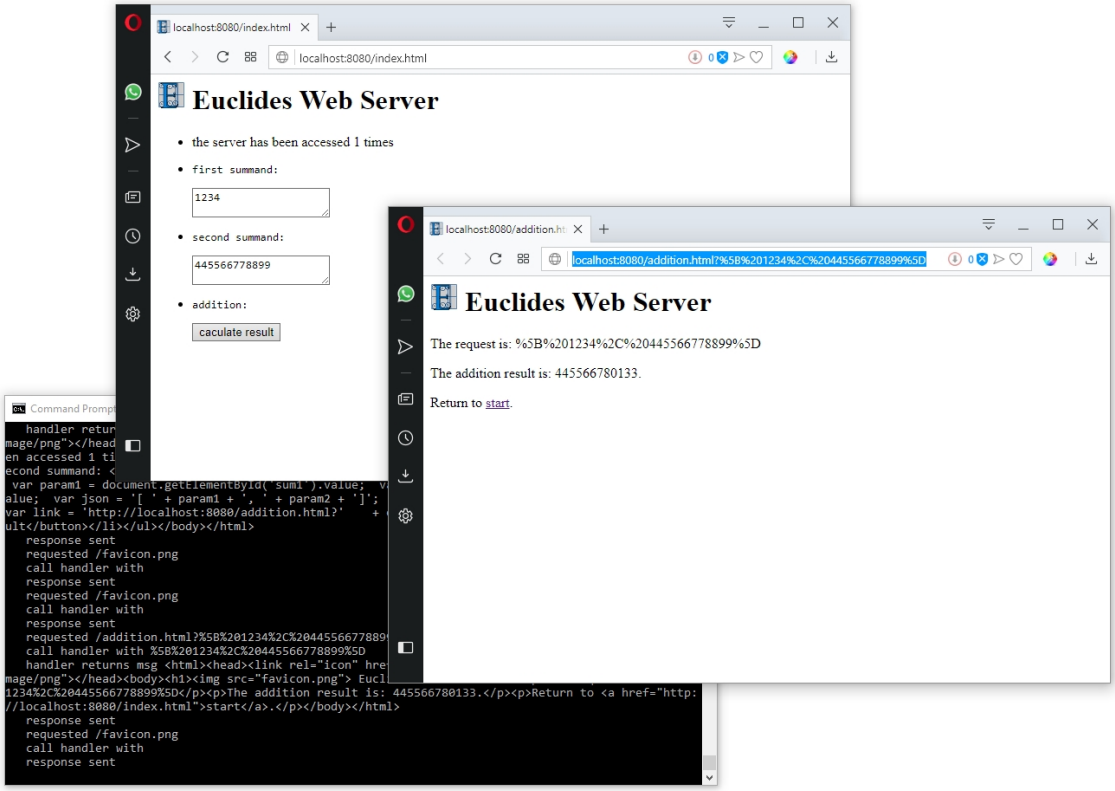


Figure 4.1: Euclides includes a small web server which can be used (among others) for micro-services.

4.2 Library: TEXT

Euclides Standard Library for text processing.

This library provides string-based functions:

- `isEmptyText`, `isBlankText` for empty and blank tests, and `isComposedText`, `isAlphaText`, `isNumericText`, `isAlphaNumericText`, `isAsciiPrintableText` for composition tests,
- `subText`, `subTextLeft`, `subTextRight`, and `abbreviateText` to access subparts of a string,
- `countText` to count occurances of strings in strings,
- `trimText`, `padTextLeft`, and `padTextRight` to remove or add whitespaces,
- `lowerCase` and `upperCase` for case conversion,
- `urlEncode` and `urlDecode` for encoding conversion,
- `splitText`, `joinText`, `repeatText`, and `replaceText` for common text modifications.

---

isemptytext:

`STRING → BOOLEAN`

The function `isEmptyText` returns **`true`**, if the passed string does not conain any characters; not even a space character. Oherwise, the function returns **`false`**:
This function is illustrated in the following souce code example:

Source Code 4.7 — Example.

```
1  import 'io'
2  import 'text'
3  //
4  // Some example strings that will be used for testing:
5  //
6  const examples = [ "", " ", "abc", " abc ",
7      "123", " 123 ", "abc123", "abc:123" ];
8  //
9  // The test routine calling the 'text' library function:
10 //
11 print("The 'isEmptyText' function of the 'text' library:");
12 for (var index : string in examples) {
13     print("'isEmptyText' applied to '" ++ ''string
14         ++ "' returns " ++ ''isEmptyText(string));
15 }
16 //
17 // The output (manually indented):
18 //
19 // The 'isEmptyText' function of the 'text' library:
20 // 'isEmptyText' applied to        '' returns TRUE
21 // 'isEmptyText' applied to       ' ' returns FALSE
22 // 'isEmptyText' applied to     'abc' returns FALSE
23 // 'isEmptyText' applied to   ' abc ' returns FALSE
24 // 'isEmptyText' applied to     '123' returns FALSE
25 // 'isEmptyText' applied to   ' 123 ' returns FALSE
26 // 'isEmptyText' applied to  'abc123' returns FALSE
27 // 'isEmptyText' applied to 'abc:123' returns FALSE
```

---

isblanktext:

STRING → BOOLEAN

The function `isBlankText` returns **`true`**, if the passed string consists only of whitespace characters. Oherwise, the function returns **`false`**.
This function is illustrated in the following souce code example:

Source Code 4.8 — Example.

```
import 'io'
import 'text'
//
// Some example strings that will be used for testing:
//
const examples = [ "", " ", "\t", "\n", "\r\n",
    "abc", " abc ", "abc123", "abc:123",
    "\u0020",    // unicode: SPACE
    "\u2000",    // unicode: EN QUAD
    "\u3000" ];  // unicode: IDEOGRAPHIC SPACE
//
// The test routine calling the 'text' library function:
//
print("The 'isBlankText' function of the 'text' library:");
for (var index : string in examples) {
    print("'isBlankText' applied to '" ++ ''string
        ++ "' returns " ++ ''isBlankText(string));
}
//
// The output (manually indented):
//
// The 'isBlankText' function of the 'text' library:
// 'isBlankText' applied to        '' returns TRUE
// 'isBlankText' applied to       ' ' returns TRUE
// 'isBlankText' applied to      '  ' returns TRUE
// 'isBlankText' applied to      '\n' returns TRUE
// 'isBlankText' applied to    '\r\n' returns TRUE
// 'isBlankText' applied to     'abc' returns FALSE
// 'isBlankText' applied to   ' abc ' returns FALSE
// 'isBlankText' applied to  'abc123' returns FALSE
// 'isBlankText' applied to 'abc:123' returns FALSE
// 'isBlankText' applied to       ' ' returns TRUE
// 'isBlankText' applied to       ' ' returns TRUE
// 'isBlankText' applied to       ' ' returns TRUE
```

iscomposedtext:

```
(STRING, STRING) → BOOLEAN
```

The function `isComposedText` returns **`true`**, if the first passed string consists only of characters contained in the second passed string. Oherwise, the function returns **`false`**. For commonly-used building blocks (such as alpha-numerical characters, etc.) specialized functions are provided: `isAlphaText`, `isNumericText`, `isAlphaNumericText`, `isAsciiPrintableText`.
These functions are illustrated in the following souce code example:

Source Code 4.9 — Example.

```
import 'io'
import 'text'
//
// Some example strings that will be used for testing:
//
const examples = [ "", " ", "abc", " abc ",
    "123", " 123 ", "aei", " aei ", "abc123", "abc:123",
    "$ 123.45", "EUR 678,99", "€ 678,99" ];
//
const vowels = "aeiouAEIOU";
//
// The test routines calling the 'text' library function:
//
for (var index : string in examples) {
    print("testing '" ++ ''string ++ "'");
    print("   is alpha?         " ++ ''isAlphaText(string));
    print("   is numeric?       " ++ ''isNumericText(string));
    print("   is alpha-numeric? " ++ ''isAlphaNumericText(string));
    print("   is ASCII?         " ++ ''isAsciiPrintableText(string));
    print("   vowels only?      " ++ ''isComposedText(string, vowels));
    print();
}
```

The output of the illustrating example can be summarized in a table:

input	is alpha?	is numeric?	is alpha-numeric?	is ASCII?	vowels only?
`""`	**true**	**true**	**true**	**true**	**true**
`" "`	**true**	**true**	**true**	**true**	**false**
`"abc"`	**true**	**false**	**true**	**true**	**false**
`" abc "`	**true**	**false**	**true**	**true**	**false**
`"123"`	**false**	**true**	**true**	**true**	**false**
`" 123 "`	**false**	**true**	**true**	**true**	**false**
`"aei"`	**true**	**false**	**true**	**true**	**true**
`" aei "`	**true**	**false**	**true**	**true**	**false**
`"abc123"`	**false**	**false**	**true**	**true**	**false**
`"abc:123"`	**false**	**false**	**false**	**true**	**false**
`"$ 123.45"`	**false**	**false**	**false**	**true**	**false**
`"EUR 678,99"`	**false**	**false**	**false**	**true**	**false**
`"€678,99"`	**false**	**false**	**false**	**false**	**false**

isalphatext:
`STRING → BOOLEAN`

This function is an shortcut for a common function call of `isComposedText`. See `isComposedText` for further details.

---

isnumerictext:
`STRING → BOOLEAN`

This function is an shortcut for a common function call of `isComposedText`. See `isComposedText` for further details.

---

isalphanumerictext:
`STRING → BOOLEAN`

This function is an shortcut for a common function call of `isComposedText`. See `isComposedText` for further details.

---

isasciiprintabletext:
`STRING → BOOLEAN`

This function is an shortcut for a common function call of `isComposedText`. See `isComposedText` for further details.

---

subtext:
`(STRING, INTEGER, INTEGER) → STRING`

The function `subText` takes a string-based text and two indices. It returns the part of the text which starts at the first index and ends at the second one.
An example of this function is:

Source Code 4.10 — Example.

```
1  import 'io'
2  import 'text'
3
4  print(        "----+----*----+----*----+----*----+----*");
5  print(subText("Once upon a midnight dreary,",          12, 20));
6  print(subText("while I pondered, weak and weary,",     18, 22));
7  print(subText("Over many a quaint and curious volume", 31, 37));
8  print(subText("of forgotten lore --",                  18, 20));
```

---

subtextleft:

`(STRING, INTEGER) → STRING`

The function `subTextLeft` takes some text and a length. It returns the left-most substring with the given length.
An example of this function is:

Source Code 4.11 — Example.

```
1  import 'io'
2  import 'text'
3
4  print(            "----+----*----+----*----+----*");
5  print(subTextLeft("While I nodded, nearly napping,", 5));
6  print(subTextLeft("suddenly there came a tapping,",  8));
7  print(subTextLeft("As of some one gently rapping,",  2));
8  print(subTextLeft("rapping at my chamber door.",     7));
```

---

subtextright:

`(STRING, INTEGER) → STRING`

The function `subTextRight` takes some text and a length. It returns the right-most substring with the given length.
An example of this function is:

Source Code 4.12 — Example.

```
1  import 'io'
2  import 'text'
3
4  print(              "*----+----*----+----*----+----");
5  print(subTextRight("'Tis some visitor,' I muttered,", 9));
6  print(subTextRight("'tapping at my chamber door -- ", 3));
7  print(subTextRight("Only this, and nothing more.'  ", 8));
```

---

abbreviatetext:

```
(STRING, INTEGER, INTEGER=50) → STRING
```

The function `abbreviateText` shortens the given text. If the passed text (first parameter) has a length smaller than or equal to five characters, the passed text is returned without modification. In the other cases the text is shortend to the given length (second parameter) by replacing inner text by "...". The placement of "..." can be controlled by the third parameter: it specifies the position in percent with the beginning at zero percent and the end at 100 percent; but the result keeps always the first and the last character.
Some example are listed in the source code:

Source Code 4.13 — Example.

```
1  import 'io'
2  import 'text'
3  //
4  // The function to abbreviate text can be used to
5  // shorten long file names:
6  //
7  print(abbreviateText(
8      "C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319\\SQL\\en",
9      20,   // shorten to max. 20 characters
10     33)); // place ... after 1/3 of the string
11
12 //
13 // The result is: 'C:\Wi...30319\SQL\en'
14 //
```

---

counttext:

```
(STRING, STRING) → INTEGER
```

The function `countText` counts the number of occurrences of the second argument in the first argument.
Some example are listed in the source code:

Source Code 4.14 — Example.

```
1  import 'io'
2  import 'text'
3
4  print(countText("ABBA",         "B" )); // output: >> 2 <<
5  print(countText("Ananas",       "na")); // output: >> 2 <<
6  print(countText("Ananas",       "a")); // output: >> 2 <<
7  print(countText("Antananarivo", "na")); // output: >> 2 <<
8  print(countText("Antananarivo", "a")); // output: >> 3 <<
9  print(countText("Tananarive",   "na")); // output: >> 2 <<
10 print(countText("Tananarive",   "a")); // output: >> 3 <<
```

---

trimtext:

```
STRING → STRING
```

The function `trimText` removes whitespaces from the start and from the end of text.

Source Code 4.15 — Example.

```
1  import 'io'
2  import 'text'
3
4  print(trimText("apple"));          // output: >> apple <<
5  print(trimText("  apple  "));      // output: >> apple <<
6  print(trimText(" \n apple \n "); // output: >> apple <<
```

---

padtextleft:

```
(STRING, INTEGER) → STRING
```

The function `padTextLeft` takes two arguments: a string and an integer. The returned string contains the text of the passed string and added whitespaces. The minimum length of the returned string is equal to the passed integer.

Source Code 4.16 — Example.

```
1  import 'io'
2  import 'text'
3
4  var sum = 0.0;
5  for(var index : value in [1.99, 2.99, 3.85, 12.95]) {
6      print(padTextLeft(''value, 8));
7      sum = sum + value;
8  }
9  print("--------");
10 print(padTextLeft(''sum, 8));
11
12 //
13 // output:
14 //
15 //     1.99
16 //     2.99
17 //     3.85
18 //    12.95
19 // --------
20 //    21.78
21 //
```

---

padtextright:

`(STRING, INTEGER) → STRING`

The function `padTextRight` is analogue to `padTextLeft`.

---

uppercase:

`STRING → STRING`

The function `upperCase` converts text into upper case text.

---

lowercase:

`STRING → STRING`

The function `lowerCase` converts text into lower case text.

---

urlencode:

`STRING → STRING`

The function `urlEncode` converts a string to the application/x-www-form-urlencoded MIME format.

---

urldecode:

`STRING → STRING`

The function `urlDecode` is reverses `urlEncode`.

