E U G E N E

A DOMAIN-SPECIFIC LANGUAGE FOR SYNTHETIC BIOLOGY

VERSION 2.0

Ernst Oberortner and Douglas Densmore

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
www.bu.edu/ece

May 25, 2015

Technical Report No. ECE-2014-NN

Contents

Chapter 1

Language Basics

The syntax of the Eugene v2.0 language case sensitive. All language statements are separated by the colon operator (`;`).

1.1 Comments

Eugene v2.0 supports two types of comments: single-line comments and multi-line comments.

```
// this is a single line comment

/*
this is a
multi-line
comment
*/
```

1.2 Primitive Values and Types

Eugene v2.0 supports three types of values: strings (i.e. character sequences), floating-point numbers, and booleans. The specification of strings must be enclosed in double quotas, e.g `"This is a string."`. Floating point numbers CAN have decimal places, such as `3.1415` or `100`. Booleans can only hold the values `true` or `false`.

Based on the supported primitive values, Eugene v2.0 provides three types of primitives: the type `txt` is used for strings, the type `num` is used for floating-point numbers, and the type `bool` is used for booleans.

1.3 Identifiers

In Eugene v2.0, identifiers are unique. That is, no two variables can have the same identifier. Identifiers are also case-sensitive. Also, identifiers must be defined before using and/or referring to them.

In Eugene v2.0, an identifier can be specified as a sequence of alphanumeric characters that does not start with a digit nor the underscore character ('_'). Hence, Eugene v2.0 identifiers must comply with the following pattern:

```
id ::= [a-zA-Z] ( [a-zA-Z0-9_] )*
```

Examples:

- declaration of an indentifier:
  `This_is_a_valid_indentifier`
- identifiers cannot start with the underscore character ('_'):
  `_This_is_an_invalid_indentifier`
- also, identifiers cannot start with a digit character (0-9):
  `0_is_an_invalid_indentifier`

1.4 Reserved Words

Reserved words can not be used as identifiers.

```
Property Part PartType \todo{Type} Device \todo{Template} Rule
```

The following list of reserved words can either be specified in upper-case as well as lower-case:

```
REPRESSES INDUCES DRIVES MORETHAN CONTAINS EXACTLY SAME_COUNT WITH
THEN BEFORE ALL_BEFORE SOME_BEFORE  AFTER ALL_AFTER SOME_AFTER NEXTTO
ALL_NEXTTO SOME_NEXTTO STARTSWITH ENDSWITH EQUALS ALL_FORWARD
ALL_REVERSE SOME_FORWARD SOME_REVERSE FORWARD REVERSE SAME_ORIENTATION
ALTERNATE_ORIENTATION AND OR NOT IF THEN ELSEIF ELSE FOR FUNCTION
```

keep this list updated

1.5 Variables

In Eugene v2.0, variables MUST have a type and a unique name (see Sections 1.2) and a unique identifier (see Section 1.3). Also, variables CAN have a value which depends of its specified type. Values can be assigned to variables by using constants, other variables, or expressions (see Section 1.7).

1.5.1 Primitive Variables

Examples:

- declaration of a primitive floating-point variable (`num`) with the identifier `n`:
  `num n;`
- declaration of a string variable with the identifier `s`:
  `txt s;`
- declaration of a boolean variable with the identifier `flag`:
  `bool flag;`

1.5.2 Arrays of Primitives

Arrays of primitives are ordered collections of primitive values of the same type. Eugene v2.0 supports two types of arrays: arrays of of floating point numbers (**num**[]) and arrays of strings (**txt**[]).

Examples:

```
num[] numbers;
txt[] dna_letters;
```

The elements in an array are indexed. Eugene v2.0 supports zero-based indices. That is, the first array element has index `0`. When accessing array elements, the index must be specified in squared brackets (`[]`).

Examples:

```
num one = numbers[0];
txt a = dna_letters[0];
```

In Eugene v2.0, the specification of an invalid index — either less than 0 or greater than the array length — will raise an error. For example, `numbers[4]` will result in an `ArrayIndexOutOfBounds` exception.

1.6 Assignments

Assignment statements assign values to variables and biological elements (as described later). Assignment statements consist of a left-hand-side (LHS), the assignment operator (`=`) and a right-hand-side (RHS). In Eugene v2.0, assignment statements are interpreted from right to left, i.e., the result of the RHS is assigned to the element specified on the LHS.

Examples of Variable Declarations and Value Assignments:

```
num pi = 3.1415;
txt hello = "Hello";
bool flag = true;
num[] numbers = [1, 2, 3, 4];
txt[] dna_letters = ["A", "T", "C", "G"];
txt a = dna_letters[0];
```

Values can be assigned to variables at the same time the variable is being declared or at later points during interpretation. An assignment overwrites the (current) values of a variable.

Examples Code of Overwriting a Variables Value:

```
// declaring and initializing a variable i
num i = 1;
// output of the value of variable i
println(i);
// incrementing the value of variable i
i = i + 1;
// output of the value of variable i
println(i);
```

This example code will ouput:

```
1
2
```

1.7 Expressions

Expressions can be useful to perform calculations based on constants and variable values. Eugene v2.0 expressions are calculated during the execution of a Eugene v2.0 script because Eugene v2.0 is an interpreted language and there are no particular pre-processing phases implemented (at the time of this writing). Eugene v2.0 also supports the use of parentheses to force a particular order of evaluating the expression. If parts of an expressions are enclosed in parentheses, then that part is evaluated first. The resulting value is used in the evaluation of the remainder of the expression. In this section we only explain numerical expressions in Eugene v2.0 that is calculated based on the following operator precedence:

```
( ) ... Parenthesis
* / ... Multiplication and Division
+ - ... Addition and Subtraction
```

In Section **??** and Chapter 5 we explain the support for boolean expressions respectively relational expressions and logical expressions of Eugene v2.0.

1.7.1 Semantics of the Expression Operators

Eugene v2.0 supports the utilization of the binary expression operators among different primitive types and values. The semantics of combining two values of different types regarding each binary expression operator is explained below. If there is no explanation provided, then the expression operator is not supported on combining the two different types.

The semantics of the binary `+` operator are defined as follows:

- The addition of two numerical values (`num + num`) results in the sum of the two numerical values.
- The addition of a numerical value and an array of numerical values (`num + num[]`) results in an array of numerical values `num[]` which is the concatenation of the numerical value and the array of numerical values.
- The addition of an array of numerical values and a numerical value (`num[] + num`) results in an array of numerical values `num[]` which is the concatenation of the array of numerical values and the numerical value.
- The addition of an numerical value and a string (`num + txt`) results in a string (`txt`) which is the concatenation of the numerical value and the string value.
- The addition of a string and an numerical value (`txt + num`) results in a string (`txt`) which is the concatenation of the string and the numerical value.
- The addition of an numerical value and an array of strings (`num + txt[]`) results in an array of strings (`txt[]`) which is the concatenation of the numerical value and the array of strings.
- The addition of an array of strings and a numerical value (`txt[] + num`) results in an array of strings (`txt[]`) which is the concatenation of the array of strings and the numerical value.
- The addition of a string and an array of strings (`txt + txt[]`) results in an array of strings (`txt[]`) which is the concatenation of the string and the array of strings.
- The addition of an array of strings and a string (`txt[] + txt`) results in an array of strings (`txt[]`) which is the concatenation of the array of strings and the string.
- The addition of two boolean values (`bool + bool`) results in a boolean value (`bool`) which is the logical conjunction of the two boolean values.

In Eugene v2.0, the binary `-`, `*`, and `/` operators are only defined on numerical primitives, namely subtraction, multiplication, and division respectively.

1.7.2 Examples of Expressions:

In all examples, we utilize the `println` function to output the result of an expression including a line feed. `println` is a built-in function in Eugene v2.0 (see Section 5.4).

- The following Eugene snippet outputs the famous "Hello World" string:

```
txt hello = "Hello";
hello_world = hello + " World";
println(hello_world);
```

- The following Eugene snippet outputs the concatenation of two DNA sequences:

```
txt seq1 = "ATCG";
txt seq2 = "CGAT";
seq = seq1 + seq2;
println(seq);
```

- The following Eugene snippet outputs the concatenation of two string arrays:

```
txt[] letters1 = ["A", "T"];
txt[] letters2 = ["C", "G"];

DNA_letters = letters1 + letters2;
println(DNA_letters);
```

- The following Eugene snippet outputs the logical conjunction of two boolean values:

```
bool t = true;
bool f = false;
conjunction = t + f;
println(conjunction);
```

- The following Eugene snippet outputs the logical conjunction of two boolean values:

```
bool t = true;
bool f = false;
conjunction = t + f;
println(conjunction);
```

1.8 Error Reporting

In Eugene v2.0 we differentiate among various types of errors ranging from syntax errors, over unknown identifiers and incompatible types, to inconsistent rules. Any type of error is thrown as a Java `IllegalArgumentException` and reported to the output console. Also, the interpretation of a defective Eugene v2.0 script stops after printing the error to the console. Error messages have the following format:

```
@Error
Line <line-no> Position <position-in-line>
<Error-Message>
```

Chapter 2

Declarative Language Features

In Eugene v2.0, users can manually specify and/or automatically import a "library" of design-specific biological components. The data exchange facilities of Eugene v2.0 are presented in Chapter 3. Furthermore, Eugene v2.0 provides "constraints" for selecting and composing the library's components into more complex biological systems.