---

splittext:

```
(STRING, STRING) → STRING
```

The function `splitText` takes two strings: the first argument contains some text, the second argument contains a separator. The result is an array of strings containing the remains of the splitting of the text at the separators.

Source Code 4.17 — Example.

```
import 'io'
import 'text'
//
// Use splitText to parse CSV data ...
//
var CSV = "1234;2345;346;;457;568;;;6798;7809";
for(var index : value in splitText(CSV, ";")) {
    print(''index ++ ". value: '" ++ value ++ "'");
}
//
// ... or to process text line by line:
//
var someTextWithLineBreaks = "abc\nefg\nhij\nklm";
for(var index : line in splitText(someTextWithLineBreaks, "\n")) {
    print(line);
}
```

jointext:

```
(ARRAY<,> STRING) → STRING
```

The function `joinText` takes two strings: an array and a string. It applies the string operator on each array element and joins the resulting strings to one string including the separator string between two consecutive array elements.

Source Code 4.18 — Example.

```
import 'io'
import 'text'

print(joinText([123, 234, 987], ","));
//
// output:
// >> 123,234,987 <<
//

print(joinText(["an apple", "a banana", "an orange"], " and "));
//
// output:
// >> an apple and a banana and an orange <<
//
```

repeattext:

```
(STRING, INTEGER) → STRING
```

The function `repeatText` simply repeats text several times.

---

replacetext:

```
(STRING, STRING, STRING) → STRING
```

The function `replaceText` takes three arguments: `text`, `pattern`, and `replacement`. It returns the string `text` with each occurrence of `pattern` replaced by `replacement`.

4.3 Library: MATH

Euclides Standard Library for mathematics.

This library provides commonly-used constants and often-used mathematical functions:

- $\pi, e, \pm\infty, \ldots$
- `isNaN`, `isInfinite`, `isFinite`
- `abs`, `ceil`, `floor`, `max`, `min`, `clamp`, `random`, `round`, `signum`
- `exp`, `log`, `log10`
- `sqrt`
- `toDegrees`, `toRadians`
- `sin`, `cos`, `tan`
- `sinh`, `cosh`, `tanh`
- `arcsin`, `arccos`, `arctan`

Most functions are defined for floating point arithmetics; i.e. for the data types float, vector, and matrix. If not stated otherwise, each operation on vectors and matrices is applied elementwisely.

---

pi:

`FLOAT`

The constant $\pi \approx 3.14159\ldots$
π is needed in many mathematical contexts. Its value can be approximated in many ways; for example by the Leibniz formula

$$\pi/4 = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

To illustrate the language Euclides (and not to approximate π), the series is implemented in Euclides:

Source Code 4.19 — Example.

```
1  import 'math'
2  import 'io'
3
4  var approximation = 0.0;
5  var error         = pi;
6
7  var k = 0;
8  while (error >= 0.0001) {
9      var term = (-1.0)^(k) / (float(k) * 2.0 + 1.0);
10     approximation = approximation + term;
11     error         = abs(4.0 * approximation - pi);
12     k = k + 1;
13     //
14     if (k % 10 == 0)
15         print("#" ++ ''k ++ " : " ++ ''(4.0 * approximation));
16 }
17 print("final error = " ++ ''error);
```

---

ee:

FLOAT

The constant $e \approx 2.71828\ldots$
Similar to π, the constant e is illustrated by an Euclides application approximating its value. The source code is based on the sum of the following infinite series (evaluated at $x = 0$):

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

Source Code 4.20 — Example.

```
import 'math'
import 'io'

var approximation = 0.0;
var error         = ee;

const fac = function(n) {
    var product = 1;
    var i = 1;
    while (i <= n) {
        product = product * i;
        i = i + 1;
    }
    return product;
};

var k = 0;
while (error >= 0.00001) {
    var term = 1.0 / float(fac(k));
    approximation = approximation + term;
    error         = abs(approximation - ee);
    k = k + 1;
    //
    print("#" ++ ''k ++ " : " ++ ''approximation);
}
print("final error = " ++ ''error);
```

---

nan:

FLOAT

The float “value” not-a-number according to IEEE-754 floating point.

---

posinf:

FLOAT

The float “value” positive infinity according to IEEE-754 floating point.

---

neginf:

FLOAT

The float "value" negative infinity according to IEEE-754 floating point. The values positive infinity, negative infinity and not-a-number can also be used in relational comparisons. The results are generated by the following source code:

Source Code 4.21 — Example.

```
import 'math'
import 'io'
//
// print output to standard out with left padding
//
const printFixed = function(output, length) {
    var text = ''output;
    var size = text@"size"();
    while (size < length) {
        printOut(" ");
        size = size + 1;
    }
    printOut(text);
};
//
// print a comparison table using the provided data and
// the given comparison function
//
const compare = function(data, comparison, msg, length) {
    print("comparison using '" ++ msg ++ "' :");
    //
    printFixed("", length);
    for (var _ : v0 in data) {
        printFixed(v0, length);
    }
    print("");
    //
    for(var i0 : v0 in data) {
        printFixed(v0, length);
        for (var i1 : v1 in data) {
            printFixed(comparison(v0, v1), 10);
        }
        print("");
    }
    print("");
};
//
// compare these values with <, >, ==, !=:
//
const values = [negInf, -1.0, -0.0, +0.0, 1.0, posInf, nan];
compare(values, function(x,y) return x  < y;,  "<", 10);
compare(values, function(x,y) return x  > y;,  ">", 10);
compare(values, function(x,y) return x == y;, "==", 10);
compare(values, function(x,y) return x != y;, "!=", 10);
```

isnan:

`FLOAT → BOOLEAN`

The function `isNan` returns true, if the passed argument is `NaN`. Otherwise it returns false.

---

isinfinite:

`FLOAT → BOOLEAN`

The function `isInfinite` returns true, if the passed argument is `posInf` or `negInf`. Otherwise it returns false.

---

isfinite:

`FLOAT → BOOLEAN`

The function `isFinite` returns true, if the passed argument is not `nan`, `posInf` or `negInf`. Otherwise it returns false.

---

abs:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `abs` returns the absolute value of the passed floating point / vector / matrix argument. In case of a vector or matrix argument, the function is applied elementwisely.

---

ceil:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `ceil` returns the smallest (closest to negative infinity) value that is greater than or equal to the passed argument and is equal to a mathematical integer.

---

floor:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `floor` returns the largest (closest to positive infinity) value that is less than or equal to the passed argument and is equal to a mathematical integer.

---

max:

```
(FLOAT, FLOAT) → FLOAT or (INTEGER, INTEGER) → INTEGER or
(VECTOR, VECTOR) → VECTOR or (MATRIX, MATRIX) → MATRIX
```

The function `max` returns the greater of two values.

---

min:

```
(FLOAT, FLOAT) → FLOAT or (INTEGER, INTEGER) → INTEGER or
(VECTOR, VECTOR) → VECTOR or (MATRIX, MATRIX) → MATRIX
```

The function `min` returns the smaller of two values.

---

clamp:

```
(FLOAT, FLOAT, FLOAT) → FLOAT or (INTEGER, INTEGER, INTEGER) → INTEGER or
(VECTOR, VECTOR, VECTOR) → VECTOR or (MATRIX, MATRIX, MATRIX) → MATRIX
```

The function `clamp` takes three arguments (`x`, `low`, `high`) and returns `x`, if `x` is within the bounds `low` and `high`. Otherwise, it returns `low`, if `x` is less than `low`; it returns `high`, if `x` is greater than `high`.

---

random:

```
( ) → FLOAT
```

The function `random` returns a value with a positive sign, greater than or equal to 0.0 and less than 1.0.

---

round:

```
FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX
```

The function `round` returns the closest value to the passed floating point argument, with ties rounding up.

---

signum:

```
FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX
```

The function `signum` returns 0.0 if the argument is zero, 1.0 if the argument is greater than zero, −1.0 if the argument is less than zero.

---

exp:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `exp` returns Euler's number e (see constant `EE`) raised to the power of the passed floating point value

---

log:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `log` returns the natural logarithm of the passed floating point value.

---

log10:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `log10` returns the base 10 logarithm of the passed floating point value.

---

sqrt:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `sqrt` returns the positive square root of the passed floating point value.

---

todegrees:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `toDegrees` converts an angle measured in radians to an approximately equivalent angle measured in degrees.

---

toradians:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `toRadians` converts an angle measured in degrees to an approximately equivalent angle measured in radians.

---

sin:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `sin` returns the trigonometric sine of an angle.

---

cos:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `cos` returns the trigonometric cosine of an angle.

---

tan:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `tan` returns the trigonometric tangent of an angle.

---

sinh:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `sinh` returns the hyperbolic sine of the passed floating point value.

---

cosh:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `sinh` returns the hyperbolic cosine of the passed floating point value.

---

tanh:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `tanh` returns the hyperbolic tangent of the passed floating point value.

---

arcsin:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `arcsin` returns the arc sine of the passed floating point argument; the returned angle is in the range of $-\frac{\pi}{2}$ through $\frac{\pi}{2}$.

---

arccos:

`FLOAT → FLOAT or VECTOR → VECTOR or MATRIX → MATRIX`

The function `arccos` returns the arc cosine of the passed floating point argument; the returned angle is in the range of 0.0 through π.

---

arctan:

```
(FLOAT, FLOAT=UNDEFINED) → FLOAT or
(VECTOR, VECTOR=UNDEFINED) → VECTOR or
(MATRIX, MATRIX=UNDEFINED) → MATRIX
```

The function `arctan` calculates the arc tangent. If only one argument is passed, it returns the arc tangent of the passed floating point value; in this case the returned angle is in the range of $-\frac{\pi}{2}$ through $\frac{\pi}{2}$. If two arguments `x` and `y` are passed the function returns the angle θ from the conversion of rectangular coordinates (x, y) to polar coordinates (r, θ). This conversion step is illustrated in the following souce code example:

Source Code 4.22 — Example.

```
1  import 'math'
2  import 'io'
3
4  var radius = 2.5;
5  var theta  = -pi;
6  var offset = pi / 17.0;
7
8  while (theta < pi) {
9      print("radius = " ++ ''radius ++ " ; "
10         ++ "theta = " ++ ''theta ++ " ("
11         ++ ''toDegrees(theta) ++ "\u00B0)");
12     //
13     // convert to cartesian coordinates:
14     //
15     var x = radius * cos(theta);
16     var y = radius * sin(theta);
17     print("  ->  x = " ++ ''x ++ " ; y = " ++ ''y);
18     //
19     // convert (back) to polar coordinates:
20     //
21     var r = sqrt(x^2 + y^2);
22     var t = arctan(x, y);
23     print("  ->  r = " ++ ''r ++ " ; t = " ++ ''t);
24     //
25     // error analysis:
26     //
27     print("  ->  | t - theta | = " ++ ''abs(t-theta));
28     //
29     theta = theta + offset;
30     print();
31 }
```

4.4 Library: SEQUENCE

Euclides Standard Library for array handling and interpolation.

This library provides generators and modifiers for arrays:

- `copy`, `range`, `nDimensional`, `cartesianProduct`
- `sortArray`, `reverseArray`, `transposeArray`, `resizeArray`
- `map`, `filter`

---

copy:

`(TEMPLATE`$_A$`, INTEGER, BOOLEAN=`**`TRUE`**`) → ARRAY<TEMPLATE`$_A$`>`

The function `copy` takes three arguments (`original`, `repetitions`, and `deepCopy`). It returns an array containing `repetitions` copies of `original`. The first argument can have arbitrary type and value. The second argument must be a semi-positive integer. It determines the size of the resulting array. The third argument is an optional boolean value. If the thrid argument (`deepCopy`) is **`TRUE`** (default), then the copies will be created using the deep copy operator `$`. Otherwise (**`false`**) the copies will be simple assignment copies. The difference between the two copy operations is explained in sections on arrays and objects in the manual.

Source Code 4.23 — Example.

```
1  import 'sequence'
2  import 'io'
3
4  print(copy(42.0, 8));
5  print(copy([ ], 8));
6  print(copy([ ], 8,  true));
7  print(copy([ ], 8, false));
```

The exmaple illustrates the difference between the two copy options. In the lines 5 (by default) and 6 (explicit) the deep copy operation is used resulting in `[[], [], [], [], [], [], [], []]`. In line 7 the reference copy is used which results in `[[], [ ... ], [ ... ], [ ... ], [ ... ], [ ... ], [ ... ], [ ... ]]` using the "already-printed" placeholder `[ ... ]` to avoid (possibly cyclic) references.

---

range:

```
(FLOAT, FLOAT, INTEGER) → ARRAY<FLOAT>
```

or

```
(VECTOR, VECTOR, INTEGER) → ARRAY<VECTOR>
```

or

```
(MATRIX, MATRIX, INTEGER) → ARRAY<MATRIX>
```

The function `range` returns an array interpolating the passed arguments. It takes three arguments (`start`, `end`, and `steps`). The first argument and the second one must be of type `FLOAT`, `VECTOR`, or `MATRIX`. Furthermore, both arguments must have the same type. The thrid argument (`steps`) has to be a positive integer. It describes the number of interpolation steps between `start` and `end`.

Source Code 4.24 — Example.

```
1  import 'sequence'
2  import 'io'
3
4  const color0 = <0.000, 1.000, 0.000>; // 00ff00 : green
5  const color1 = <0.864, 1.000, 0.000>; // dcff00 :
6  const color2 = <0.900, 1.000, 0.000>; // e6ff00 :
7  const color3 = <0.986, 1.000, 0.000>; // fbff00 :
8  const color4 = <1.000, 1.000, 0.000>; // ffff00 : yellow
9  const color5 = <1.000, 0.911, 0.000>; // ffe800 :
10 const color6 = <1.000, 0.900, 0.000>; // ffe600 :
11 const color7 = <1.000, 0.036, 0.000>; // ff0900 :
12 const color8 = <1.000, 0.000, 0.000>; // ff0000 : red
13
14 const colorMap =
15     range(color0, color1, 36) ++
16     range(color2, color3, 12) ++
17     range(color4, color5, 12) ++
18     range(color6, color7, 36);
19
20 for (var index:color in colorMap) {
21     print("color #" ++ ''index ++ ": " ++ ''color);
22 }
```

ndimensional:

```
(ARRAY<INTEGER>,
 ARRAY<INTEGER> → TEMPLATE_A=FUNCTION (indices){ RETURN UNDEFINED; })
→ ARRAY<TEMPLATE_A>
```

The function `nDimensional` creates a multi-dimensional array by two parameters: `dimensions` and `values`. The parameter `dimensions` defines the corresponding array size in each dimension; the parameter `values` is a function, which will be called with an array of indices and whose results are stored in the n-dimensional array. By default, the `values` function to be used always returns **`undefined`**.

Source Code 4.25 — Example.

```
1  import 'sequence'
2  import 'io'
3
4  var arrayNDim1 = nDimensional([ ]);
5  print(arrayNDim1);            // output: >> UNDEFINED <<
6
7  var arrayNDim2 = nDimensional([0]);
8  print(arrayNDim2);            // output: >> [] <<
9
10 var arrayNDim3 = nDimensional([2]);
11 print(arrayNDim3);           // output: >> [UNDEFINED, UNDEFINED] <<
12
13 var arrayNDim4 = nDimensional([2, 3]);
14 print(arrayNDim4);           // output: >>
15                              //  [[UNDEFINED, UNDEFINED, UNDEFINED],
16                              //   [UNDEFINED, UNDEFINED, UNDEFINED]] <<
17
18 var arrayNDim5 = nDimensional([2, 3, 4],
19     function(indices) { return "f(" ++ ''indices ++ ")"; });
20
21 print(arrayNDim5);
22 // output (with additional line breaks and spaces for readability):
23 // >>
24 // [[[f([0, 0, 0]), f([0, 0, 1]), f([0, 0, 2]), f([0, 0, 3])],
25 //   [f([0, 1, 0]), f([0, 1, 1]), f([0, 1, 2]), f([0, 1, 3])],
26 //   [f([0, 2, 0]), f([0, 2, 1]), f([0, 2, 2]), f([0, 2, 3])]],
27 //
28 //  [[f([1, 0, 0]), f([1, 0, 1]), f([1, 0, 2]), f([1, 0, 3])],
29 //   [f([1, 1, 0]), f([1, 1, 1]), f([1, 1, 2]), f([1, 1, 3])],
30 //   [f([1, 2, 0]), f([1, 2, 1]), f([1, 2, 2]), f([1, 2, 3])]]]
31 // <<
```

cartesianproduct:

(ARRAY<TEMPLATE$_A$>, ARRAY<TEMPLATE$_A$>, BOOLEAN=**TRUE**) → ARRAY<ARRAY<TEMPLATE$_A$>>

The function `cartesianProduct` takes two arrays (`arrayA` and `arrayB`) and returns an array of arrays with all combinations of values which are elements of the first and the second array. Example: The expression `cartesianProduct([1, 2], [3, 4])` evaluates to an array containing all combinations of elements of the first array with elements of the second one: `[[1, 3], [1, 4], [2, 3], [2,4]]`.

---

sortarray:

(ARRAY<TEMPLATE$_A$>,
(TEMPLATE$_A$, TEMPLATE$_A$) → BOOLEAN=**FUNCTION**(x, y){ **RETURN** x<y; })
→ ARRAY<TEMPLATE$_A$>

The `sortArray` function sorts an array in-place and returns its reference. The array elements are compared via a binary relation function with the "natural" order as default implementation. The current implementation is not stable; i.e. the relative order of elements with equal values is not maintained.

---

reversearray:

ARRAY<TEMPLATE$_A$> → ARRAY<TEMPLATE$_A$>

The `reverseArray` function reverses an array in-place and returns its reference.

---

transposearray:

(ARRAY<ARRAY<TEMPLATE$_A$>>) → ARRAY<ARRAY<TEMPLATE$_A$>>

The function `transposeArray` takes one argument. The argument is an array of arrays, which will be transposed (similar to matrix transposition). The input 2d-array will be interpreted as a rectangular layout; out of range elements will be added in the result automatically with the value **`undefined`**.

---

resizearray:

(ARRAY<TEMPLATE$_A$>, INTEGER, TEMPLATE$_A$=**UNDEFINED**, BOOLEAN=**TRUE**
) → ARRAY<TEMPLATE$_A$>

The function `resizeArray` takes four parameters – namely, `inputArray`, `newSize`, `value`, and `deepCopy`. Its result is a new array of size `newSize` containing copies of the array elements from the input array. If the new array has more elements, it will be filled with copies of `value` (by default **`undefined`**). The boolean flag `deepCopy` is optional with default value **`true`**.

---

map:

`(ARRAY<TEMPLATE`$_A$`>, TEMPLATE`$_A$ `→ TEMPLATE`$_B$`) → ARRAY<TEMPLATE`$_B$`>`

The function `map` applies a function to every element of an array. The function takes two arguments: the first argument is an array. The second argument is a function which can be applied to the elements of the given array. The returned values of the function are stored in a new array, which is returned by `map`.

Source Code 4.26 — Example.

```
1  import 'sequence'
2  import 'io'
3  import 'math'
4
5  const square = function(x) { return x*x; };
6  const isEven = function(x) { return integer(x) % 2 == 0; };
7
8  var numbers           = range(0.0, 100.0, 100);
9  var squareNumbers     = map(numbers, square);
10 var evenSquareNumbers = filter(squareNumbers, isEven);
11
12 print(numbers);
13 print(squareNumbers);
14 print(evenSquareNumbers);
```

---

filter:

`(ARRAY<TEMPLATE`$_A$`>, TEMPLATE`$_A$ `→ BOOLEAN) → ARRAY<TEMPLATE`$_A$`>`

The function `filter` applies a filter-function to every element of an array. The function takes two arguments: the first argument is an array. The second argument is a function which can be applied to the elements of the given array and which returns **`true`** or **`false`**. If the function value is **`true`**, the array element to which the filter has been applied is stored in a new array; otherwise, it is skipped.

4.5 Library: BLAS

Euclides Standard Library for basic linear algebra subprograms.

This library provides linear algebra functions:

- `crossProd`
- `normalize`, `norm1`, `norm2`, `normInf`, `orthoNormal`
- `identity`, `hilbert`
- `translation3d`
- `rotation3dX`, `rotation3dY`, `rotation3dZ`
- `transformationAxisAngle`
- `transformationEulerAngles`
- `transformationQuaternion`

This library imports the library *math*.

---

crossprod:
`(VECTOR, VECTOR) → VECTOR`

The function `crossProd` returns the cross product of two 3-dimensional vectors. In detail, with the input vectors a and b the function calculates $a \times b$:

$$a \times b = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} \times \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_1 \cdot b_2 - a_2 \cdot b_1 \\ a_2 \cdot b_0 - a_0 \cdot b_2 \\ a_0 \cdot b_1 - a_1 \cdot b_0 \end{pmatrix}.$$

Source Code 4.27 — Example.

```
1  import 'blas'
2  import 'io'
3
4  const some3DVectors = [ < 0.0,  0.0,  0.0>, < 1.0,  1.0,  1.0>,
5      < 1.0,  0.0,  0.0>, < 0.0,  1.0,  0.0>, < 0.0,  0.0,  1.0>,
6      < 1.0,  2.0,  3.0>, < 4.0,  5.0,  6.0>, < 7.0,  8.0,  9.0>,
7      <-1.0,  2.0, -3.0>, < 4.0, -5.0,  6.0>, <-7.0,  8.0,  9.0>];
8
9  for (var indexA:vectorA in some3DVectors) {
10     for (var indexB:vectorB in some3DVectors) {
11         print("a = " ++ ''vectorA);
12         print("b = " ++ ''vectorB);
13         print("   crossProd(a, b)        = " ++ ''crossProd(vectorA,
               vectorB));
14         print("   dot product of a and b = " ++ ''(vectorA * vectorB));
15     }
16 }
```

normalize:

VECTOR → VECTOR

The function `normalize` returns a scaled, normalized vector; i.e. the vector with Euclidean unit length.

Source Code 4.28 — Example.

```
import 'blas'
import 'io'

var value = 1.0;
var dimension = 5;

while (value < 10000.0) {
    var vector = <| dimension |>;
    for (var index : component in vector)
        vector@index = 1.0 / value;
    print(vector);
    //
    print("   " ++ ''normalize(vector));
    print("   norm1 = " ++ ''norm1(vector)
        ++ ", norm2 = " ++ ''norm2(vector)
        ++ ", normInf = " ++ ''normInf(vector));
    //
    value = value + 100.0;
}
//
print("normalize zero vector:");
print(normalize(<0.0, 0.0, 0.0, 0.0>));
```

This example shows that normalizing a zero vector results in a NaN-vector.

---

norm1:

VECTOR → FLOAT

The function `norm1` returns the 1-norm of the passed input vector – the so-called absolute value sum.

---

norm2:

VECTOR → FLOAT

The function `norm2` returns the 2-norm of the passed input vector – the so-called Euclidean norm.

---

norminf:

`VECTOR → FLOAT`

The function `normInf` returns the ∞-norm of the passed input vector – the so-called maximum norm.

---

orthonormal:

`VECTOR → VECTOR`

The function `orthonormal` returns a vector which is orthonormal to the given 3-dimensional vector. This function should not be used with a zero vector (as illustrated in the following example):

Source Code 4.29 — Example.

```
1  import 'blas'
2  import 'io'
3
4  const demo = function(vec) {
5      var onVec = orthonormal(vec);
6      print(vec);
7      print("   orthonormal      = " ++ ''onVec);
8      print("   is ortho  (= 0)? = " ++ ''(vec * onVec));
9      print("   is normal (= 1)? = " ++ ''norm2(onVec));
10 };
11
12 demo(<0.0, 0.0, 0.0>);
13 demo(<1.0, 0.0, 0.0>);
14 demo(<0.0, 1.0, 0.0>);
15 demo(<0.0, 0.0, 1.0>);
16 demo(<1.0, 1.0, 1.0>);
17 demo(<2.0, 3.0, 4.0>);
```

---

identity:

`(INTEGER, INTEGER) → MATRIX`

The function `identity` returns an identity matrix; i.e. a matrix of size (`rows`, `columns`) – first and second argument – with its entries at position (i, j) being 1.0, if i equals j, and 0.0 otherwise.

---

hilbert:

`(INTEGER, INTEGER) → MATRIX`

The function `hilbert` returns a Hilbert matrix; i.e. a matrix of size (`rows`, `columns`) – first and second argument – with entries $1 + i + j$ at position (i, j).

---

translation3d:

`VECTOR → MATRIX`

The function `translation3D` returns the 4×4 transformation matrix of a 3D translation defined by the passed argument **`vector`** in homogenous coordinates (x, y, z, w).

Source Code 4.30 — Example.

```
1  import 'blas'
2  import 'io'
3
4  var point = < 1.2,  3.4, 5.6, 1.0>;
5  print(point);                 // output: >> <1.2, 3.4, 5.6, 1.0> <<
6
7  var direction  = <10.0, 30.0, 0.0, 0.0>;
8  var movedPoint = translation3d(direction) * point;
9  print(movedPoint);            // output: >> <11.2, 33.4, 5.6, 1.0> <<
```

---

rotation3dx:

`FLOAT → MATRIX`

The function `rotation3DX` returns the 4×4 transformation matrix of a 3D rotation around the x-axis. The given argument `angle` has to be in radians. With c and s being `cos(angle)` and `sin(angle)` respectively, the returned matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

---

rotation3dy:

`FLOAT → MATRIX`

The function `rotation3DY` returns the 4×4 transformation matrix of a 3D rotation around the y-axis. The given argument `angle` has to be in radians. With c and s being `cos(angle)` and `sin(angle)` respectively, the returned matrix is

$$\begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

---

rotation3dz:

```
FLOAT → MATRIX
```

The function `rotation3DZ` returns the 4×4 transformation matrix of a 3D rotation around the z-axis. The given argument `angle` has to be in radians. With c and s being `cos(angle)` and `sin(angle)` respectively, the returned matrix is

$$\begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

---

transformationaxisangle:

```
(VECTOR, FLOAT) → MATRIX
```

The function `transformationAxisAngle` returns the 4×4 transformation matrix of a 3D rotation around the `axis` (the first argument in homogeneous coordinates (x, y, z, w) with $w = 0$). The given rotation parameter `angle` (the second argument) has to be in radians.

---

transformationeulerangles:

```
(FLOAT, FLOAT, FLOAT) → MATRIX
```

The function `transformationEulerAngles` returns the 4×4 transformation matrix representing the rotation by the given Euler angle (in radians) using the NASA standard aeroplane conventions applying `heading` (the first parameter), `attitude` (the second parameter), `bank` (the third parameter) in this order.

---

transformationquaternion:

```
(FLOAT, FLOAT, FLOAT, FLOAT) → MATRIX
```

The function `transformationQuaternion` returns the 4×4 transformation matrix representing the rotation by the given quaternion $(qw + I \cdot qx + J \cdot qy + K \cdot qz)$, respectively by the parameters `qw`, `qx`, `qy`, and `qz`.

4.6 Library: COLOR

Euclides Standard Library for color handling. This library provides often used color maps in visualizations. All maps consist of an array with 101 colors. As a consequence, any visualization should map the input values to a corresponding scale that can be discretized to an integer index between 0 and 100. Each color is represented by three RGB values within 0.0 and 1.0 stored in a three-dimensional vector.

This library imports the libraries *math* and *sequence*.

---

colornames:

```
OBJECT{
“beige” : VECTOR, “black” : VECTOR, “blue” : VECTOR, “brown” : VECTOR,
“coral” : VECTOR, “cyan” : VECTOR, “grey” : VECTOR, “green” : VECTOR,
“lavender” : VECTOR, “lime” : VECTOR, “magenta” : VECTOR, “maroon” : VECTOR,
“mint” : VECTOR, “navy” : VECTOR, “olive” : VECTOR, “orange” : VECTOR,
“pink” : VECTOR, “red” : VECTOR, “teal” : VECTOR, “white” : VECTOR,
“yellow” : VECTOR, “scheme” : ARRAY<STRING>}
```

This constant contains all color names of this library. Additionally to the color names, it defines an array of color names called `scheme`, which provides a color scheme based on / similar to Kenneth L. Kelly’s 22 colors of maximum contrast (without black and white).

---

colormaps:

```
OBJECT{
“alarmScale” : ARRAY<VECTOR>,
“blackBodyRadiation” : ARRAY<VECTOR>,
“blueRedScale” : ARRAY<VECTOR>,
“clippedHueScale” : ARRAY<VECTOR>,
“hueScale” : ARRAY<VECTOR>,
“isoluminantGrayBlueScale” : ARRAY<VECTOR>,
“isoluminantGrayGreenScale” : ARRAY<VECTOR>,
“isoluminantGrayRedScale” : ARRAY<VECTOR>,
“isoluminantGreenRedScale” : ARRAY<VECTOR>,
“isoluminantYellowBlueScale” : ARRAY<VECTOR>,
“luminanceGrayScale” : ARRAY<VECTOR>,
“saturationGreenScale” : ARRAY<VECTOR>,
“saturationRedScale” : ARRAY<VECTOR>}
```

This constant contains all color maps of this library.