Eugene v2.0 divides the specification of a system's design into two complementing categories:

- *Facts* represent biological components in various levels of abstractions, such DNA sequence, parts, or devices. Facts at the Part level have a type (such as `Promoter`, `Coding Sequence`) and user-defined characteristics (such as `strength`, *operator sites*).

- *Rules* represent constraints that restrict and ensure the proper selection of *Facts* in a design based on user-defined characteristics, as well as the biologically valid composition of *Facts*. For example, a ribosome binding must appear immediately upstream of a coding sequence in the same orientation. *Rules* are described more detailed in Chapter 4.

In addition, Eugene v2.0 provides *Built-In Functions* that enable the automated enumeration of rule-compliant compositions of biological facts. The `product` and `permute` functions are explained in Section 5.4.

2.1 Specification of Biological Components — "Facts"

In Eugene v2.0, biological facts can either be specified manually or automatically imported using data exchange standards. In this section, we demonstrate the manual specification of facts in Eugene v2.0. The automated imported is described in Chapter 3.

2.1.1 Characteristics of Biological Components — `Property`

Properties represent attributes and characteristics of primitive biological components, such as parts. Each property is defined with the `Property` keyword, a unique identifier (see Section 1.3), a primitive type (see Section **??**) specified in parenthesis, and the semi-colon character (';'). In the following we provide examples of specifying properties of all provided types in Eugene v2.0.

Examples:

```
// A numerical property named strength
Property strength(num);

// A textual property named part_name
Property part_name(txt);

// A property of type bool having the name is_valid
Property is_valid(bool);

// The letters property is of an array of strings
Property letters(txt[]);
```

2.1.2 Types of Components — `PartType`

Types specify common attributes and characteristics (i.e. properties) of biological components. Hence, a type is an abstract representations of the common structure of genetic elements, such as parts. Eugene v2.0 supports the definition of part types. That is, part types specify common properties of parts. For example, a part type `Promoter` can define promoter characteristics of interest, such as name, sequence, operator sites, or strength.

In Eugene v2.0, part types are defined using the `PartType` keyword followed by a unique name of the part type and a list of properties in parenthesis and concluded with the semi-colon character (';').

Examples:

```
// A part type w/o properties
PartType GenericPart();

// A promoter has a name, operator sites, a strength,
// and a flag that indicates if a promoter is inducible
Property name(txt);
Property operator_sites(txt[]);
Property strength(num);
```

```
Property is_inducible(bool);
PartType Promoter(name, op_sites, strength, is_inducible);

// A repressor coding-sequence is orthogonal to a promoter
Property orthogonal_promoter(txt);
PartType CodingSequence(name, orthogonal_promoter);
```

In Eugene v2.0 the properties of a part type must be defined before the part type declaration (see Section 2.1.1).

To support backward compatibility to earlier Eugene versions, Eugene v2.0 also supports the or `Part` keyword for the declaration of part types.

As described in the next section, each part has two pre-defined properties: `SEQUENCE` and `PIGEON`.

2.1.3 Instantiations of Part Types — `Part`

As described in the previous section, types define common properties of biological components. In Eugene v2.0, biological components are instances of types. That is, parts are instances of part types with concrete values of properties. For examples, an instance of the type `Promoter` can have a specific name (e.g. "`BBa_I14018`").

In Eugene v2.0, the declaration of a type instance starts with the name of the type followed by a unique name of the biological component, the values of its properties in parenthesis (`( )`), and the terminating semi-colon character (`;`).

```
// Properties
Property name(txt);

// PartType declaration
PartType Promoter(name);

// Instantiation of a Promoter
Promoter I14018("BBa_I14018");
```

The first identifier of an instance definition refers to the type of the instance (see Section 2.1.2). The second identifier is the name of the instance. Now, let's consider the following Eugene script specifying three properties and one part type.

```
// Properties
Property txtProp(txt);
Property numProp(num);
Property boolProp(bool);

// Part Type
PartType MyPartType(txtProp, numProp, boolProp);
```

Eugene v2.0 supports two variants of specifying the property values of parts:

- **The "list" Notation**:
  When using the "list" notation the property values must be specified in the same order as defined in the type's declaration. For example:

```
// myPart1 has all three properties set
MyPartType myPart1("part1", 1, true);

// myPart2 has only the first and second property set
MyPartType myPart2("part2", 2);

// myPart3 has only the first property set
MyPartType myPart3("part3");
```

- **The "dot" Notation**:
  When using the "dot" notation, the name of the property must be specified and its value. The "dot" notation allows to set the property values in a user-specific order, making it also possible to keep the value of certain properties empty.

```
// myPart1 has all three properties set
MyPartType myPart1(.txtProp("part1"), .numProp(1), .boolProp(true));

// myPart2 has only the second and last property set
MyPartType myPart2(.numProp(2), .boolProp(false));

// myPart3 has only the last property set
MyPartType myPart3(.boolProp(true));
```

In Eugene v2.0, we do not support the assignment of property values to undefined properties. Each property must be defined in the type specification. However, in Eugene v2.0 every part has two pre-defined properties of type `txt`:

- `SEQUENCE`
  This property serves to specify the DNA sequence of a part.

- `PIGEON`
  This property serves to specify the Pigeon statement for visualizing a part.[1]

The values of both properties can either be set by using the property name in upper- or lower-case. That is `PIGEON`, `pigeon`, `SEQUENCE`, `sequence`.

Note!

- When using the "dot" notation, the names of properties are case-sensitive, except the two predefined properties `SEQUENCE` and `PIGEON`.
- Eugene v2.0 does perform type checking of the property values with the type of the property. For example, it is not possible to assign a `num` value to a property of type `txt`.
- The two pre-defined properties cannot be specified using the "list" notation. Hence, setting the two pre-defined properties `SEQUENCE` and `PIGEON` must be done using the "dot" notation.

```
Promoter I14018(
  .name("BBa_I14018"),
  .SEQUENCE("...tacataggcgagtactctgttatgg"),
  .PIGEON("p BBa_I14018 14 nl"));
```

- Eugene v2.0 does not support mixing the "dot" and "list" notations. The following statement is not allowed, for example.

```
Promoter I14018("BBa_I14018",
  .SEQUENCE("...tacataggcgagtactctgttatgg"),
  .PIGEON("p BBa_I14018 14 nl"));
```

2.1.4 Composite Biological Facts — `Device`

Eugene v2.0 supports also the manual specification of composite biological facts, so-called *Devices*. Devices can be composed of *Parts* and other *Devices*.

The specification of a device starts with the `Device` keyword, followed by the device's identifier, and a comma-separated list (`,`) of the device's sub-components enclosed in parenthesis (`()`).

In addition, the *orientation* of the device's sub-components can be specified. Eugene v2.0 supports three types of orientation: *forward* or *reverse*, or *undefined*. Forward oriented sub-components must be denoted with the plus character (`+`), and reverse oriented sub-components must be denoted with the minus character (`-`). If there's no orientation specified, then the sub-component has an undefined orientation.

Examples:

- The `proms` device composes the `I14018` promoter twice, each having an undefined orientation.

```
Device proms(I14018, I14018);
```

- The `fproms` device is a composition of two forward oriented promoters: `I14018` and `I14018`.

```
Device fproms(+I14018, +I14018);
```

- The `rproms` device is a composition of two reverse oriented promoters.

  ```
  Device rproms(-I14018, -I14018);
  ```

- The `hierarchical` device is a composition of the `fproms` device and an additional forward oriented promoter.

  ```
  Device hierarchical(fproms, +I14018);
  ```

2.1.5 Relations and Interactions among Biological Facts

Chapter 3

Data Exchange Capabilities

Eugene v2.0 supports the following types of data exchange facilities:

- INCLUDE of Eugene scripts (Section 3.1).
- IMPORT and ASSIGNMENT of data represented in Eugene syntax (Section 3.2).
- IMPORT of biological data from the iGEM partsregistry (Section 3.3).
- IMPORT and EXPORT of biological data from Genbank files (Section 3.4).
- IMPORT and EXPORT of SBOL-compliant biological data (Section 3.5).
- VISUALIZATION using the SBOL Visual graphical notation (Section 3.6).

3.1 Including Eugene Scripts — `include`

Eugene v2.0 supports to include Eugene scripts within Eugene scripts via the `include` keyword. This features comes handy if common functionalities and data needs to be shared among several Eugene scripts. For example, the `include` keyword enables to include a library of function prototypes or biological components within several Eugene scripts.

All data and functionalities (e.g. function prototypes) are then available in the script that specifies the `include`. For example, if a `main.eug` script includes the `functions.eug` script, which defines the `foo` function, then the `foo` function becomes available in the `main.eug` script. That is, when utilizing the `include` statement, then the Eugene interpreter executes the included Eugene script.