---

colorconversion:

```
OBJECT{
"rgb2hex" : VECTOR → VECTOR,
"hex2rgb" : VECTOR → VECTOR,
"rgb2cmy" : VECTOR → VECTOR,
"cmy2rgb" : VECTOR → VECTOR,
"rgb2cmyk" : VECTOR → VECTOR,
"cmyk2rgb" : VECTOR → VECTOR,
"rgb2hsv" : VECTOR → VECTOR,
"hsv2rgb" : VECTOR → VECTOR,
"rgb2yuv" : VECTOR → VECTOR,
"yuv2rgb" : VECTOR → VECTOR,
"rgb2gray" : VECTOR → FLOAT}
```

This constant contains conversion functions to represent colors in different color spaces:

- `rgb2hex` and `hex2rgb` convert an rgb color vector into an rgb color string; e.g. the blue color `<0.0, 0.5, 0.8>` is converted to the string `0080CC`.
- `rgb2cmy` and `cmy2rgb` convert between rgb and cmy color spaces. WARNING: This routine does not use any color profile. Therefore, its usage is rather limited.
- `rgb2cmyk` and `cmyk2rgb` convert between rgb and cmyk color spaces. WARNING: This routine does not use any color profile. Therefore, its usage is rather limited.
- `rgb2hsv` and `hsv2rgb` convert between rgb and hsv color spaces. The hsv parameter range is $[0-360]\times[0,1]\times[0,1]$.
- `rgb2yuv` and `yuv2rgb` convert between rgb and yuv color spaces. The yuv parameter range is not limited.
- `rgb2gray` converts a color (r,g,b) into a gray value according to the linear formular $0.299\cdot r+0.587\cdot g+0.114\cdot b$.

4.7 Library: IMAGE

Euclides Standard Library for image handling.

This library provides a constructor function to create an empty (black) image object with member function to set pixels and get pixels. Furthermore, the image object has member functions to export the image as a file in JPG and PNG format. The file can be written to harddisk via the `fileOut` routine of the *io* library. In order to import an image, the reverse function (`fileIn` by the *io* library) and the `imageFromFile` function of this library can be used. The following source code illustrates this pipeline:

Source Code 4.31 — Example.

```
import 'io'
import 'image'

//
// use io library's fileIn function to load an image file
//
var inputFile = fileIn("../data/test_image.jpeg");

//
// convert bytes of binary input file to an image
//
var testImage = imageFromFile(inputFile);

//
// convert all pixels (r, g, b) to grayscale 0.21r + 0.72g + 0.07b
//
const luminosity = < 0.21, 0.72, 0.07 >;
for (var index : pixel in testImage@"pixels") {
    //
    // this loop is an alternative to using the image
    // functions testImage@"getPixel" and testImage@"setPixel"
    //
    var gray = luminosity * pixel;
    testImage@"pixels"@index = < gray, gray, gray >;
}

//
// convert image to jpg bytes and set output file name
//
var outputFile = testImage@"toFileJPG"();
outputFile@"name" = "./test_image_grayscale.jpg";

//
// write file
//
fileOut(outputFile);
```

image:

```
(INTEGER, INTEGER) → OBJECT{
 “width” : INTEGER,
 “height” : INTEGER,
 “pixels” : ARRAY<VECTOR>,
 “getPixel” : (INTEGER, INTEGER) → VECTOR,
 “setPixel” : (INTEGER, INTEGER, VECTOR) → ( ),
 “toString” : ( ) → STRING,
 “toFileJPG” : ( ) → OBJECT{},
 “toFilePNG” : ( ) → OBJECT{} }
```

The image function is the constructor of a data structure for image handling. The returned `image` data structure provides methods to get and set pixels `image@"getPixel"(x, y)`, `image@"setPixel"(x, y, color)`. The image's size and the pixels are stored in `image@"width"`, `image@"height"`, and `image@"pixels"`. Each pixel is an rgb-encoded color $[0,1]^3$ stored in a 3d-vector.
Furthermore, the object provides conversion functions to represent the image data as a string (having only the size information) `toString()` or as a file (including pixel information) `toFileJPG()`, `toFilePNG()` (see *io* library for file details).

---

imagefromfile:

```
OBJECT{ file (see library 'io') } → OBJECT{
 “width” : INTEGER,
 “height” : INTEGER,
 “pixels” : ARRAY<VECTOR>,
 “getPixel” : (INTEGER, INTEGER) → VECTOR,
 “setPixel” : (INTEGER, INTEGER, VECTOR) → ( ),
 “toString” : ( ) → STRING,
 “toFileJPG” : ( ) → OBJECT{},
 “toFilePNG” : ( ) → OBJECT{} }
```

The function `imageFromFile` interprets a file data structure as an image and returns the `image` data structure (see `image` function). This data structure provides methods to get and set pixels `image@"getPixel"(x, y)`, `image@"setPixel"(x, y, color)`. The image's size and the pixels are stored in `image@"width"`, `image@"height"`, and `image@"pixels"`. Each pixel is an rgb-encoded color $[0,1]^3$ stored in a 3d-vector.
Furthermore, the object provides conversion functions to represent the image data as a string (having only the size information) `toString()` or as a file (including pixel information) `toFileJPG()`, `toFilePNG()` (see *io* library for file details).

4.8 Library: MATERIAL

Euclides Standard Library for material handling in the context of computer graphics modeling.

The material handling is minimalistic. It consists of three rgb colors for `ambient`, `diffuse`, `specular` attributes, a float value between 0.0 and 1.0 defining `shininess` and a texture defined by an image:

- `Material` is a constructor function defining a material object,
- `MaterialLibrary` is a collection of predefined commonly-used materials.

This library imports the libraries *image* and *math*.

---

material:

```
(VECTOR=<0.0, 0.0, 0.0>, VECTOR=<0.0, 0.0, 0.0>,
 VECTOR=<0.0, 0.0, 0.0>, FLOAT=0.0,
 OBJECT{ image (see library 'image') }=UNDEFINED) → OBJECT{
 "ambient" : VECTOR,
 "diffuse" : VECTOR,
 "specular" : VECTOR,
 "shininess" : FLOAT,
 "texture" : OBJECT{ image (see library 'image') }}
```

The function `Material` creates an material object. The function takes five optional arguments: `ambient`, `diffuse`, `specular`, `shininess`, and `texture`:

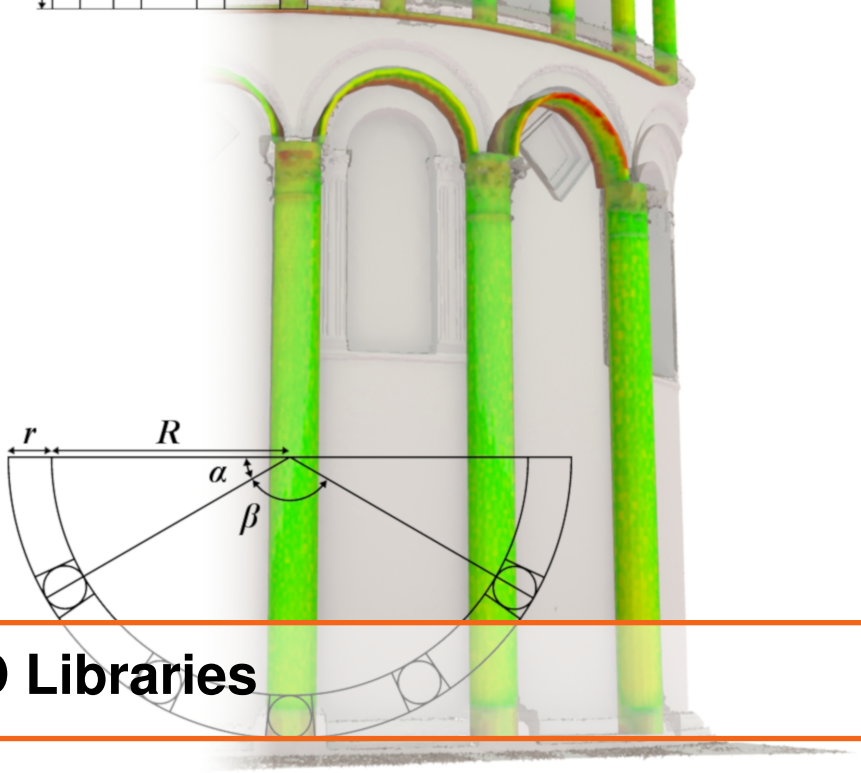
- `ambient`, `diffuse`, and `specular` are 3d color vectors of rgb values (with default values `< 0.0, 0.0, 0.0 >` i.e. black),
- `shininess` is a float value with default `0.0`,
- `texture` is an image (by default **`undefined`**).

---

materiallibrary:

```
OBJECT{
 “blackPlastic” : OBJECT{ material },
 “blackRubber” : OBJECT{ material },
 “bluePlastic” : OBJECT{ material },
 “blueRubber” : OBJECT{ material },
 “brass” : OBJECT{ material },
 “bronze” : OBJECT{ material },
 “chrome” : OBJECT{ material },
 “copper” : OBJECT{ material },
 “cyanPlastic” : OBJECT{ material },
 “cyanRubber” : OBJECT{ material },
 “emerald” : OBJECT{ material },
 “gold” : OBJECT{ material },
 “greenPlastic” : OBJECT{ material },
 “greenRubber” : OBJECT{ material },
 “jade” : OBJECT{ material },
 “magentaPlastic” : OBJECT{ material },
 “magentaRubber” : OBJECT{ material },
 “obsidian” : OBJECT{ material },
 “pearl” : OBJECT{ material },
 “redPlastic” : OBJECT{ material },
 “redRubber” : OBJECT{ material },
 “ruby” : OBJECT{ material },
 “silver” : OBJECT{ material },
 “turquoise” : OBJECT{ material },
 “whitePlastic” : OBJECT{ material },
 “whiteRubber” : OBJECT{ material },
 “yellowPlastic” : OBJECT{ material },
 “yellowRubber” : OBJECT{ material }}
```

Material definitions from the OpenGL demos.
(C) Silicon Graphics, Inc., 1994, Mark J. Kilgard.



5. Standard CAD Libraries

The EUCLIDES framework is designed for generative modeling in the field of computer-aided (geometric) design. As a consequence, it includes a set of CAD libraries:

cadPolyFace The *cadPolyFace* library implements a low-level container to represent geometry.

cadPolyFaceTools Simple routines to modify *cadPolyFace* instances are collected in the *cadPolyFaceTools* library.

cadPolyFaceFactory Example geometry based on *cadPolyFace* can be generated using the factory methods in the *cadPolyFaceFactory* library.

cadAutomation The *cadAutomation* library provides functionality for CAD-based design automation.

5.1 Library: CADPOLYFACE

Euclides Standard Library for Computer-Aided Design: data structure polyface.

This library provides a constructor function to generate a polyface. All polyfaces consist of vertices, faces, normals, colors, and uv-coordinates. This data structure is used as a low-level container to represent geometry. Its purpose is low-level data exchange – it is not designed to be used for modeling purposes. As a consequence, it mainly provides functions for geometry exports to OBJ, STL, X3D, and SVG. The returned file is a zipped archive (containing e.g. in case of zippedOBJ-export an OBJ, an MTL and image files) and can be written to harddisk via the `fileOut` routine of the *io* library.

This library imports the libraries *blas*, *color*, *material*, *math*, and *sequence*.

---

cadpolyface:

```
(VECTOR, ARRAY<ARRAY<INTEGER>>,
 VECTOR=UNDEFINED, VECTOR=UNDEFINED,
 VECTOR=UNDEFINED, VECTOR=UNDEFINED,
 OBJECT{ (see 'material' in library 'material') }=MaterialLibrary@"chrome",
 STRING="unknown") → OBJECT{
    "name" : STRING,
    "polyfaces" : ARRAY<OBJECT{
       "vertices" : VECTOR,
       "normals" : VECTOR,
       "colors" : VECTOR,
       "textures" : VECTOR,
       "attributes" : VECTOR,
       "faces" : ARRAY<ARRAY<INTEGER>>,
       "transformation" : MATRIX,
       "material" : OBJECT{ (see 'material' in library 'material')},
       "name" : STRING} >,
    "toString" : ( ) → STRING,
    "toFileZippedOBJ" : ( ) → OBJECT{ (see 'file' in library 'io') },
    "toFileZippedSTL" : ( ) → OBJECT{ (see 'file' in library 'io') },
    "toFileZippedSVG" : (MATRIX, MATRIX,
       BOOLEAN=FALSE) → OBJECT{ (see 'file' in library 'io') },
    "toFileZippedX3D" : ( ) → OBJECT{ (see 'file' in library 'io') }}
```

The poly face function is the constructor of a data structure for low level geometry representation. The returned `cadPolyFace` data structure provides methods to export geometry data as a string (having only the size information) or as a file (OBJ, STL, SVG, X3D; see `io` library for details).

---

cadpolyfacefromfile:

```
OBJECT{(see 'file' in library 'io')} → ( )
```

Import the geometry of a zipped or non-zipped file. Supported file formats are binary STL and ascii OBJ.

The following example illustrates the usage of the *cadPolyFace* library:

Source Code 5.1 — The library for polygonal faces in CAD.

```
1  import 'blas'
2  import 'cadpolyface'
3  import 'io'
4  import 'sequence'
5
6  //
7  // This example 3D-plots a function.
8  // The function f(u, v) is a so-called "monkey saddle". The surface
9  // normal is represented by a function n(u, v) and its mean
10 // curvature is implemented in mean(u, v):
11 //
12 const f = function(u, v) {
13     return < u, v, u^3 - 3.0*u*v^2 >;
14 };
15
16 const n = function(u, v) {
17     return normalize(< -3.0*u^2 + 3.0*v^2, 6.0*u*v, 1.0 >);
18 };
19
20 const mean = function(u, v) {
21     return (-27.0*u^5 + 54.0*u^3*v^2 + 81.0*u*v^4)
22         / (1.0 + 9.0*u^4 + 18.0*u^2*v^2 + 9.0*v^4)^1.5;
23 };
24
25 //
26 // The surface plot consists of a regular grid. The grid data
27 // comprehends vertices (three-dimensional: x, y, z), normals
28 // (three-dimensional: nx, ny, nz), and colors (r, g, b). The
29 // data layout consists of three flat vectors which store the
30 // vertex coordinates, the normals vector elements and the RGB
31 // colors one after the other.
32 //
33 const gridSize = 50;
34 var vertices = <| 3 * gridSize * gridSize |>;
35 var normals  = <| 3 * gridSize * gridSize |>;
36 var colors   = <| 3 * gridSize * gridSize |>;
```

```
37
38  //
39  // The face indices do not depend on the geometry. They have a fixed
40  // topology which only depends on the global parameter gridSize:
41  //
42  var faces =
43      map(
44          cartesianProduct(
45              range(0.0, float(gridSize-2), gridSize-2),
46              range(0.0, float(gridSize-2), gridSize-2)),
47          function(indices) {
48              var u0 = integer(indices@0);
49              var v0 = integer(indices@1);
50              var index = function(u, v) { return u * gridSize + v; };
51              return [
52                  index(u0, v0), index(u0+1, v0),
53                  index(u0+1, v0+1), index(u0, v0+1)];
54          });
55
56  //
57  // In two nested for loops the coordinates, surface normals,
58  // and the mean curvature can be calculated and stored in the vectors.
59  //
60  var uRange = range(-1.0, 1.0, gridSize - 1);
61  var vRange = range(-1.0, 1.0, gridSize - 1);
62  for (var uIndex : uValue in uRange) {
63      for (var vIndex : vValue in vRange) {
64          var xyz   = f(uValue, vValue);
65          var nxyz  = n(uValue, vValue);
66          var h     = mean(uValue, vValue);
67          //
68          // Map mean curvature (which is in the range of [-1, 1] for
69          // parameters in [-2, 2] x [-2, 2]) to color map indices.
70          //
71          var rgb   = colorMaps@"hueScale"@(integer((h + 1.0)*50.0));
72          //
73          // Store all values in corresponding vectors.
74          //
75          var index = uIndex * gridSize + vIndex;
76          vertices@(3*index + 0) = xyz@0;
77          vertices@(3*index + 1) = xyz@1;
78          vertices@(3*index + 2) = xyz@2;
79          normals@(3*index + 0) = nxyz@0;
80          normals@(3*index + 1) = nxyz@1;
81          normals@(3*index + 2) = nxyz@2;
82          colors@(3*index + 0) = rgb@0;
83          colors@(3*index + 1) = rgb@1;
84          colors@(3*index + 2) = rgb@2;
85      }
86  }
87
88  //
89  // Create polyface data structure and export the plot to X3D:
90  //
91  var plot = cadPolyFace(vertices, faces, normals, colors);
92  plot@"name" = "monkey_saddle";
93  fileOut(plot@"toFileZippedX3D"());
```

The executed example creates a ZIP-archive named “monkey_saddle.zip” with the file “euclides.x3d” inside. Extensible 3D (X3D) is the open standard for Web-delivered three-

dimensional graphics. It specifies a declarative geometry definition language, a run-time engine, and an application programming interface that provide an interactive, animated, real-time environment for 3D graphics [DB07]. The plotting result is illustrated in Figure 5.1 using the X3D viewer instantreality[1].

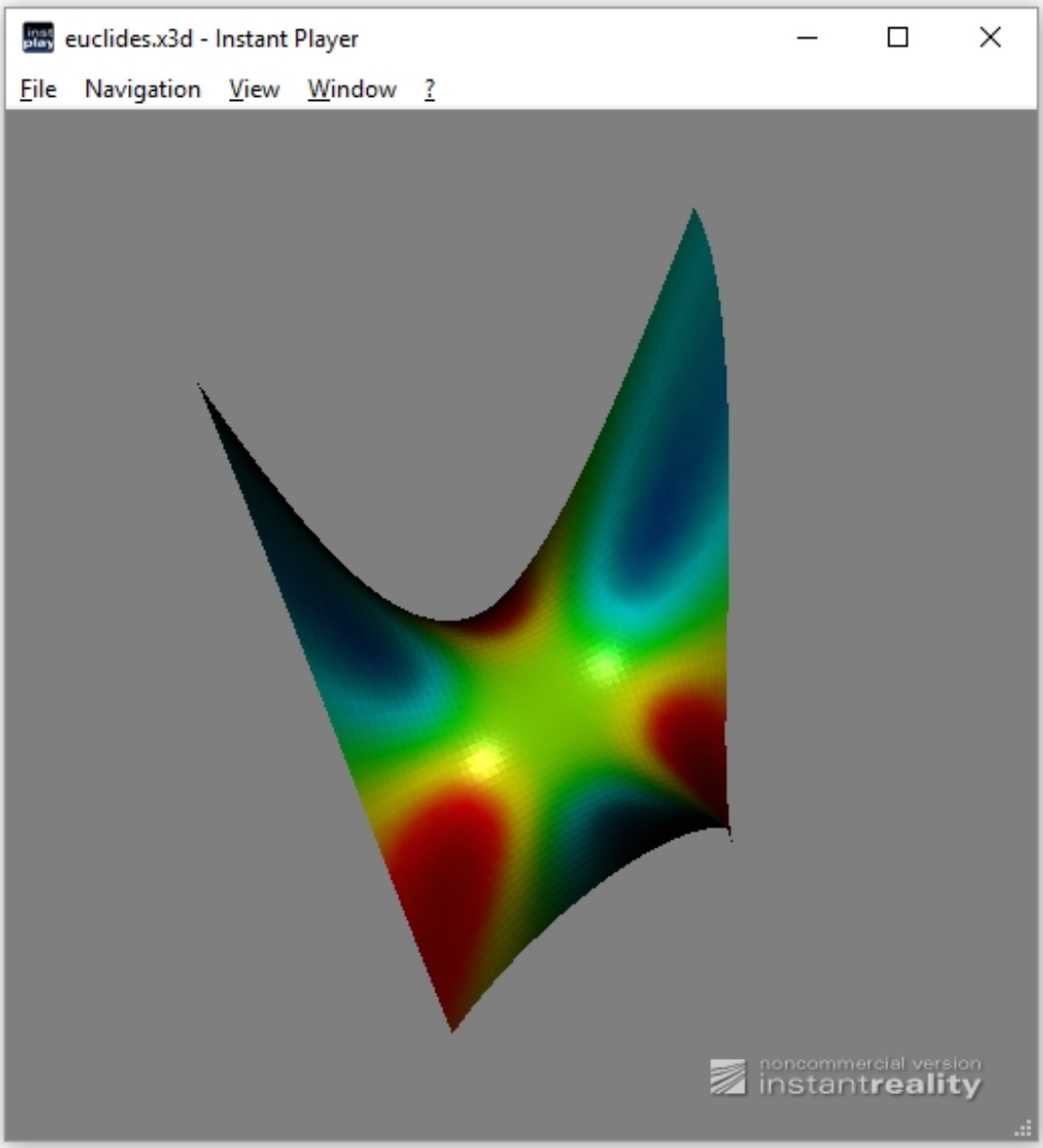


Figure 5.1: A simple grid-layout-based 3D plot. The colors of the surface indicate the mean curvature.

[1] www.instantreality.org

5.2 Library: CADPOLYFACETOOLS

Euclides Standard Library for Computer-Aided Design: polyface tools.

This library provides a set of functions to access, manipulate, and modify the polyface data structure:

- `cadPolyFaceTools@"find"`
- `cadPolyFaceTools@"findAll"`
- `cadPolyFaceTools@"get"`
- `cadPolyFaceTools@"set"`
- `cadPolyFaceTools@"merge"`

Furthermore, the `cadPolyFaceTools` object contains two auxiliary functions for the SVG export of `cadPolyFace`:

- `cadPolyFaceTools@"perspective"`
- `cadPolyFaceTools@"modelView"`

This library imports the library *math*.

---

cadpolyfacetools: The `cadPolyFaceTools` object provides the following functions to access and modify the polyface data structure:

- `cadPolyFaceTools@"find"` takes a `cadPolyFace` object `scene` and the parameter `name`. It returns the first object of `scene@"polyfaces"` with an attribute `"name"` equal to the parameter `name`.
- `cadPolyFaceTools@"findAll"` takes a `cadPolyFace` object `scene` and the parameter `name`. It returns an array of all objects of `scene@"polyfaces"` with an attribute `"name"` equal to the parameter `name`.
- `cadPolyFaceTools@"get"` takes a `cadPolyFace` object `scene`, the parameter `name`, and the parameter `attribute`. It uses `cadPolyFaceTools@"findAll"` to retrieve an array / subset of `scene@"polyfaces"` and substitutes all array elements `element` by `eleemnt@attribute`. The resulting array is returned.
- `cadPolyFaceTools@"set"` takes the same parameters as `cadPolyFaceTools@"get"` and an additional `value`. Instead of retrieving attributes (as in ecsCodethis@"get"), this method sets / overwrites the corresponding attributes.
- `cadPolyFaceTools@"merge"` merges an array of `cadPolyFace` objects to one `cadPolyFace` object.

Furthermore, the `cadPolyFaceTools` object contains two auxiliary functions for the SVG export of `cadPolyFace`:

- `cadPolyFaceTools@"perspective"` uses `perspectiveViewAngle` in degrees, `perspectiveAspectRatio` (should be 1.0), and the parameters to specify the near- and far-planes `perspectiveZNear`, `perspectiveZFar` (all parameters in this order) to calculate the persective matrix of the OpenGL fixed-function pipeline.
- `cadPolyFaceTools@"modelView"` needs three three-dimensional vectors `cameraEyePoint`, `cameraLookAtDirection`, `cameraLookUpDirection` to calculate a model view matrix of the OpenGL fixed-function pipeline.

The combination of the *cadPolyFace* library and its corresponding tool set *cadPolyFaceTools* is used to reimplement Brian E. Paul's famous 3-D gears program (gears.c) in EUCLIDES.