The syntax of the `include` statement is defined as follows:

⟨*include-statement*⟩ ::= **include** `string`

The string argument must specify the absolute path or the relative path of the included Eugene script to the actual Eugene script. Eugene v2.0 tries to include the specified file regardless of its extension. That is, it does not matter if it the specified

file ends with `.eug` or `.h`. The only requirements are that the specified file exists and that it contains well-formed Eugene syntax. Otherwise, an error is reported.

Examples

- Including a Eugene script with **absolute path** specification.

  ```
  include /eugene_scripts/includes/my_promoters.h
  ```

- Including a Eugene script with **relative path** specification.

  ```
  include ./includes/my_promoters.h
  ```

Note!

- Eugene v2.0 does not check for cyclic inclusions of Eugene script. For example, if a Eugene script `A.eug` includes `B.eug` and `B.eug` includes `A.eug`, then an endless loop is the result.
- When multiple included Eugene scripts contain elements with the same name, then an error is reported. For example, if a Eugene script `main.eug` includes two Eugene scripts each containing a promoter `p1`.

3.2 Data Import from other Eugene Scripts — `import`

Similarly to the `include` statement (Section **??**), Eugene v2.0 provides the `import` statement. The main difference is that the imported data can be assigned to a Eugene v2.0 `Collection` (Section 5.3.1). Furthermore, function prototypes are not available in the in the script that specifies the `import` statement.

The syntax of the `include` statement is defined as follows:

⟨*import-statement*⟩ ::= (⟨*assignment*⟩)? **import** (`string`)

⟨*assignment*⟩ ::= `name` =

The `assignment` of the imported data is optional (denoted by `?`). If the assignment is specified, then the imported data is assigned to the element named by `name` and is accessible via the `name` element. Otherwise, the imported data is saved into, and accessible via, the symbol tables.

Examples

- Importing of the data specified in the `promoters.eug` script and printing the `p1` promoter to the console.

```
// importing the components of the promoters.eug script
import ("/eugene_scripts/data/promoters.eug");
// console output of the p1 promoter
println(p1);
```

- Importing of the data specified in the `promoters.eug` script, assigning it to the `promoters` collection, and printing the `p1` promoter to the console.

```
// assigning the imported components to the promoters collection
promoters = import ("/eugene_scripts/data/promoters.eug");
// now, we need to access the p1 promoter via the promoters container
println(promoters.p1);
```

Note!

- The `import` statement comes handy if multiple collections of biological data need to be imported, since the imported data can be assigned to elements in the main Eugene script. Otherwise, we recommend utilizing the `include` statement.

3.3 Data Import from the iGEM Partsregistry — `Registry.import`

what is the iGEM partsregistry? [1]

3.4 Data Exchange using GenBank

3.4.1 Data Import from GenBank — `Genbank.import`

3.4.2 Data Export to Genbank — `Genbank.export`

what is Genbank? [2]

[1] http://parts.igem.org
[2] http://www.ncbi.nlm.nih.gov/genbank/

3.5 Data Exchange using the SBOL Standard

The Synthetic Biology Open Language (SBOL)[3] is a community-driven effort to standardize the exchange of biological data and the design of synthetic biological systems.

Eugene v2.0 is an SBOL v1.1.0 compliant language and supports

1. **SBOL Import** (i.e. the deserialization of biological data and design represented in SBOL format) via the Eugene v2.0 `SBOL.import` statement (Section 3.5.2), and

2. **SBOL Export** (i.e. the representation of Eugene designs in serialized SBOL format), via the Eugene v2.0 `SBOL.export` statement (Section 3.5.3).

A synthetic biology design workflow consisting of the utilization of various genetic design automation tools, including Eugene, is presented in.[2] The SBOL files, which are utilized to exemplify the Eugene v2.0 SBOL import and export facilities, are available in Appendix A.

3.5.1 Mapping between Eugene v2.0 and SBOL v1.1.0

In Table 3.1 we list the types of the Eugene v2.0 data model and its corresponding types of SBOL v1.1.0 core data model.[3]

Eugene v2.0	**SBOL v1.1.0**
`Collection`	`Collection`
`Array`	`Collection`
`Part`	`DnaComponent` w/ `DnaSequence` but w/o `SequenceAnnotation`
`PartType`	`DnaComponent` w/o `DnaSequence`
`Device`	`DnaComponent` w/ `DnaSequence` and `SequenceAnnotation`

Table 3.1: Mapping of the Eugene v2.0 data model to the SBOL v1.1 data model

The Eugene v2.0 `Collection` and `Array` data types are mapped to a SBOL v1.1.0 `Collection` type.

A Eugene v2.0 `PartType` is an abstract type and, hence, is mapped to a SBOL v1.1.0 `DnaComponent` without a `DnaSequence`. SBOL v1.1.0 supports terms of the Sequence Ontology (SO)[4]. That is, if a Eugene v2.0 `PartType` is named using a term of the SO, such as `Promoter`, then the SO term is preserved via SBOL v1.1.0.

In Eugene v2.0, every `PartType` has a pre-defined `SEQUENCE` property, making it possible to specify the sequence of a `Part`. If a Eugene v2.0 does contain a non-empty `SEQUENCE` property value, then the Eugene v2.0 `Part` is mapped to a SBOL v1.1.0 `DnaComponent` containing a `DnaSequence`. However, if a Eugene v2.0 `Part` has an

[3] http://sbolstandard.org
[4] http://www.sequenceontology.org

empty `SEQUENCE` property value, then the `Part` is only mapped to a SBOL v1.1.0 `DnaComponent`.

As in SBOL v1.1.0, Eugene v2.0 supports the specification of a hierarchically composed `Device`. That is, a `Device` that contains sub-components where at least one sub-component is a `Device` (Section 2.1.4). Also, Eugene v2.0 supports the specification of *abstract devices* and *device instances*. Abstract devices are also called *Design Templates* (see 4.4) an enable the enumeration of all rule-compliant devices using Eugene v2.0 built-in functions `product` and `permute` (see Section 5.4). A Eugene v2.0 `Device` is mapped — depending on its structure — to a SBOL v1.1.0 `DnaComponent` with `SequenceAnnotations` that enable the specification of the sub-components of the`DnaComponent`. Also, Eugene v2.0 generates the `DnaSequence` of a device if the `Device` is composed of only parts.

3.5.2 Import from SBOL — `SBOL.import`

Eugene v2.0 provides the `SBOL.import` statement to import data and designs represented in the SBOL file format. The `SBOL.import` statement takes as input one argument, i.e. the absolute or relative path of the SBOL file. Then, it returns the imported data whose type depends on the content of the SBOL format.

Examples

- Importing data from an SBOL file and assigning the data to the `sbolData` Eugene element.

```
// import and assignment
sbolData = SBOL.import("./sbol/my_sbol_data.sbol");
// console output of the imported data
print(sbolData);
```

- Assignment of a hierarchically composed device that is represented in SBOL to a Eugene device.

```
// import and assignment
Device aDevice = SBOL.import("device.sbol");
// console output of the imported data
print(aDevice);
```

Note!

- If assigning the content of an SBOL file to a Eugene element and the content is not known, then there should be not type specified at the left-hand-side element of the assignment. That is, if the type of the SBOL content does not match the type of the Eugene element, then an error is reported.

3.5.3 Export to SBOL — `SBOL.export`

Eugene v2.0 provides the `SBOL.export` statement in order to serialize Eugene data to SBOL files. The `SBOL.export` statement requires two arguments: (1) the name of the Eugene element that should be serialized to SBOL and (2) the name of the SBOL file. If the specified name does not refer to a Eugene element, then an exception will be reported. Also, Eugene v2.0 will create all directories and the SBOL file, if permissions are granted. Otherwise, an exception is reported. Also, if the SBOL file exists, then it will be overwritten.

Examples

- Serialize the `myDesign` device to the `myDesign.sbol` file in the current directory (`./`).

```
// specification of a design
Device myDesign(p,r,c,t);
// export the design to SBOL
SBOL.export(myDesign, "./myDesign.sbol");
```

- Enumerate an array of rule-compliant designs using the `product` function (Section 4.5) and export the designs to the `allDesigns.sbol` file in the `./sbol/` directory.

```
// specification of a design template
Device myTemplate(Promoter, RBS, CDS, Terminator);
// enumeration of all designs
allDesigns = product(myTemplate);
// export to SBOL
SBOL.export(allDesigns, "./sbol/allDesigns.sbol");
```

3.6 SBOL Visual Compliant Design Visualization — `SBOL.visualize`

SBOL Visual is community-driven effort to standardize symbols for a visual representation of biological systems.[5]. Eugene supports the SBOL Visual standard by utilizing the language-based Pigeon toolkit[16] and, hence, generates an image that visualizes a design using the SBOL Visual glyphs.

Eugene v2.0 provides the `SBOL.visualize` statement which takes as input two arguments: The required first argument specifies the Eugene element that should be supported. The optional second argument specifies the name of the image file. If the

[5] http://www.sbolstandard.org/visual
[6] http://www.pigeoncad.org

second argument is not specified, then Eugene generates a randomly named image file and stores it into the `./exports/pigeon/` directory.

Eugene v2.0 only supports the visualization of `Device` elements that can be contained in a Eugene `Collection`. That is, if a part should be visualized, then the part should be wrapped by a `Device` element

For every part, Eugene tries to figure out the corresponding SBOL Visual symbol and Pigeon statement automatically and uses a random coloring. If Eugene cannot detect the type of the part, then it utilizes the SBOL Visual *user defined* symbol, which is specified by a `'?'` in the Pigeon language. Hence, every `Part` in Eugene v2.0 has a predefined `PIGEON` property which enables to customize the design visualization. If the `PIGEON` property value of a part is empty, then Eugene tries to figure out the Pigeon statement automatically. Otherwise, Eugene utilizes the specified value of the `PIGEON` property that must contain a well-formed Pigeon statement. Eugene v2.0 does not perform any sort of checking if the `PIGEON` property value is a well-formed Pigeon statement.

Examples

- Visualize the `p1` promoter by wrapping it into a `Device`

```
// specification of the p1 promoter and its Pigeon statement
PartType Promoter;
Promoter p1(.PIGEON("p p1 14"));
// wrap the p1 promoter into a Device for visualization
Device p1Wrapper(p1);
// visualize the device and name the image
SBOL.visualize(p1Wrapper, "./p1.png");
```

- Enumerate rule-compliant designs and visualize them

```
// specification of a design template
Device myDesign(Promoter, RBS, CDS, Terminator);
// specification of a rule
Rule forwardOrientation(ON myDesign:
    all_forward);
// enumerate all designs
designs = product(myDesign);
// visualize the enumerated designs
SBOL.visualize(designs);
```

Chapter 4

Rules and Constraints

Eugene v2.0 supports two categories of constraints:

- **Selection Constraints** ensure or restrict that only biological components with required characteristics occur in a design (Section 4.2).
- **Structural Constraints** ensure or restrict that biological components only occur (1) a required number of times, (2) at required relative or absolute positions, and (3) face a required orientation (Section 4.3).

4.1 Specification of Constraints in Eugene v2.0

4.1.1 Design Templates — `Device`

describe the semantics of the + and - operators here!

4.1.2 Constraints on Design Templates — `Rule`

4.2 Selection Constraints

Examples:

- Select all promoters whose strength is greater than 5:
  `Promoter.strength > 5`
- Select all pairs of repressors and promoters that have the following relation:
  `Repressor.repress == Promoter.name`
- Select all pairs of promoters and RBSs that have following relation:
  `Promoter.sequence.size + RBS.sequence.size <= 500`
- Select all devices on that the following equation holds **true**.
  `Device[0].sequence.size <= Device[1].sequence.size`

- find and provide more examples

4.3 Structural Constraints

Counting Constraints	**to constrain the number of occurrences of components.**	
CONTAINS α	α must occur at least once.	CONTAINS *p1*
α MORETHAN k	α must occur more than k times.	*p1* MORETHAN *0*
α EXACTLY k	α must occur exactly k times.	*p1* EXACTLY *1*
α SAME_COUNT β	α must occur as many times as β.	*p1* SAME_COUNT *p2*
Pairing Constraints	**to constrain the pair-wise occurrences of components.**	
α WITH β	Both components (α and β) must occur.	*p1* WITH *p2*
α THEN β	If α does occur, then β must occur.	*c2* THEN *GFP*
Positioning Constraints	**to constrain the (relative and absolute) positions of components.**	
STARTSWITH α	α must occur at the first position.	STARTSWITH *p1*
ENDSWITH α	α must occur at the last position.	ENDSWITH *GFP*
[k] EQUALS α	α must occur at the k-th position.	[0] EQUALS *p1*
[k] EQUALS [l]	The names of the components at position k and l must be equal.	[0] EQUALS [4]
α ALL_BEFORE β	If α and β occur, then all α must occur at any position BEFORE the first occurrence of β.	*p1* ALL_BEFORE *p2*
α BEFORE β	(same as α ALL_BEFORE β)	*c1* BEFORE *GFP*
α SOME_BEFORE β	If α and β occur, then at least one α must occur at any position BEFORE the first occurrence of β.	*p1* SOME_BEFORE *p2*
α ALL_AFTER β	If α and β occur, then all α must occur at any position AFTER the last occurrence of β.	*p2* ALL_AFTER *p1*
α AFTER β	(same as α ALL_AFTER β)	*GFP* AFTER *c2*
α SOME_AFTER β	If α and β occur, then at least one α must be at any position AFTER the last occurrence of β.	*c2* SOME_AFTER *c2*
α ALL_NEXTTO β	If α and β occur, then all α must occur immediately NEXT TO (either left or right) all occurrences of β.	*p1* ALL_NEXTTO *c2*
α NEXTTO β	(same as α ALL_NEXTTO β)	*p2* NEXTTO *c1*
α SOME_NEXTTO β	If α and β occur, then at least one α must occur immediately NEXT TO (either left or right) at least one occurrence of β.	*GFP* SOME_NEXTTO *c1*
Orientation Constraints	**to constrain the direction/orientation of the components.**	
ALL_FORWARD	all components in the designs must be forward oriented.	ALL_FORWARD
ALL_REVERSE	all components in the design must be reverse oriented.	ALL_REVERSE
ALTERNATE_ORIENTATION	the orientation of the components must alternate.	ALTERNATE_ORIENTATION
ALL_FORWARD α	All occurrences of α must be forward oriented.	ALL_FORWARD *p2*
FORWARD α	(same as ALL_FORWARD α)	FORWARD *c1*
SOME_FORWARD α	At least one occurrence of α must be forward oriented.	SOME_FORWARD *GFP*
ALL_REVERSE α	All occurrences of α must be reverse oriented.	ALL_REVERSE *p1*
REVERSE α	(same as ALL_REVERSE α)	REVERSE *c2*
SOME_REVERSE α	At least one occurrence of α must be reverse oriented.	REVERSE *c2*
α SAME_ORIENTATION β	All occurrences of α must have the same orientation as all β	*p2* SAME_ORIENTATION *c2*
Interaction Constraints	**to constrain and/or specify interactions among the components.**	
α INDUCES β	α induces β.	*in1* INDUCES *pIn1*
α DRIVES β	α drives the expression of β, that is α and β must have the some orientation, α must occur upstream to β, and there must be no terminator between α and β.	*p1* DRIVES *c2*
α REPRESSES β	α represses β.	*c2* REPRESSES *p2*

Table 4.1: The provided constraints in Eugene v2.0[4]

4.4 Logical Composition of Constraints

In Eugene v2.0, rules are logical compositions of constraints. The supported logical operators are: `AND`, `OR`, and `NOT`.

4.5 Enumeration of Rule-compliant Designs

In Eugene, rules are conditions which can only take the values **true** or **false**.

Chapter 5

Imperative Language Features

Imperative languages features serve to specify *HOW* design activities should be executed in what order and under what conditions.

Eugene v2.0 provides the following imperative language features:

1. **Control-Flow Statement** serve to specify the flow of — eventually repetitive — design activities based on conditions (Section 5.2).
2. **Library Generation** features provide means to automate the generation and/or import of libraries of biological elements (Section 5.5).
3. **Rule Builders** enable to automate the specification and logical composition of *Rules* and *Constraints* depending on the library's elements and/or conditions (Section 5.6).
4. **Containers** are groups of biological components, making it possible to process them in a uniform way (Section 5.3).
5. **Built-In Functions** are functions provided in Eugene v2.0 in order to perform commonly required tasks in a more efficient way (Section 5.4).

5.1 Eugene v2.0 Scoping

Before explaining the imperative language features of Eugene v2.0, we cover an important topic that all three imperative features have in common, namely **Scopes** of variables and biological components.

describe scoping here!

5.2 Control-Flow Statements in Eugene v2.0

Eugene v2.0 supports to following types of statements to control the flow of design activities:

- **Conditional Branches** control the flow of statement execution depending on conditions (Section 5.2.2).
- **Loops** enable to (1) iterate over enumerated designs of Eugene's declarative features and/or (2) to successively execute a sequence of statements while a certain condition is satisfied (Section 5.2.3).
- **Functions** represent user-defined design activities and group a sequence of Eugene v2.0 statements into reusable modules whose execution can be invoked when required and/or based on conditions (Section 5.2.4).

5.2.1 Conditions in Eugene v2.0

Control-Flow statements enable to control the execution of statements based on logical conditions. Hence, before learning the Eugene v2.0 control-flow statements, we first must define the specification of logical conditions in Eugene v2.0.