Source Code 5.2 — The 3-D gears program.

```
import 'blas'
import 'cadpolyface'
import 'cadpolyfacetools'
import 'io'
import 'math'
import 'sequence'

/**
 * This function is based on Brian Paul's 3-D gears.
 */
const gear = function(innerRadius, outerRadius, width, teeth,
    toothDepth, gearMaterial, gearName) {
    //
    // The radii of the gear:
    //
    const r0 = innerRadius;
    const r1 = outerRadius - toothDepth / 2.0;
    const r2 = outerRadius + toothDepth / 2.0;
    //
    // The geometry is represented by vertices (vs),
    // normals (ns), and faces (fs).
    //
    var vs = < 0.0, 0.0, 0.0 >;
    var ns = < 1.0, 1.0, 1.0 >;
    var fs = [ ];
    //
    const repetition4 = <
        < 1.0, 0.0, 0.0 >, < 0.0, 1.0, 0.0 >, < 0.0, 0.0, 1.0 >,
        < 1.0, 0.0, 0.0 >, < 0.0, 1.0, 0.0 >, < 0.0, 0.0, 1.0 >,
        < 1.0, 0.0, 0.0 >, < 0.0, 1.0, 0.0 >, < 0.0, 0.0, 1.0 >,
        < 1.0, 0.0, 0.0 >, < 0.0, 1.0, 0.0 >, < 0.0, 0.0, 1.0 > >;
    //
    var i = 0;
    while (i < teeth) {
        //
        // precomputed sinus and cosinus values
        //
        const angle000 = float(   i   * 2) * PI / float(teeth);
        const angle100 = float((i+1) * 2) * PI / float(teeth);
        //
        const cos000 = cos(angle000);
        const cos025 = cos(0.75 * angle000 + 0.25 * angle100);
        const cos050 = cos(0.50 * angle000 + 0.50 * angle100);
        const cos075 = cos(0.25 * angle000 + 0.75 * angle100);
        const cos100 = cos(angle100);
        //
        const sin000 = sin(angle000);
        const sin025 = sin(0.75 * angle000 + 0.25 * angle100);
        const sin050 = sin(0.50 * angle000 + 0.50 * angle100);
        const sin075 = sin(0.25 * angle000 + 0.75 * angle100);
        const sin100 = sin(angle100);
        //
```

```
53          var offset;
54          //
55          // top faces
56          //
57          offset = vs@"getDimension"() / 3;
58          vs = vs ++ <r0 * cos000, r0 * sin000, width * 0.5>;
59          vs = vs ++ <r1 * cos000, r1 * sin000, width * 0.5>;
60          vs = vs ++ <r2 * cos025, r2 * sin025, width * 0.5>;
61          vs = vs ++ <r2 * cos050, r2 * sin050, width * 0.5>;
62          ns = ns ++ repetition4 * < 0.0, 0.0, 1.0 >;
63          //
64          vs = vs ++ <r1 * cos075, r1 * sin075, width * 0.5>;
65          vs = vs ++ <r0 * cos075, r0 * sin075, width * 0.5>;
66          vs = vs ++ <r0 * cos100, r0 * sin100, width * 0.5>;
67          vs = vs ++ <r1 * cos100, r1 * sin100, width * 0.5>;
68          ns = ns ++ repetition4 * < 0.0, 0.0, 1.0 >;
69          //
70          fs = fs ++ [
71              [offset+0, offset+1, offset+2, offset+3, offset+4, offset+5],
72              [offset+4, offset+7, offset+6, offset+5] ];
73          //
74          // bottom faces
75          //
76          offset = vs@"getDimension"() / 3;
77          vs = vs ++ <r0 * cos000, r0 * sin000, width * -0.5>;
78          vs = vs ++ <r1 * cos000, r1 * sin000, width * -0.5>;
79          vs = vs ++ <r2 * cos025, r2 * sin025, width * -0.5>;
80          vs = vs ++ <r2 * cos050, r2 * sin050, width * -0.5>;
81          ns = ns ++ repetition4 * < 0.0, 0.0, -1.0 >;
82          //
83          vs = vs ++ <r1 * cos075, r1 * sin075, width * -0.5>;
84          vs = vs ++ <r0 * cos075, r0 * sin075, width * -0.5>;
85          vs = vs ++ <r0 * cos100, r0 * sin100, width * -0.5>;
86          vs = vs ++ <r1 * cos100, r1 * sin100, width * -0.5>;
87          ns = ns ++ repetition4 * < 0.0, 0.0, -1.0 >;
88          //
89          fs = fs ++ [
90              [offset+0, offset+5, offset+4, offset+3, offset+2, offset+1],
91              [offset+4, offset+5, offset+6, offset+7] ];
92          //
93          // inner cylinder
94          //
95          offset = vs@"getDimension"() / 3;
96          vs = vs ++ <r0 * cos000, r0 * sin000, width *  0.5>;
97          vs = vs ++ <r0 * cos075, r0 * sin075, width *  0.5>;
98          vs = vs ++ <r0 * cos000, r0 * sin000, width * -0.5>;
99          vs = vs ++ <r0 * cos075, r0 * sin075, width * -0.5>;
100         ns = ns ++ repetition4 * crossprod( < 0.0, 0.0, 1.0 >,
101             normalize(<r0 * cos075, r0 * sin075, width * 0.5>
102                     - <r0 * cos000, r0 * sin000, width * 0.5>));
103         //
104         vs = vs ++ <r0 * cos075, r0 * sin075, width *  0.5>;
105         vs = vs ++ <r0 * cos100, r0 * sin100, width *  0.5>;
106         vs = vs ++ <r0 * cos075, r0 * sin075, width * -0.5>;
107         vs = vs ++ <r0 * cos100, r0 * sin100, width * -0.5>;
108         ns = ns ++ repetition4 * crossprod(< 0.0, 0.0, 1.0 >,
109             normalize(<r0 * cos100, r0 * sin100, width * 0.5>
110                     - <r0 * cos075, r0 * sin075, width * 0.5>));
111         //
```

```
112             fs = fs ++ [
113                 [offset+0, offset+1, offset+3, offset+2],
114                 [offset+4, offset+5, offset+7, offset+6] ];
115             //
116             // outer surface
117             //
118             offset = vs@"getDimension"() / 3;
119             vs = vs ++ <r1 * cos000, r1 * sin000, width *  0.5>;
120             vs = vs ++ <r2 * cos025, r2 * sin025, width *  0.5>;
121             vs = vs ++ <r1 * cos000, r1 * sin000, width * -0.5>;
122             vs = vs ++ <r2 * cos025, r2 * sin025, width * -0.5>;
123             ns = ns ++ repetition4 * crossprod(normalize(
124                 <r2 * cos025, r2 * sin025, width * 0.5>
125               - <r1 * cos000, r1 * sin000, width * 0.5>),
126                 < 0.0, 0.0, 1.0 >);
127             //
128             vs = vs ++ <r2 * cos025, r2 * sin025, width *  0.5>;
129             vs = vs ++ <r2 * cos050, r2 * sin050, width *  0.5>;
130             vs = vs ++ <r2 * cos025, r2 * sin025, width * -0.5>;
131             vs = vs ++ <r2 * cos050, r2 * sin050, width * -0.5>;
132             ns = ns ++ repetition4 * crossprod(normalize(
133                 <r2 * cos050, r2 * sin050, width * 0.5>
134               - <r2 * cos025, r2 * sin025, width * 0.5>),
135                 < 0.0, 0.0, 1.0 >);
136             //
137             vs = vs ++ <r2 * cos050, r2 * sin050, width *  0.5>;
138             vs = vs ++ <r1 * cos075, r1 * sin075, width *  0.5>;
139             vs = vs ++ <r2 * cos050, r2 * sin050, width * -0.5>;
140             vs = vs ++ <r1 * cos075, r1 * sin075, width * -0.5>;
141             ns = ns ++ repetition4 * crossprod(normalize(
142                 <r1 * cos075, r1 * sin075, width * 0.5>
143               - <r2 * cos050, r2 * sin050, width * 0.5>),
144                 < 0.0, 0.0, 1.0 >);
145             //
146             vs = vs ++ <r1 * cos075, r1 * sin075, width *  0.5>;
147             vs = vs ++ <r1 * cos100, r1 * sin100, width *  0.5>;
148             vs = vs ++ <r1 * cos075, r1 * sin075, width * -0.5>;
149             vs = vs ++ <r1 * cos100, r1 * sin100, width * -0.5>;
150             ns = ns ++ repetition4 * crossprod(normalize(
151                 <r1 * cos100, r1 * sin100, width * 0.5>
152               - <r1 * cos075, r1 * sin075, width * 0.5>),
153                 < 0.0, 0.0, 1.0 >);
154             //
155             fs = fs ++ [
156                 [offset+0,  offset+2,  offset+3,  offset+1],
157                 [offset+4,  offset+6,  offset+7,  offset+5],
158                 [offset+8,  offset+10, offset+11, offset+9],
159                 [offset+12, offset+14, offset+15, offset+13]];
160             //
161             i = i + 1;
162         }
163         return cadPolyFace(vs, fs, ns,
164             undefined, undefined, undefined, gearMaterial, gearName);
165     };
166
```

```
//
// Create three gear instances:
//
var gearRed   = gear(1.0, 4.0, 1.0, 20, 0.7,
                     MaterialLibrary@"redPlastic",  "red_gear");
var gearGreen = gear(0.5, 2.0, 2.0, 10, 0.7,
                     MaterialLibrary@"greenPlastic", "green_gear");
var gearBlue  = gear(1.3, 2.0, 0.5, 10, 0.7,
                     MaterialLibrary@"bluePlastic",  "blue_gear");
//
// Merge the three gears to one CAD scene:
//
var scene = cadPolyFaceTools@"merge"(
   [gearRed, gearGreen, gearBlue], "3DGears");
//
// Modify the polyfaces in the scene by setting a new
// transformation matrix for each object. The function
// cadPolyFaceTools@"set" offers easy access to nested objects.
// The transformation matrices are calculated using the blas
// library functions translation3d and rotation3dz.
//
const angle = 13.456;
cadPolyFaceTools@"set"(scene, "red_gear", "transformation",
    translation3d(<-3.0, -2.0, 0.0, 0.0>)*rotation3dz(angle));
cadPolyFaceTools@"set"(scene, "green_gear", "transformation",
    translation3d(< 3.1, -2.0, 0.0, 0.0>)*rotation3dz(-2.0*angle-9.0));
cadPolyFaceTools@"set"(scene, "blue_gear", "transformation",
    translation3d(<-3.1,  4.2, 0.0, 0.0>)*rotation3dz(-2.0*angle-25.0));
//
// Export 3D scene to zipped X3D: 3DGears.zip
//
fileOut(scene@"toFileZippedX3D"());
//
// The export to SVG requires a 3D-to-2D projection in form of
// OpenGL matrices:
//
var perspective = cadPolyFaceTools@"perspective"(60.0, 1.0, 0.1, 1000.0);
var modelView   = cadPolyFaceTools@"modelView"(
  <10.0, 10.0, 10.0>, <-1.0, -1.0, -1.0>, <-1.0, -1.0, 2.0>);
//
// Rename scene to 2DGears and export it as zipped SVG: 2DGears.zip
//
scene@"name" = "2DGears";
fileOut(scene@"toFileZippedSVG"(perspective, modelView));
```

The executed gears application creates two ZIP-archives. The file “3DGears.zip” with the file “euclides.x3d” inside contains the scene description in X3D format (see Figure 5.2).

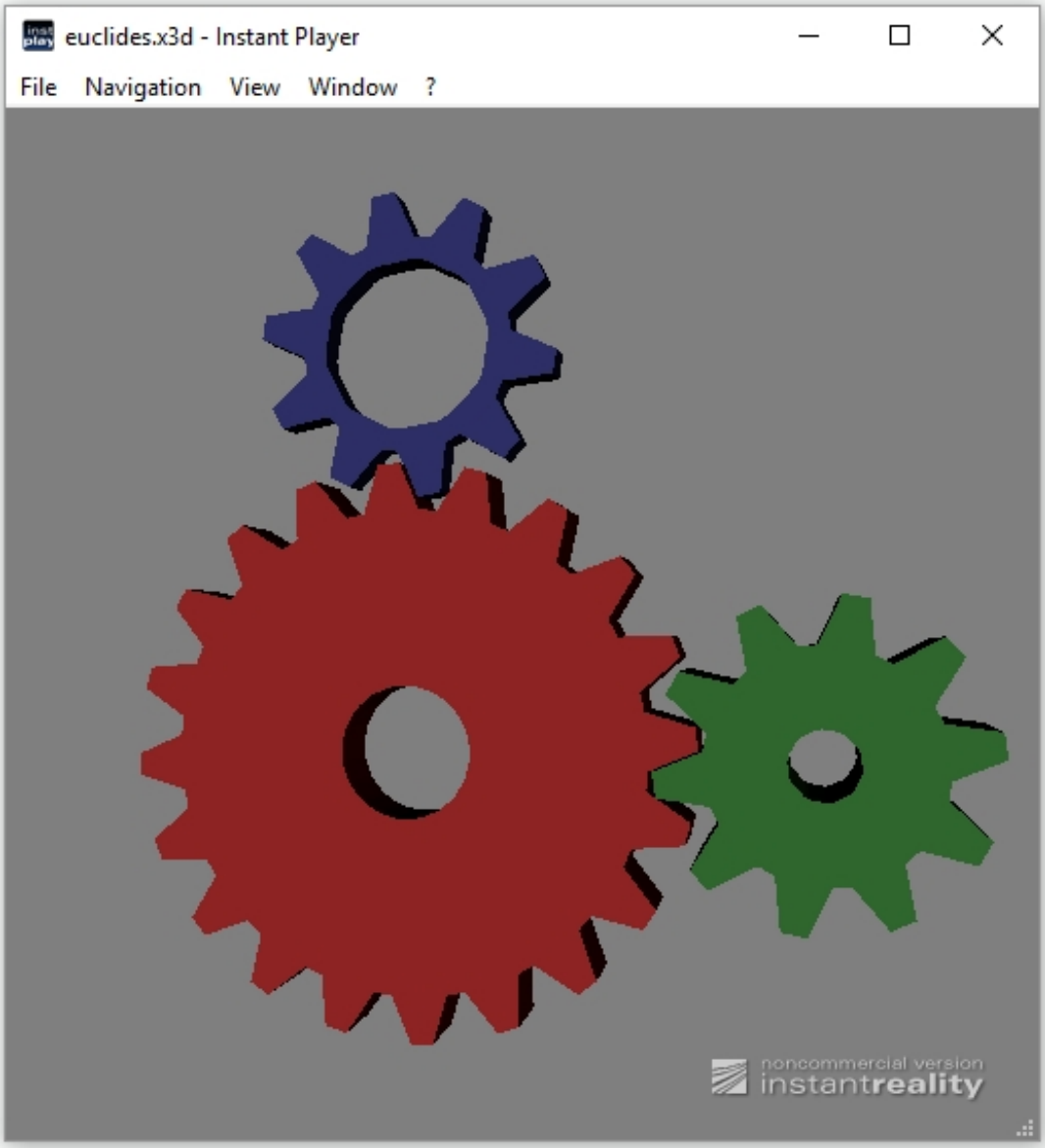


Figure 5.2: Brian E. Paul's famous 3-D gears program adopted to EUCLIDES and exported to X3D.

The file "2DGears.zip" is a 2D projection of the scene in SVG format. Scalable Vector Graphics (SVG) is an XML-based vector image format for two-dimensional graphics. As XML files, SVG images can be created and edited with any text editor, as well as with drawing software. Furthermore, all major modern web browsers have SVG rendering support. The projection (3D to 2D) is done using the *cadPolyFaceTools* functions `cadPolyFaceTools@"perspective"` and `cadPolyFaceTools@"modelView"` (analogue to gluPerspective of the OpenGL fixed-function pipeline API). The result is shown in Figure 5.3.

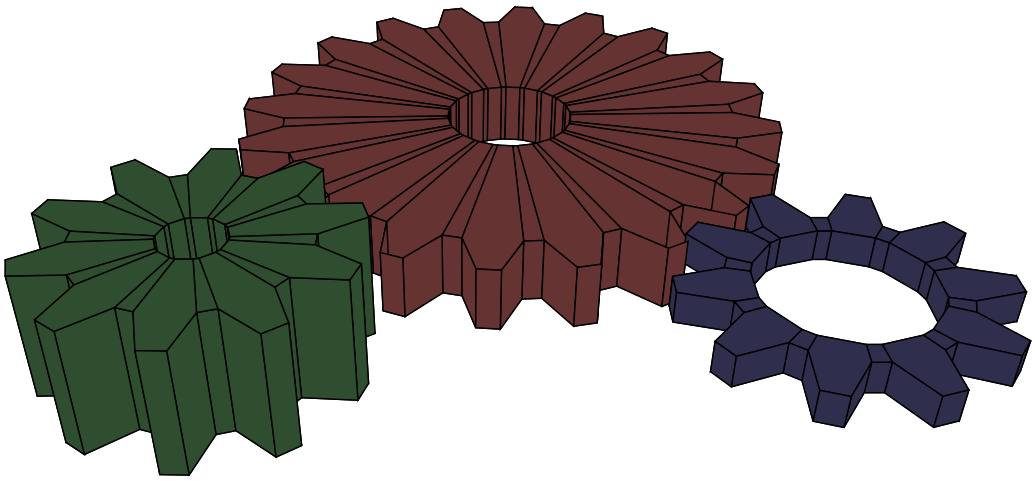


Figure 5.3: The gears example seen from eye point `<10.0, 10.0, 10.0>` with look-at direction `<-1.0, -1.0, -1.0>`. The up-vector of the camera is set to `<-1.0, -1.0, 2.0>` (see "The 3-D gears program", source code 5.2, line 204/205).

With increasing complexity the convenience of the tool library becomes clearer. For example, an imported ZIP archive may contain several STL files. The routine `cadPolyFaceFromFile` of the *cadPolyFace* library imports them all and names them according to the file names. Using the `"find"` function of *cadPolyFaceTools* these "building blocks" can be accessed by the respective file names. In this way, new scenes can be created by reusing static building blocks. The following example illustrates this concept.