In Eugene v2.0, a *logical condition* consists of three elements: (1) a left-hand-side (*LHS*) operand, (2) a *Comparison Operator*, and (3) a right-hand-side (*RHS*) operand. The LHS and RHS operands must be either a constant or a primitive variable (Section 1.5.1) of type `num`, `txt`, or `bool`. Eugene v2.0 supports the following comparison operators:

```
<   ... less than
<=  ... less than or equals
==  ... equals
!=  ... not equals
>=  ... greater than or equals
>   ... greater than
```

Eugene v2.0 does not require that the LHS and RHS operands are of the same type. Here is an example of a Eugene v2.0 `IF-ELSE` conditional branch (Section 5.2.2), that always results in the execution of the `ELSE` branch since Eugene v2.0 evaluates the condition `n<"ACGT"` as `FALSE`, regardless of the comparison operator.

```
num n = 1;
if ( n < "ACGT" )
{
    println("TRUE");
}
else
{
    println("FALSE");
}
```

Note!

- Eugene v2.0 does not support the logical negation ¬ ("not") operator. Hence, it is the user's responsibility to properly change the comparison operator in order to specify a logical negation.
- Eugene v2.0 supports a primitive specification of logical compositions of conditions using the logical ∧ ("and") and ∨ ("or") operators. The ∧ operator has a higher precedence than the ∨ operator. In Eugene v2.0 syntax, the logical ∧ operator can be specified in the following ways: (1) natural language like lower- or upper-case `and` (`AND`), (2) logic like syntax /\, and (3) programming language like syntax `&&`. Equivalently, the logical ∨ operator can be specified as: (1) natural language like lower- or upper-case `or` (`OR`), (2) logic like syntax \/, and (3) programming language like syntax ||.
- Eugene v2.0 does not support the specification of parenthesis, which enable the specification of the evaluation order of the ∧ and ∨ operators. Hence, it is the user's responsibility to proper specify the required logical composition using, for example, transformation rules, such as De Morgan's laws, and associative or distributive properties.

5.2.2 Conditional Branches — `IF-ELSE`

Conditional Branches enable the execution of different statements depending on conditions. Eugene v2.0 supports `IF-ELSE` conditional branches.

The `IF` statement evaluates a logical condition that is specified within parenthesis (`()`). If the condition is satisfied, then the statements specified in the `IF` branch are executed. Otherwise, the statements specified in the `ELSE` branch are executed. The specification of an `ELSE` branch is optional in Eugene v2.0. That is, an `IF` statement does not necessarily require a corresponding `ELSE` branch.

Example: Print the smallest of two numbers:

```
// define two numbers and initialize them
num n1 = 1;
num n2 = 2;
IF ( n1 <= n2 ) // IF condition
{
    // IF branch
    println(n1);
}
ELSE
{
    // ELSE branch
    println(n2);
```

```
}
```

In addition, Eugene v2.0 supports the specification of zero of more `ELSEIF` conditions and branches. That is, if the `IF` condition is not satisfied, then the `ELSEIF` conditions are evaluated in sequential order. If an `ELSEIF` condition is satisfied, then the corresponding `ELSEIF` branch is executed and all remaining `ELSEIF` conditions are being ignored. If the `IF` condition and all `ELSEIF` conditions are not satisfied, then — if specified — the `ELSE` branch is executed.

Example: Print the relation ($<$, $==$, $>$) between two numbers.

```
num n1 = 1;
num n2 = 2;
IF ( n1<n2 )          // IF condition
{
    // IF branch
    println(n1, "<", n2);
}
ELSEIF ( n1 == n2 )   // ELSEIF condition
{
    // ELSEIF Branch
    println(n1, "==", n2);
}
ELSE
{
    // ELSE branch
    println(n1, ">", n2);
}
```

5.2.3 Loops — `WHILE`, `FOR`

Loops enable to control how many times a statement or a sequence of statements must be performed successively. The number of times can be controlled through the specification of a condition.

`WHILE` loops

Eugene v2.0 supports `WHILE` loops that execute a sequence of operations while a condition is satisfied, i.e. `true`.

Examples:

- Using a `WHILE` loop to print the numbers from `1` to `10` to the output console:

```
num i = 1;
while(i <= 10)
{
    println(i);
    i = i + 1;
}
```

In this example, we declare a numeric variable `i` and assign it the value `1`. The `while` loop executes the statements `println(i);` and `i=i+1;` in succession until the condition `i<=10` is not satisfied.

`FOR` loops

`FOR` loops, which are also supported by Eugene v2.0, represent an extension to `WHILE` loops, since `FOR` loops combine the declaration and initialization statements of the loop iteration variable `i` (`num i=1`), the condition while the sequence of statements should be executed (`i<=10`), as well as the adjustment of the loop iteration variable (`i=i+1`) after each execution of loop's body (`println(i);`). Those three statements must be separated by a semicolon (';'). The `WHILE` loop example from above looks as follows when expressed using a `FOR` loop.

Examples:

- Using a `FOR` loop to print the numbers from `1` to `10` in ascending order to the output console:

  ```
  for( num i=0 ; i <= 10; i = i + 1)
  {
      println(i);
  }
  ```

  The `FOR` loop combines (1) the declaration a numeric variable `i` and its initialization with the value `1`, (2) the condition `i<=10` that must be satisfied in order to execute the loop body (i.e. `println(i);`), and (3) the incrementation statement of the iteration variable `i` loop after the execution of the loop body.

- Print the numbers from `1` to `10` in descending order to the output console:

  ```
  for( num i=10 ; i >= 1; i = i - 1)
  {
      println(i);
  }
  ```

Note!
Eugene v2.0 does not perform any checks regarding the correctness of the loops statements. For example, if a loop's iteration variable `i` is not properly incremented or decremented after the execution of the loop's body, then the loop may not be executed as desired. That is, it is the responsibility of the Eugene v2.0 user to specify the correct behavior.

5.2.4 Function Prototyping

Defining a Function in Eugene v2.0

A *Function Prototype* is a non-empty sequence of statements that are executed in a sequential order. A *Function Defintion* is the specification of a function prototype. In Eugene v2.0, the definition of a function prototype has the following required and optional characteristics:

1. An optional **function return type** specifies the type of the value that a function returns to its caller. Eugene v2.0 supports only primitive types as function return types. Also, a function can return values to its caller regardless if a function return type is specified or not. To return values to the function caller, Eugene v2.0 provides the `return` statement.

2. A required **function name**, which identifies the function. Eugene v2.0 has the following restrictions on naming a function:

   - Every function must have a unique *Identifier* (Section 1.3).
   - *Reserved Words* cannot be utilized as function names (Section 1.4) — either lower- or upper-case.
   - A function cannot have the same name as a Eugene v2.0 *Built-In Function* (Section 5.4) — either lower- or upper-case.

3. An optional list of typed **function parameters** that allow to pass specific values to the function.

4. A required **function body** that contains the non-empty sequence of the function's statements which describe the function's behavior.

Examples

- Definition of the `HelloWorld` function that outputs the string `Hello World` to the console. The `HelloWorld` function has no return type and takes zero parameters.

```
HelloWorld()
{
    println("Hello World");
}
```

Note!

- If a function does not take any parameters, then parenthesis (`()`) must be specified after the function name.
- A function prototype can be specified anywhere in a Eugene script.

- Definition of an `output` function, which takes as argument a string that should be printed to the output console.

```
output ( txt output_string )
{
    println (output_string);
}
```

Note!

- Eugene v2.0 requires the specification of the type of each parameter.
- If a function parameter has the same name than a global variable, then the function parameter has higher priority then the global variable.

```
num a = 5;

myFunction(num a)
{
  a = 2;
  return a;
}

b = myFunction(a);
println("a: ", a);
println("b: ", b);
```

- Definition of a `add` function, which takes two numbers `n1` and `n2` as parameters and returns a number that is the sum of two arguments.

```
num add ( num n1, num n2 )
{
    num sum = n1 + n2;
    return sum;
}
```

Note!

- If a function takes more than one parameter, then Eugene v2.0 requires the specification of the parameters list as a comma-separated list.

Calling a Function in Eugene v2.0

Besides the specification of a *Function Definition*, Eugene v2.0 supports *Function Calls* that enable the execution of functions in the executing a Eugene script. In Eugene v2.0, a function call requires (1) the specification of the **function name** and (2) the specification of the **parameter values** that are being passed to the function. The number and the types of the parameter values must match the number and types of the parameters specified in the function definition. Optionally, if the function returns a value, then the returned value can be assigned to a variable.

Examples

- Calling the `HelloWorld` function that has no parameters and does not return a value.

  ```
  HelloWorld();
  ```

  Note!

  - When calling a function without parameters in Eugene v2.0, then the parenthesis (`()`) must be specified after the function's name.
  - In Eugene v2.0, a semi-colon (`;`) ends a function call statement.

- Calling the `add` function and assign the returned sum to a variable

  ```
  num one = 1;
  sum = add(one, 2);
  ```

- Nesting function calls

  ```
  sum = add( add(1, 2), 3);
  ```

Recursive Functions in Eugene v2.0

Eugene v2.0 supports *recursive functions*. A recursive function f is a function that contains at least one function call statement which calls itself, i.e., the function f.

Examples

- Print the number from 1 to 5 in descending order.

  ```
  printNumbers(num n)
  {
      if(n > 0)     // Breaking-Condition of recursion
      {
  ```

```
        // output of the current number
        println(n);

        // "Recursion"
        // call the same function with the next number to print
        printNumbers(n - 1);
    }
}

// call the function
printNumbers(5);
```

- In programming languages, a prominent example to demonstrate recursion is the definition of a recursive function that outputs the *Fibonacci Numbers*. The Fibonacci numbers are a sequence of numbers where each number is the sum of its two predecessors, i.e., 1, 1, 2 (=1+1), 3 (=1+2), 5 (=2+3) and so on. The following recursive function — specified in Eugene v2.0 — calculates the sum of all Fibonacci numbers up to a given number n.

```
num Fibonacci(num n)
{
    if ( n == 0 || n == 1 )
    {
        return 1;
    }
    else
    {
        sum = Fibonacci(n-1) + Fibonacci(n-2);
        return sum;
    }
}

// call the recursive function
Fibonacci(13);
```

Note!
Every recursive function requires a *breaking-condition* which determines when to stop the recursion. Eugene v2.0 does not perform any analysis that checks if a proper *breaking-condition* is specified.

5.3 Containers

Eugene v2.0 provides two types of containers for biological facts and rules:

- **Collections** are unordered groups of biological data with no duplicate elements.
- **Arrays** are ordered groups of biological data.

Eugene v2.0 supports nested containers. That is, a collection can itself contain collections and/or arrays. Also, arrays can contain other arrays and/or collections.

5.3.1 Collections

Declaring and Defining Collections

In Eugene v2.0, the definition of a collection starts with the `Collection` keyword followed by the collection's name, a comma-separated list (`','`) of its elements in parenthesis (`()`), and a closing semi-colon (`';'`).

Examples

- Two alternatives of declaring an empty collection

```
Collection empty1();
Collection empty2;
```

- Defining a collection of variables that are defined **within** the scope of the collection.

```
Collection variables
(
    num var1 = 1,
    num var2 = 2
);
```

- Defining a collection of variables that are defined **outside** the scope of the collection.

```
num var1 = 1;
num var2 = 2;
Collection variables (var1, var2);
```

- Defining a collection of collections.

```
num var1 = 1;
num var2 = 2;
Collection variables12 (var1, var2);

num var3 = 3;
```

```
num var4 = 4;
Collection variables34 (var3, var4);

Collection collectionVariables (
    variables12, variables34
);
```

Accessing the Elements of a Collection

In Eugene v2.0, the elements of a collection can be accessed using the dot-notation. That is, the name of the collection must be specified, followed by the dot character ('.') and the name of the desired element.

Examples

- Console output of a collection's elements.

  ```
  Collection variables
  (
      num var1 = 1,
      num var2 = 2
  );
  // accessing the var1 element of the variables collection
  println(variables.var1);
  // accessing the var1 element of the variables collection
  println(variables.var2);
  ```

  The `println(variables.var1)` statement demonstrates that the variables `var1` and `var2` are accessible only via the scope of the `variables` collection. If the scope is not specified, then Eugene v2.0 reports an error.

- Accessing the variables that are defined within and outside the scope of the collection.

  ```
  num var1 = 1;
  num var2 = 2;
  Collection variables
  (
      num var1 = 11,
      num var2 = 22
  );

  // accessing the var1 variable
  ```

```
println(var1);

// accessing the var1 variable of the variables collection
println(variables.var1);
```

The `println(var1)` statement demonstrates that the variables `var1` and `var2` are accessible outside the scope of the `variables` collection.

Note!

- The elements of a Eugene v2.0 `Collection` cannot be accessed by an index since collections are an **unordered** set of elements. If collection elements are being accessed with the index-based notation (`[]`), then an exception is reported.
- Unfortunately, Eugene v2.0 does not support to iterate over the elements of collection.

Assigning values to the Elements of a Collection

In Eugene v2.0, values can be assigned to the elements of a collection after the collection's definition and initialization. On the left-hand-side (LHS) of the assignment (Section 1.6) the desired collection element must be accessed. Then, the new value of the element must be specified on the right-hand-side (RHS) of the assignment. The new value can either be a constant, an expression, or an element. When interpreting the assignment, then Eugene v2.0 compares the types of the LHS and RHS elements. If the types match, then the assignment is executed. Otherwise an exception is reported.

Examples

- Assigning a new value to the elements of a collection using the assignment `=` operator.

```
Collection variables
(
    num var1 = 1,
    num var2 = 1
);

// assign a new value to the var2 variable
variables.var2 = 2;

// output the new value of the var2 variable of
// the variables collection
println(variables.var2);
```

- Defining a collection of a variable that was defined and initialized outside the scope of the collection. To demonstrate the copying of the variables into a collection, we assign new values to the collection's variables and output the values.

```
num var1 = 1;
Collection variables ( var1 );

// output the current values
println(var1);
println(variables.var1);

// assign new values
variables.var1 = 11;

// output the new values
println(var1);
println(variables.var1);
```

Adding new elements to a Collection

In Eugene v2.0, elements can be added to an existing collection using the plus operator ('+'). That is, besides the semantics of the plus operator as described in Section 1.7.1, the plus operator has the semantics of a set *union* when being utilized at collections.

Examples

- Adding a variable to a collection using the + operator.

```
num var1 = 1;
Collection variables
(
    num var2 = 2
);

// add the var1 variable to
// the variables collection
// using the + operator
variables = variables + var1;

// output the newly added var1 variable of
// the variables collection
println(variables.var1);
```

Note!

- To further process the resulting collection, it should be assigned to a collection which could be either a new collection or the modified one.

```
// assign the resulting collection to
// the modified variables collection
variables = variables + var1;
```

or

```
// assign the resulting collection to
// the new new_variables collection
new_variables = variables + var1;
```

- When adding a collection c_2 (and/or array) to a collection c_1, then all the elements of the c_2 collection/array are added to the collection c_1.

```
Collection c1 ( ... );
Collection c2 ( ... );

// nesting the c2 collection into the c1 collection
// and assigning the resulting collection to c1
c1 = c1 + c2;
```

Removing elements from a Collection

In Eugene v2.0, elements can be removed from an collection using the minus operator ('-'). That is, besides the semantics of the minus operator as described in Section 1.7.1, the minus operator has the semantics of a set *intersection* when being utilized on collections.

Examples

- Removing one element from a collection using the - operator.

```
Collection variables
(
    num var1 = 1,
    num var2 = 2
);

// remove the var2 variable from the variables collection
// using the - operator
variables = variables - variables.var2;
println(variables);
```

Note!
When removing elements from a collection, then the elements must be accessed using the dot operator (`'.'`). If the specified element does not exist in the collection, then an exception is reported.

5.3.2 Arrays

An array is similar to a collection. However, the elements in an array are ordered, making it possible (1) to store the same element multiple times in an array and (2) to access an array's elements by an index.

Declaring and Defining Arrays

In Eugene v2.0, the definition of an array starts with the `Array` keyword followed by the array's name, a comma-separated list (`','`) of its elements in parenthesis (`()`), and a closing semi-colon (`';'`).

Examples

- Two alternatives of declaring an empty array

```
Array empty1();
Array empty2;
```

- Defining an array of variables that are defined **within** the scope of the array.

```
Array variables
(
    num var1 = 1,
    num var2 = 2
);
```

- Defining an array of variables that are defined **outside** the scope of the array.

```
num var1 = 1;
num var2 = 2;
Array variables (var1, var2, var1);
```

- Defining an array of collections.

```
num var1 = 1;
num var2 = 2;
Collection variables12 (var1, var2);
```

```
num var3 = 3;
num var4 = 4;
Collection variables34 (var3, var4);

Array arrayOfCollections
(
    variables12, variables34
);
```

Accessing the Elements of an Array

In Eugene v2.0, the elements of an array can be accessed using the dot-notation or the index of the desired element.

- Using the **dot-notation** to access array elements
  When using the dot-notation, then the name of the array must be specified, followed by the dot character (`'.'`) and the name of the desired element. If the accessed element occurs multiple times, then its first occurrence is returned.

- Using the **Index-based notation** access of array elements
  Eugene v2.0 utilizes zero-based indices. That is, the first element of an array has index zero (0). When using the index-based access method, then the name of the array must be specified, followed by the index between squared brackets (`[]`), such as `<array-name> [ <index> ]`. In Eugene v2.0, the index can also be specified either as constant, a variable, or an expression. However, if the specified index is negative (i.e., < 0) or greater than equals the length of the array, then an error is reported.