Source Code 5.3 — The Gothic Cathedral.

```
1  import 'blas'
2  import 'cadpolyface'
3  import 'cadpolyfacetools'
4  import 'io'
5  import 'sequence'
6  //
7  // All building blocks are stored as STL files.
8  // All STL files are zipped in "gothic_cathedral.zip".
9  //
10 // The building blocks have been created by Skimbal (2010),
11 // licensed under the Creative Commons - Attribution license,
12 // and published as "Gothic Cathedral Play Set"
13 // at https://www.thingiverse.com/thing:2030
14 //
```

```
15  const zipArchive = fileIn("../data/gothic_cathedral.zip");
16  const buildingBlocks = cadPolyFaceFromFile(zipArchive);
17  //
18  // The Cathedral generater function:
19  //
20  // The first four parameters determine the length of the axes (semi-
21  // positive integers describing the number of building blocks) in
22  // the corresponding points of the compass (north, east, south, west)
23  // The second four parameters determine the completion of the building
24  // in the corresponding points of the compass (north, east, south,
25  // west). Possible values are 0 for sanctuary, 1 for transept facade,
26  // and 2 for main facade.
27  //
28  const buildCathedral = function(
29      nExtend, eExtend, sExtend, wExtend,
30      nFacade, eFacade, sFacade, wFacade) {
31      //
32      // access the building blocks
33      //
34      const bbCrossing = cadPolyFaceTools@"find"(buildingBlocks,
35          "CathedralCrossing.stl");
36      const bbCrossingTower = cadPolyFaceTools@"find"(buildingBlocks,
37          "CathedralCrossingTowerBase.stl");
38      const bbNave = cadPolyFaceTools@"find"(buildingBlocks,
39          "CathedralNave.stl");
40      const bbTower = cadPolyFaceTools@"find"(buildingBlocks,
41          "CathedralTower.stl");
42      const bbFacadeMainLeft = cadPolyFaceTools@"find"(buildingBlocks,
43          "CathedralFrontFacadeLeft.stl");
44      const bbFacadeMainRight = cadPolyFaceTools@"find"(buildingBlocks,
45          "CathedralFrontFacadeRight.stl");
46      const bbFacadeTranseptLeft = cadPolyFaceTools@"find"(buildingBlocks,
47          "CathedralTranseptFacadeLeft.stl");
48      const bbFacadeTranseptRight = cadPolyFaceTools@"find"(buildingBlocks,
49          "CathedralTranseptFacadeRight.stl");
50      const bbSanctuaryLeft = cadPolyFaceTools@"find"(buildingBlocks,
51          "CathedralSanctuaryLeft.stl");
52      const bbSanctuaryRight = cadPolyFaceTools@"find"(buildingBlocks,
53          "CathedralSanctuaryRight.stl");
54      //
55      // iteration names and values
56      //
57      const iteration = [ "A", "B", "C", "D" ];
58      const extends   = [ nExtend, eExtend, sExtend, wExtend ];
59      const facades   = [ nFacade, eFacade, sFacade, wFacade ];
60      //
61      // small helper function to copy a building block into the scene
62      //
63      const copyIntoScene = function(
64          scene, buildingBlock, name, transformation) {
65          var part = cadPolyFace(
66              buildingBlock@"vertices",
67              buildingBlock@"faces",
68              buildingBlock@"normals",
69              undefined, undefined, undefined,
70              MaterialLibrary@"silver", name);
71          cadPolyFaceTools@"set"(part, name, "transformation",
72              transformation);
73          scene@"back"(part);
74      };
```

```
75      //
76      // build scene
77      //
78      var scene = [ ];
79      //
80      // build crossing
81      //
82      for (var index : value in iteration) {
83          copyIntoScene(scene, bbCrossing, "crossing " ++ value,
84              rotation3dz(float(index) * PI / 2.0) *
85              translation3d(<-36.2674, -23.8965, 0.0, 0.0>));
86      }
87      //
88      // build crossing spire
89      //
90      copyIntoScene(scene, bbCrossingTower, "crossing tower base",
91          translation3d(<0.0, 0.0, 137.4361, 0.0>) *
92          rotation3dy(PI));
93      copyIntoScene(scene, bbTower, "crossing tower spire",
94          translation3d(<-9.1574, 0.0001, 137.4360, 0.0>));
95      //
96      // build nave
97      //
98      for (var index : value in iteration) {
99          var repeat = 0;
100         while (repeat < extends@index) {
101             var position =
102                 rotation3dz(float(index) * PI / 2.0) *
103                 translation3d(<float(repeat) * 51.5454, 0.0, 0.0, 0.0>);
104             //
105             copyIntoScene(scene, bbNave,
106                 "nave left " ++ value ++ ''repeat,
107                 position *
108                 translation3d(<90.2098, -23.8964, 0.0, 0.0>));
109             copyIntoScene(scene, bbNave,
110                 "nave right " ++ value ++ ''repeat,
111                 position *
112                 translation3d(<87.7823, 23.8855, 0.0, 0.0>) *
113                 rotation3dz(PI));
114             //
115             repeat = repeat + 1;
116         }
117     }
118     //
119     // build facades & sanctuary
120     //
121     for (var index : value in iteration) {
122         var position =
123             rotation3dz(float(index) * PI / 2.0) *
124             translation3d(< float(extends@index) * 51.5454,
125                             0.0, 0.0, 0.0>);
126         //
127         switch (facades@index)
128             case ([ 0 ]) {
129                 copyIntoScene(scene, bbSanctuaryLeft,
130                     "sanctuary left " ++ value,
131                     position *
132                     translation3d(<88.7261, 24.6747, 0.0, 0.0>) *
133                     rotation3dz(PI));
```

```
                    copyIntoScene(scene, bbSanctuaryRight,
                        "sanctuary right " ++ value,
                        position *
                        translation3d(<91.5078, -22.4851, 0.0, 0.0>));
                }
                case ([ 1 ]) {
                    copyIntoScene(scene, bbFacadeTranseptLeft,
                        "facade left " ++ value,
                        position *
                        translation3d(<89.3319, -33.6316, 0.0, 0.0>) *
                        rotation3dz(PI / 2.0));
                    copyIntoScene(scene, bbFacadeTranseptRight,
                        "facade right " ++ value,
                        position *
                        translation3d(<89.3319, 21.1189, 0.0, 0.0>) *
                        rotation3dz(PI / 2.0));
                }
                case ([ 2 ]) {
                    copyIntoScene(scene, bbFacadeMainLeft,
                        "facade left " ++ value,
                        position *
                        translation3d(<92.5864, -26.9248, 0.0, 0.0>) *
                        rotation3dz(PI));
                    copyIntoScene(scene, bbFacadeMainRight,
                        "facade right " ++ value,
                        position *
                        translation3d(<91.2848, 26.2318, 0.0002, 0.0>) *
                        rotation3dz(PI));
                    //
                    copyIntoScene(scene, bbTower,
                        "facade left tower " ++ value,
                        position *
                        translation3d(<86.1953, -37.2877, 109.6620, 0.0>));
                    copyIntoScene(scene, bbTower,
                        "facade right tower " ++ value,
                        position *
                        translation3d(<86.2950, 37.2758, 109.6620, 0.0>));
                }
        }
        //
        var result = cadPolyFaceTools@"merge"(scene);
        result@"name" = "Euclides Cathedral";
        return result;
};

//
// generate a cathedral
//
var cathedral = buildCathedral(1, 1, 2, 1, 0, 1, 2, 1);

//
// The export file name will be set automatically according
// to the name of the CAD model.
//
fileOut(cathedral@"toFileZippedX3D"());
```

Results of different configurations of the "Cathedral Generator" script are shown in Figure 5.4.

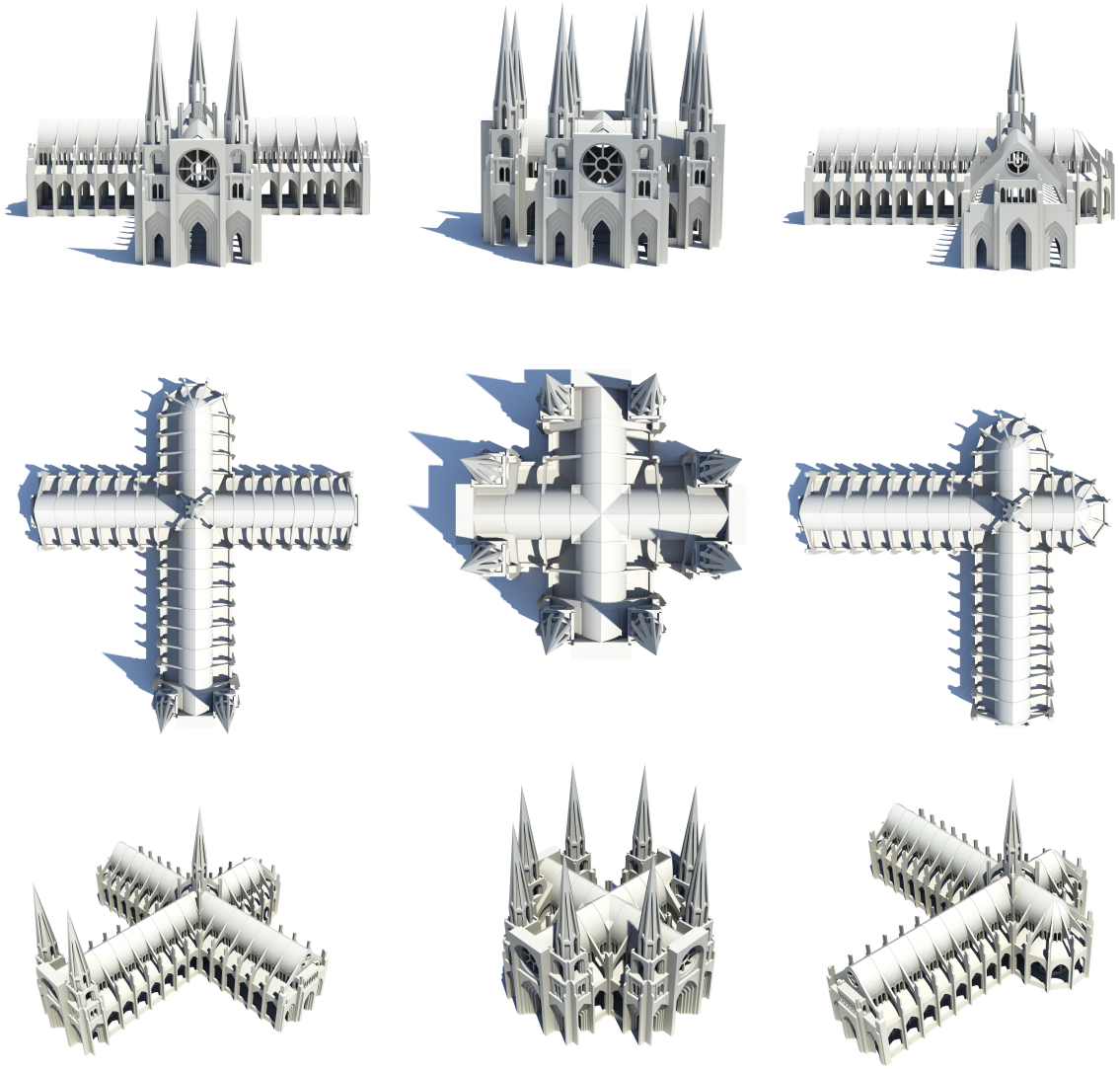


Figure 5.4: Using architectural building blocks in a generative system is an easy way to generate complex objects. The used building blocks of Gothic architecture have been created by Skimbal (2010), published as "Gothic Cathedral Play Set" at www.thingiverse.com.

5.3 Library: CADPOLYFACEFACTORY

Euclides Standard Library for Computer-Aided Design: factory for polyface data structures.

This library provides functions to generate the polyface data structures of polyhedra. All polyfaces consist of vertices, faces, normals, and uv-coordinates.

This library imports the libraries *math* and *cadpolyface*.

---

cadpolyfacefactory:

```
OBJECT{
  “Cuboctahedron” : ( ) → OBJECT{},
  “Dodecahedron” : ( ) → OBJECT{},
  “GreatDodecahedron” : ( ) → OBJECT{},
  “GreatIcosahedron” : ( ) → OBJECT{},
  “GreatRhombicuboctahedron” : ( ) → OBJECT{},
  “GreatRhombiicosidodecahedron” : ( ) → OBJECT{},
  “GreatStellatedDodecahedron” : ( ) → OBJECT{},
  “Hexahedron” : ( ) → OBJECT{},
  “Icosahedron” : ( ) → OBJECT{},
  “Icosidodecahedron” : ( ) → OBJECT{},
  “Octahedron” : ( ) → OBJECT{},
  “SmallRhombicuboctahedron” : ( ) → OBJECT{},
  “SmallRhombiicosidodecahedron” : ( ) → OBJECT{},
  “SmallStellatedDodecahedron” : ( ) → OBJECT{},
  “SnubCube” : ( ) → OBJECT{},
  “SnubDodecahedron” : ( ) → OBJECT{},
  “Tetrahedron” : ( ) → OBJECT{},
  “TruncatedDodecahedron” : ( ) → OBJECT{},
  “TruncatedHexahedron” : ( ) → OBJECT{},
  “TruncatedIcosahedron” : ( ) → OBJECT{},
  “TruncatedOctahedron” : ( ) → OBJECT{},
  “TruncatedTetrahedron” : ( ) → OBJECT{}
}
```

This constant contains all polyhedra generators of this library.

The following example (source code 5.4) creates all polyhedra of the library *cadPolyFace-Factory* and arranges them on a three-dimensional Fibonacci sphere.

Source Code 5.4 — The Polyhedra-Zoo.

```
1  import 'blas'
2  import 'cadpolyface'
3  import 'cadpolyfacefactory'
4  import 'cadpolyfacetools'
5  import 'math'
6  import 'io'
7
```

```
//
// Generate 3d positions on a sphere:
//
const fibonacciSphere = function(n, scaling=10.0) {
    const offset = 2.0 / float(n);
    const increment = PI * (3.0 - sqrt(5.0));
    //
    var points = [ ];
    var i = 0;
    while (i < n) {
        var y   = ((float(i) * offset) - 1.0) + (offset / 2.0);
        var r   = sqrt(1.0 - y^2);
        var phi = float((i + 1) % n) * increment;
        var x   = cos(phi) * r;
        var z   = sin(phi) * r;
        points@"back"(<scaling * x, scaling * y, scaling * z, 1.0>);
        //
        i = i + 1;
    }
    return points;
};

const positions = fibonacciSphere(cadPolyFaceFactory@"size"());

//
// Create polyhedra via polyface-factory:
//
var polyhedra = [ ];
for (var name : fun in cadPolyFaceFactory) {
    //
    // call factory function
    //
    var polyhedron = fun();
    //
    // set position
    //
    polyhedron@"polyfaces"@0@"transformation" =
        translation3d(positions@(polyhedra@"size"()));
    //
    // set material
    //
    var index = (polyhedra@"size"()) % (colornames@"scheme"@"size"());
    var color = colornames@(colornames@"scheme"@index);
    polyhedron@"polyfaces"@0@"material" =
        material(color, color, color, 1.0);
    //
    polyhedra@"back"(polyhedron);
}
//
// merge all polyhedra into one scene
//
var scene = cadPolyFaceTools@"merge"(polyhedra, "polyhedra_zoo");
//
// print zoo to standard out as XML
// and export to a zipped OBJ file:
//
print(<>scene);
fileOut(scene@"toFileZippedOBJ"());
```

The polyhedra zoo (source code 5.4) is exported to an OBJ file. The Object (OBJ) file format is a simple data-format that represents 3D geometry alone – namely, the position of each vertex, the UV position of each texture coordinate vertex, vertex normals, and the faces that make each polygon defined as a list of vertices. The file format is open and has been adopted by many 3D graphics applications. The exported result of the polyhedra zoo is visualized using Meshlab[2] [CCC+08] in Figure 5.5.