Examples

- Using the dot-notation to output of the array's elements to the console.

```
Array variables
(
    num var1 = 1,
    num var2 = 2
);

// output of the array's var1 element
println(variables.var1);
// output of the array's var2 element
println(variables.var2);
```

- Index-based access to output of the array's elements to the console.

```
Array variables
(
    num var1 = 1,
    num var2 = 2
);

// output of the first element, i.e. index 0
println(variables[0]);
// output of the second element, i.e. index 1
println(variables[1]);
```

- Console output of the array elements, using a `FOR` loop (Section **??**) and the `sizeof` function (Section 5.4) to iterate over the array.

```
Array variables
(
    num var1 = 1,
    num var2 = 2
);

// using a FOR loop and the SIZEOF function to
// iterate over the array's elements
for(num i=0; i<sizeof(variables); i=i+1)
{
    // console ouput of the i-th element
    println(variables[i]);
}
```

Note!
When defining an array's elements outside the scope of the array, then the array will contain copies of the elements. That is, assigning new values to the elements outside the array's scope does not change the values of the elements of the array. The following Eugene script will output the values `1, 1, 3, 1`.

```
num var1 = 1;
num var2 = 2;
Array variables (var1, var2);

// accessing the var1 variable
println(var1);
// accessing the var1 variable of the variables collection
println(variables.var1);

// assign a new value to var1
```

```
var1 = 3;

// accessing the var1 variable
println(var1);
// accessing the var1 variable of the variables collection
println(variables.var1);
```

Assigning values to the Elements of a Collection

In Eugene v2.0, values can be assigned to the elements of an after the array's definition and initialization. On the left-hand-side (LHS) of the assignment (Section 1.6) the desired array element must be accessed. This can either be done using the **dot-notation** or by specifying the elements **index**. When using the **dot-notation** and if multiple elements with the same exist in the array, then the new value is assigned to the element's first occurrence. The new value must be specified on the right-hand-side (RHS) of the assignment. The new value can either be a constant, an expression, or an element. When interpreting the assignment, then Eugene v2.0 compares the types of the LHS and RHS elements. If the types match, then the assignment is executed. Otherwise an exception is reported.

Example
In this example, we assign new values to the array elements using the **dot-** and **index-based** notation.

```
Array variables
(
    num var1 = 1,
    num var2 = 2
);

// using the index-based notation to assign a new
// value to the first element (i.e., var1) of the array
variables[0] = 11;

// using the dot-notation to assign a new value
// to the var2 variable
variables.var2 = 22;

// ouput the new values of the array's elements
println(variables[0]);
println(variables.var2);
```

Adding Elements to an Array

In Eugene v2.0, elements can be added to an existing array using the plus operator ('+'). That is, besides the semantics of the plus operator as described in Section 1.7.1, the plus operator has the semantics of appending elements to an array.

Examples

- Adding a variable to a array using the `+` operator.

```
num var1 = 1;
Array variables
(
    num var2 = 2
);

// add the var1 variable to
// the variables array
// using the + operator
variables = variables + var1;

// output the variables array
println(variables);
```

Note!

- To further process the resulting collection, it should be assigned to a collection which could be either a new collection or the modified one.

```
// assign the resulting collection to
// the modified variables collection
variables = variables + var1;
```

or

```
// assign the resulting collection to
// the new new_variables collection
new_variables = variables + var1;
```

- When adding an array a_2 (and/or collection) to an array a_1, then all the elements of the a_2 array/collection are appended to the array c_1.

```
Array a1 ( ... );
Array a2 ( ... );

// append a2's elements to the a1 array
// and assign the resulting array to a1
a1 = a1 + a2;
```

Removing Elements from an Array

In Eugene v2.0, elements can be removed from an array using the minus operator (`'-'`). That is, besides the semantics of the minus operator as described in Section 1.7.1, the minus operator has the semantics of a set *intersection* when being utilized on arrays.

Examples

- Removing one element from an array using the `-` operator and the dot-notation.

```
Array variables
(
    num var1 = 1,
    num var2 = 2
);

// remove the var2 variable from the variables collection
// using the - operator
variables = variables - variables.var2;
```

- Removing one element from an array using the `-` operator and the index-based notation.

```
Array variables
(
    num var1 = 1,
    num var2 = 2
);

// remove the var2 variable from the variables collection
// using the - operator
variables = variables - variables[1];
```

Note!
As exemplified, elements can be removed from an array using either the dot operator (`'.'`) or the index operator (`[]`). When using the dot operator and if the specified

element does not exist in the array, then an exception is reported. Eugene v2.0 uses a zero-based indexing mechanism, i.e. the first element of an array has index 0. Hence, when using the index operator and if the index is out of the array's bounds, then an exception is reported.

5.3.3 Assorted Example on utilizing Eugene Containers

Here, we show a Eugene v2.0 script that enumerates all forward and reverse oriented `Promoter` and `CodingSequence` pairs. In both orientation cases, a `Promoter` should occur upstream of the `CodingSequence` element. In this example, we first define a library of `Promoter` and `CodingSequence` parts. Then, we utilize Eugene v2.0 *Design Templates* (Section 4.4 to specify the structure of the forward and reverse oriented designs. We also define rules on the design templates, ensuring that all elements are forward or reverse oriented in the forward and reverse designs, respectively. After utilizing `product` *Built-In Function* (Section 5.4), we utilize the `+` operator to union forward and reverse oriented design. Lastly, we print all designs to the console.

```
// specification of part types
PartType Promoter;
PartType CodingSequence;

// specification of parts
Promoter p1;
Promoter p2;
CodingSequence cds1;
CodingSequence cds2;

// design template and rule for forward oriented designs
Device fd(Promoter, CodingSequence);
Rule rfd(ON fd: all_forward);

// design template and rule for reverse oriented designs
Device rd(CodingSequence, Promoter);
Rule rrd(ON rd: all_reverse);

// enumerate forward and reverse oriented designs
fds = product(fd);
rds = product(rd);

// union the forward oriented and reverse oriented designs
devices = fds + rds;

// print all designs to the console
println(devices);
```

5.4 Built-in Functions

Eugene v2.0 provides a set of functions that can are particularly helpful in the imperative design of biological systems. The functions can either be called using upper-case as well as lower-case characters, such as `sizeof` and `SIZEOF`.

- `print`, `PRINT`
  The `print` function takes as one parameter as input, interprets the input parameter, and prints the interpretation result onto the output console. The input parameter can either be a static string, and identifier, or a string concatenation (using the '`,`' character) consisting of static strings and identifiers.

  Examples:

  - The statement `print("ACGT");` outputs the static string "ACGT".
  - The statement `print("A", "C", "G", T");` outputs also the string "ACGT", which is, in this case, a concatenation of the four "A", "C", "G", and "T" characters.
  - Assume the four string variable `a`, `t`, `c`, and `g` contain respectively the string values "A", "C", "G", and "T". Then, the statement `print(a, c, g, t);` also outputs the string "ACGT", which is the concatenation of the values of the four string variables.

- `println`, `PRINTLN`
  The `println` statement is equivalently to the `print` statement plus it outputs a line feed. The `println` function takes as one parameter as input, interprets the input parameter, and prints the interpretation result onto the output console. The `println` function also

- `sizeof`, `SIZEOF`
  The `sizeof` function takes as input an identifier (`id`) and returns the size of the corresponding element.

  Examples:

```
// first, we declare an array and initialize it with
// the numbers 1, 2, 3, 4
num[] array = [1, 2, 3, 4];

// next, we utilize the sizeof function to determine
// the size of the array
num array_size = sizeof(array);
```

```
// lastly, we output the a concatenated string
// including the size of the array
println("the size of the array ", array, " is ", array_size);
```

- `sequence_of`, `SEQUENCE_OF`
  The `sequence_of` function takes as input an identifier (`id`) and returns the sequence of the corresponding element only if the identifier corresponds to a biological component. That is, either a part or a device.

  Examples:

  – Output of a part's sequence.

  ```
  // first, we declare a part type
  PartType Promoter;

  // second, we create a promoter part
  Promoter p1(.SEQUENCE("ACGT"));

  // next, we output the p1 promoter's sequence
  // using the sequence_of function
  println(sequence_of(p1));
  ```

  – Output of a device's sequence consisting of only forward oriented parts.

  ```
  // first, we declare the part types
  PartType Promoter;
  PartType CodingSequence;

  // second, we create some parts
  Promoter p1(.SEQUENCE("ACGT"));
  CodingSequence cds1(.SEQUENCE("ATG"));

  // next, we define a device
  Device dev (p1, cds1);

  // lastly, we output the device's sequence
  println(sequence_of(dev));
  ```

  – Enumeration of device instances and calculation of their sequences.

  ```
  // first, we declare the part types
  PartType RBS;
  PartType CodingSequence;

  // second, we create some parts
  ```

```
RBS p1(.SEQUENCE("AAA"));
CodingSequence cds1(.SEQUENCE("ATG"));

// next, we define a design template
Device dev (
    [RBS | CodingSequence], [RBS | CodingSequence]);

// definition of a rule to ensure correct pairing,
// orientation, and positioning of the
// RBS-CodingSequence pairs
Rule ruleDev( ON dev:
    RBS WITH CodingSequence AND
    RBS SAME_ORIENTATION CodingSequence AND
    NOT FORWARD RBS AND RBS BEFORE CodingSequence AND
    NOT REVERSE RBS AND RBS AFTER CodingSequence);

// enumerate all devices
devices = product(dev);

// for every device, we output its sequence
for (num i=0; i<sizeof(devices); i=i+1)
{
    println(sequence_of(devices[i]));
}
```

Note!
The `sequence_of` function utilizes the values of the pre-defined `SEQUENCE` property in order to determine the sequence of biological component. That is, if the sequence of a part is not specified in the `SEQUENCE` property, then the `sequence_of` function is not able to determine the correct sequence.

- `random`, `RANDOM`
  The `random` function takes as input two numerical values (`lb, ub`) and returns a randomly generated number in the range from those two numbers (`[lb...ub]`). Hence, the first input parameter represents the lower bound (`lb`) and the second input parameter denotes the upper bound (`ub`) of the randomly generated number. For example, `random(0, 100)` will return a randomly generated number from `0` to `100` (inclusive). If the lower bound is greater then the upper bound (`lb > ub`), then the `random` function will throw an exception.

- `save`, `SAVE`
  Imperative language features — branches, loops, and function prototyping facilities — enable to automate the generation of library elements, such as parts.

However, Eugene v2.0 supports scopes, which avoids to save automatically generated library elements into the global library. Therefore, Eugene v2.0 provides the `save` built-in function, which receives as input the identifier (`id`) — this can also be a dynamic name (see Section 5.5) — and stores the element in the global library.

- `product`, `PRODUCT`
  describe product() here!

  coding: only devices that contain at least one part type sub-component are allowed!

- `permute`, `PERMUTE`
  describe permute() here!

- `flip`, `FLIP`
  describe flip() here!

- `orientation_of`, `ORIENTATION_OF`
  describe orientation_of() here!

- `query`, `query`

  The built-in `QUERY` function enables to retrieve a collection of parts from the library with desired property values. In Eugene v2.0, the `QUERY` function takes as argument a *Selection Constraint* (Section 4.2) and returns a `Collection` (Section 5.3.1) of all parts that satisfy the selection constraint. If no part satisfies the selection constraint, then an empty collection is returned.

  Examples

  – Query all parts from the library of type "Promoter" and assign them to the `promoters` collection.

```
// Definition of a type Property
Property type(txt);

// Definition of a SequencedPart PartType
// with the type Property
PartType SequencedPart(type);

// Specify some SequencePart
// and their type
SequencedPart gp1(.type("Promoter"));
SequencedPart gp2(.type("CDS"));
SequencedPart gp3(.type("Terminator"));
```

```
// Use the QUERY built-in function to
// retrieve all parts with type of Promoter
promoters = QUERY ( SequencedPart.type == "Promoter");
```

– Query all parts whose sequence starts with the start codon "ATG"

```
// Definition of a type Property
Property type(txt);

// Definition of a SequencedPart PartType
// with the type Property
PartType SequencedPart(type);

// Specify some SequencePart
// and their type
SequencedPart gp1(.type("Promoter"), .SEQUENCE("AAAA"));
SequencedPart gp2(.type("CDS"), .SEQUENCE("ATG"));
SequencedPart gp3(.type("Terminator"), .SEQUENCE("TATAA"));

// Use the QUERY built-in function to
// retrieve all parts with type of Promoter
codingSequences = QUERY (
    SequencedPart.SEQUENCE STARTSWITH "ATG");

// output of the coding sequences
println(codingSequences);
```

5.5 Dynamic Naming of Parts — The ${} Operator

Every part in Eugene v2.0 is an instance of a `PartType` (Section 2.1.2) and requires a unique `id` (Section 1.3). The part's `id` must be *statically* specified (i.e., when writing the Eugene v2.0 script), making it impossible to name a part, for example, based on the evaluation of conditions, i.e. the control-flow.

Therefore, Eugene v2.0 provides the ${} operator to name biological parts during the execution of a Eugene v2.0 script. The ${} requires the specification of an expression (Section 1.7) between the curly brackets {} and can be utilized wherever a part is defined and accessed.

Examples

- Definition of ten parts of type `Promoter` having the name p_1, p_2, ... p_{10}.

```
// Definition of the Promoter type
```

```
PartType Promoter;

// utilize a FOR loop to enumerate 10 Promoters
FOR(num i=1; i<=10; i=i+1)
{
    // utilize the ${} operator to
    // identify each promoter properly
    Promoter ${"p"+i};

    // save the promoters for utilization
    // outside of the scope of the FOR loop
    SAVE(${"p"+i});
}

// access individual Promoter parts and
// print them
println(p1);
println(p10);
```

5.6 Rule Builders — The `AND` Built-In Function

Eugene v2.0 supports the *Disjunctive Normal Form (DNF)* to define rules. That is, Eugene v2.0 builds (1) the logical disjunction ($\vee$) of all rules that are defined on the same design template (Section 4.4), and (2) the logical conjunction ($\wedge$) of the the constraints specified within one rule definition (Section 4.1.2).

To add constraints the a rule definition, Eugene v2.0 provides the `AND` built-in function. The `AND` function takes two arguments: (1) the `id` of a rule and (2) a constraint (Section 4.2 and Section 4.3), which can be either an atomic constraint or a logical disjunction of constraints. The `AND` function does not return a new rule. It builds the logical conjunction of the given rule and the constraint.

Example

- In this example, we build up a rule definition to ensure that the library promoters p_1 and p_2 occur in the design.

```
// Definition of a PartType
PartType Promoter;

// Definition of the Promoter library
Promoter p1;
Promoter p2;
```

```
// Definition of a Design Template
Device Promoters(Promoter, Promoter);

// Definition of a Rule on the Promoters design template
// but without any constraints
Rule containsPromoters(ON Promoters:);

// utilize the AND built-in function to ensure
// the p1 Promoter occurs in the design
AND(containsPromoters, CONTAINS p1);

// utilize the AND built-in function to ensure
// the p2 Promoter occurs in the design
AND(containsPromoters, CONTAINS p2);
```

- In the following example, we (1) generate a library of two promoters (p_1, p_2), (2) add the Promoter part type to a design template D, (3) ensure that each promoter occurs forward oriented in the design using the `AND` built-in function, (4) use the `PRODUCT` built-in function to enumerate all designs, and (5) iterate over the designs and print every design to the console.

```
// Declaration of the Promoter PartType
PartType Promoter;

// Declaration of a Design Template
Device D;

// Definition of a Rule on the design template D
// without any constraints
Rule R(ON D:);

FOR ( num i=1; i<=2; i=i+1)
{
    // add the Promoter part type to the
    // design template D
    D = D + Promoter;

    // instantiate a Promoter part
    Promoter ${"p"+i};

    // save the Promoter part in the library
    SAVE(${"p"+i});

    // utilize the AND built-in function to ensure
```

```
        // that the Promoter part occurs forward oriented
        AND(R, FORWARD ${"p"+i});
        AND(R, CONTAINS ${"p"+i});
    }

    // utilize the PRODUCT built-in function
    results = PRODUCT(D);

    // print every design
    FOR ( num i=0; i<SIZEOF(results); i=i+1)
    {
        println(results[i]);
    }
```

5.6.1 Assorted Example

Here, we exemplify the combination of the Eugene v2.0 **Design Templates**, **Dynamic Part Naming** and **Rule Builders**. In the example, we

1. generate a library of `Promoter` parts,
2. define a *Design Template* to specify the arrangement of the promoters in the design,
3. utilize the `AND` built-in function to concatenate rules in the disjunctive normal form to ensure that the `Promoter` parts occur in the design at the proper position and with the proper orientation,
4. utilize the `PRODUCT` built-in function to enumerate all designs, and
5. visualize the resulting designs using the `SBOL.visualize` statement.

```
// definition of the Promoter part type
PartType Promoter;

// specification of a Design Template
Device D(Promoter, Promoter);

// definition of two rules to ensure
// that all Promoter parts must be
// either forward or reverse oriented
Rule rbForward(ON D: all_forward);
Rule rbReverse(ON D: all_reverse);

// create instances of the Promoter part type and
```

```
// ensure that the instances appear correctly
// in the ultimate designs
for(num i=1; i<=SIZEOF(D); i=i+1) {

    // create a part of the Promoter part type
    // using Dynamic Naming
    Promoter ${"p_"+i}(.pigeon("p p " + i + " nl"));

    // store the part in the library
    SAVE(${"p_"+i});

    // ensure that the created part appears
    // in the design
    AND(rbForward, ${"p_"+i} EXACTLY 1);
    AND(rbReverse, ${"p_"+i} EXACTLY 1);

    if(i >= 2)
    {
        // ensure that the created part appears
        // after the previously created part
        AND(rbForward, ${"p_"+(i-1)} BEFORE ${"p_"+i});
        AND(rbReverse, ${"p_"+(i-1)} AFTER ${"p_"+i});
    }
}

// output the resulting rules
println(rbForward);
println(rbReverse);

// generate all rule-compliant designs
// of the device template D
lod = product(D);

// visualize the designs
SBOL.visualize(lod);
```

Bibliography

[1] Bhatia, S.; Densmore, D. *ACS Synthetic Biology* **2013**, *2*, 348–350.

[2] Galdzicki, M.; *et al.*, *Nature Biotechnology* **2014**,

[3] Galdzicki, M.; *et al.*, Synthetic Biology Open Language (SBOL) Version 1.1.0. http://hdl.handle.net/1721.1/73909.

[4] Oberortner, E.; Densmore, D. *ACS Synthetic Biology* **0**, *0*, null, PMID: 25426642.

Appendix A

SBOL Files