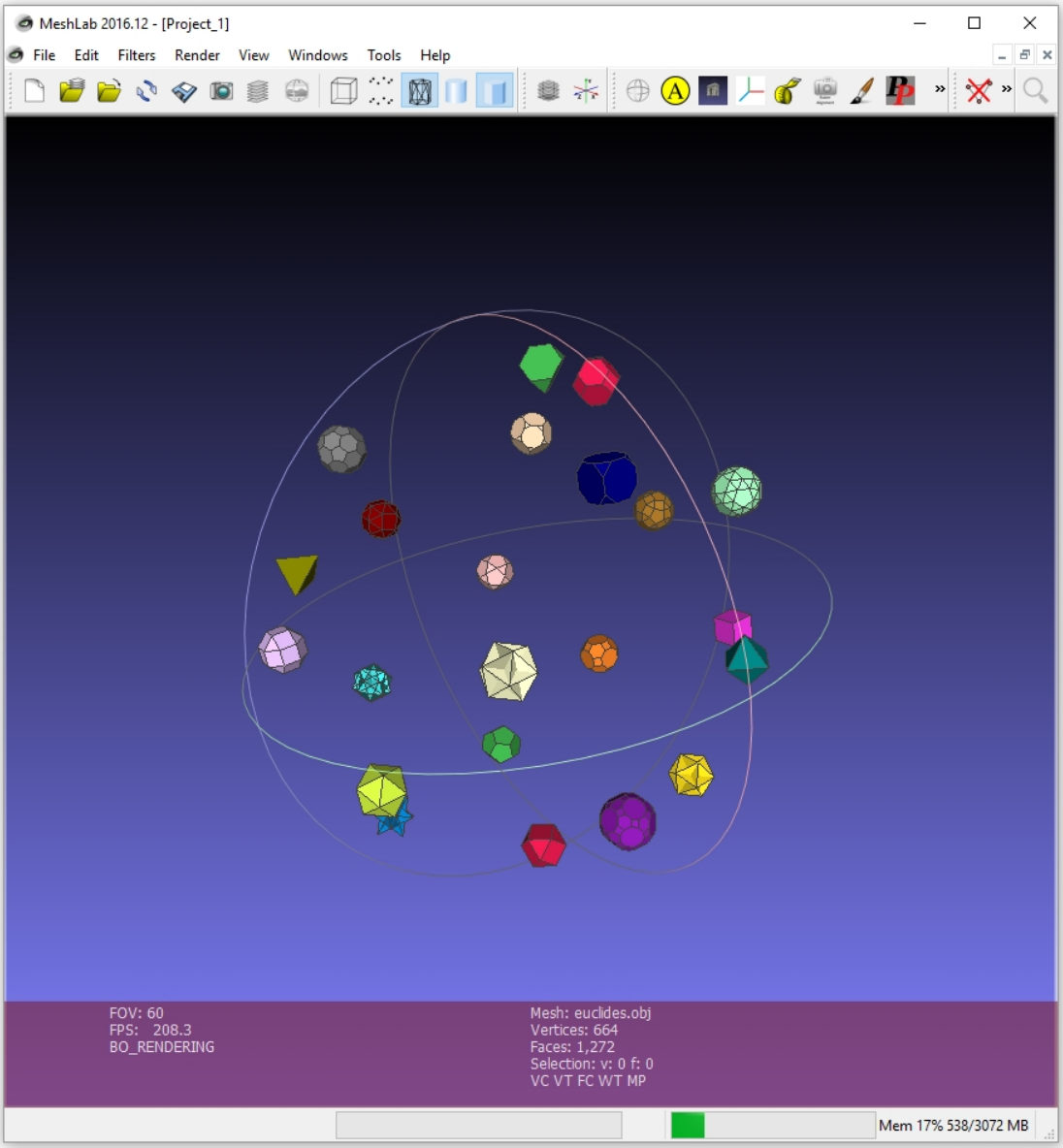


Figure 5.5: In geometry, polyhedra are solids in three dimensions with flat polygonal faces, straight edges and sharp corners or vertices. Source code 5.4 generates the shown zoo of polyhedra.

[2] www.meshlab.net

5.4 Library: CADAUTOMATION

Euclides Standard Library for design automation. This library provides one function: CADAutomation. It simplifies the publication of and the access to CAD designs.

This library imports the libraries *io* and *text*.

---

cadautomation:

```
(STRING, STRING,
 STRING, STRING,
 OBJECT{(design parameters)} → OBJECT{(see 'cadPolyFace' in library 'cadPolyFace')},
 OBJECT{(parameter description)})
  → OBJECT{(web service description)}
```

The function `cadAutomation` simplifies the publishing process of a generative model according to the idea of design automation. It takes several configuration parameters and returns a web service definition suitable to the `publish` function of the `io` library.
The first `name`, second `author`, third `information`, and fourth `copyright` parameter are strings that will be embedded into an HTML page describing the service. These strings will be copied without HTML transcoding, escaping, etc.
The fifth parameter is a function which generates a design (i.e. a `cadPolyFace`) based on the passed parameter object. The description of the parameters of the design function is passed as sixth parameter: an array of objects with a name, a short information and a default value.

The definition of the *cadAutomation* library function is explained and completed by the following example (source code 5.5). It creates platonic solids using the library *cadPolyFaceFactory*, arranges them on a three-dimensional Fibonacci sphere and exports the geometry generation process via a web service according to the design automation paradigm.

Source Code 5.5 — The Platonic Solids.

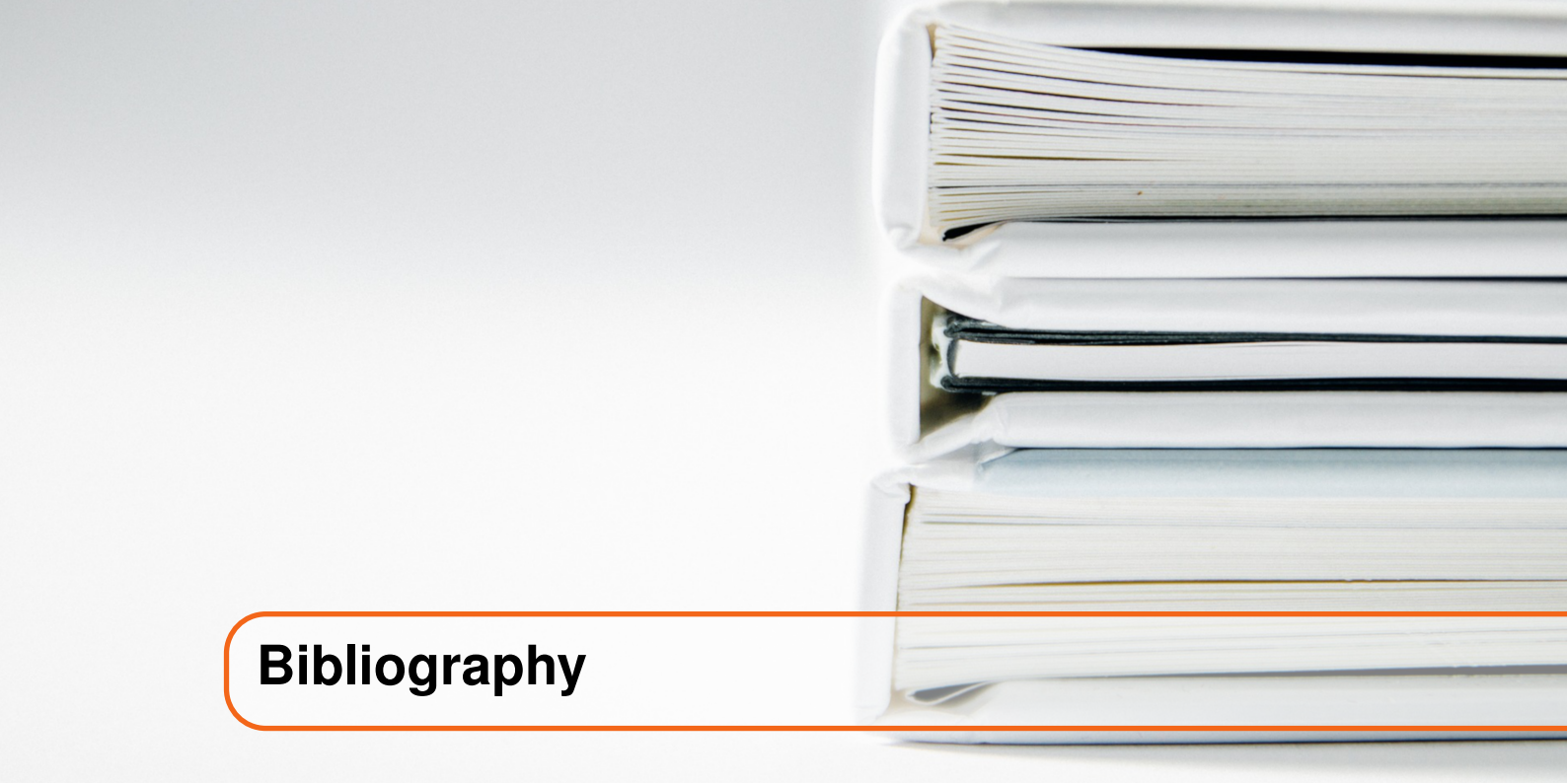
```
1  import 'blas'
2  import 'cadautomation'
3  import 'cadpolyface'
4  import 'cadpolyfacefactory'
5  import 'cadpolyfacetools'
6  import 'math'
7  import 'io'
8
9  //
10 // The web service has static information about itself:
11 //
12 const name      = "Euclides Polyhedra Webservice";
13 const author    = "Torsten Ullrich";
14 const info      = "A simple web service example powered by Euclides.";
15 const copyright = "&copy; T. Ullrich, 2019";
16
```

```
//
// The design to export is determined by a design space and a
// corresponding design generating function. The design spcae defines
// parameter types and ranges of valid parameters. The following example
// shows the definition of a string-based selection type. Besides the
// strings "name" and "info" it contains a string "value" with the
// default value and an array of strings containing all possible values:
//
var selectionType = {
    "name"  : "polyhedron",
    "info"  : "The name of the polyhedron.",
    "value" : "Dodecahedron",
    "range" : ["Tetrahedron", "Hexahedron", "Octahedron",
               "Dodecahedron", "Icosahedron"]
};

//
// For numerical values an arithmetic type (with all values being
// integer, float, or same-dim. vectors) can be used. The range of such
// a type defines its minimum, its step size, and its maximum:
//
var integerType = {
    "name" : "number",
    "info" : "The number of the polyhedra.",
    "value" : 5,
    "range" : [1 /** min */, 1 /** step */, 100 /** max*/]
};

//
// A vector example:
//
var vectorType = {
    "name"  : "color",
    "info"  : "The color of the polyhedron.",
    "value" : <1.0, 0.0, 0.0>,
    "range" : [ <0.0, 0.0, 0.0>, <0.1, 0.1, 0.1>, <1.0, 1.0, 1.0> ]
};

//
// For the sake of completeness, an example of a boolean type
// and a text type: Both types consist of "name", "info", and
// "value". A "range" is not needed, as it is not limited (text)
// or implicitly known (boolean).
//
var textType = {
    "name"  : "freeText",
    "info"  : "This type is not used in the example",
    "value" : "default text"
};

var booleanType = {
    "name"  : "trueOrFalse",
    "info"  : "This type is not used in the example",
    "value" : true
};

```

```
//
// The final design space consists of:
//
var design = [ selectionType, integerType, vectorType ];

//
// The corresponding design generating function is called by the
// web service. It receives an object with valid, corresponding
// parameters:
//
var generator = function(instanceParameters) {
    print("design generator called with:");
    print(''instanceParameters);
    //
    // generate 3d positions on a sphere
    //
    const fibonacciSphere = function(n, scaling=10.0) {
        const offset = 2.0 / float(n);
        const increment = PI * (3.0 - sqrt(5.0));
        //
        var points = [ ];
        var i = 0;
        while (i < n) {
            var y   = ((float(i) * offset) - 1.0) + (offset / 2.0);
            var r   = sqrt(1.0 - y^2);
            var phi = float((i + 1) % n) * increment;
            var x   = cos(phi) * r;
            var z   = sin(phi) * r;
            points@"back"(<scaling * x, scaling * y, scaling * z, 1.0>);
            //
            i = i + 1;
        }
        return points;
    };
    const positions = fibonacciSphere(instanceParameters@"number");
    //
    // create material
    //
    var color = instanceParameters@"color";
    var polyMaterial = material(color, color, color, 1.0);
    //
    // create polyhedra via polyface-factory:
    //
    var polyhedra = [ ];
    for (var _ : pos in positions) {
        //
        var polyhedron = cadPolyFaceFactory@(instanceParameters@"
            polyhedron")();
        polyhedron@"polyfaces"@0@"transformation" =
            translation3d(positions@(polyhedra@"size"()));
        //
        polyhedron@"polyfaces"@0@"material" = polymaterial;
        polyhedra@"back"(polyhedron);
        //
    }
    //
    return cadPolyFaceTools@"merge"(polyhedra, "polyhedra_zoo");
};

```

```
131 //
132 // Having defined all web service parts, it can be configured ...
133 //
134 var webservice = cadautomation(
135     name, author, info, copyright, generator, design);
136 //
137 // ... and published, respectively started:
138 //
139 publish(webservice);
140
141 //
142 // It is now accessible at locolhost:8080/index.html
143 //
```

References



Bibliography

[CCC+08] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. **MeshLab: an Open-Source Mesh Processing Tool**. Eurographics Italian Chapter Conference, 8:129–136, 2008.

[DB07] Leonard Daly and Don Brutzman. **X3D: Extensible 3D Graphics Standard**. IEEE Signal Processing Magazine, 24(6):130–135, 2007.

[HF07] Johan Ludvig Heiberg and Richard Fitzpatrick, editors. **Euclid's Elements Of Geometry**. Richard Fitzpatrick, 2007.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. **The C Programming Language**. Prentice Hall, 1978.

[KSU16] Ulrich Krispel, Christoph Schinko, and Torsten Ullrich. **A Survey of Algorithmic Shapes**. Remote Sensed Data and Processing Methodologies for 3D Virtual Reconstruction and Visualization of Complex Architectures, 219:498–529, 2016.

[SSUF10a] Christoph Schinko, Martin Strobl, Torsten Ullrich, and Dieter W. Fellner. **Modeling Procedural Knowledge – a generative modeler for cultural heritage**. Proceedings of EUROMED 2010 (Lecture Notes on Computer Science), 6436:153–165, 2010.

[SSUF10b] Martin Strobl, Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. **Euclides – A JavaScript to PostScript Translator**. Proceedings of the International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (Computation Tools), 1:14–21, 2010.