E U G E N E

A DOMAIN-SPECIFIC LANGUAGE FOR SYNTHETIC BIOLOGY

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

December 23, 2014

Technical Report No. ECE-2014-NN

Contents

1 Introduction to the Eugene Language **1**
- 1.1 The Motivation for Eugene from a Synthetic Biology Perspective . . . 1
- 1.2 The Language Characteristics of Eugene v2.0 from a Computer Science Perspective . . . 2
- 1.3 How to use and download Eugene v2.0? . . . 2

2 Language Basics **3**
- 2.1 Comments . . . 3
- 2.2 Primitive Values and Types . . . 3
- 2.3 Identifiers . . . 4
- 2.4 Reserved Words . . . 4
- 2.5 Variables . . . 4
  - 2.5.1 Primitive Variables . . . 5
  - 2.5.2 Arrays of Primitives . . . 5
- 2.6 Assignments . . . 5
- 2.7 Expressions . . . 6
  - 2.7.1 Semantics of the Expression Operators . . . 7
  - 2.7.2 Examples of Expressions: . . . 8
- 2.8 Error Reporting . . . 9

3 Declarative Language Features **10**
- 3.1 Declaration of Biological Facts . . . 10
  - 3.1.1 Properties of Biological Facts . . . 10
  - 3.1.2 Types of Components — Part Types . . . 11
  - 3.1.3 Instantiations of Types . . . 12
  - 3.1.4 Composite Biological Facts . . . 14
  - 3.1.5 Relations and Interactions among Biological Facts . . . 15
- 3.2 Containers for Biological Facts and Rules . . . 16
  - 3.2.1 Collections . . . 16
  - 3.2.2 Arrays . . . 16

4 Templates, Rules, and Constraints **17**
- 4.1 Design Templates . . . 18
- 4.2 Rules and Constraints . . . 19

4.2.1 Structural Constraints 19
4.2.2 Relational Expression Constraints 20

5 Data Exchange Capabilities **21**
5.1 Inclusion of Eugene Scripts 21
5.2 Data Import of Eugene Containers 22
5.3 Data Import from the iGEM Partsregistry 22
5.4 GenBank Data Import and Export 22
5.5 SBOL Data Import and Export 22
5.6 SBOL Visual Compliant Design Visualization 22

6 Imperative Language Features **23**
6.1 Eugene v2.0 Scoping 23
6.2 Conditional Branches 23
6.3 Loops 23
6.3.1 `WHILE` loops 24
6.3.2 `FOR` loops 24
6.4 Dynamic Naming of Genetic Parts 24
6.5 Function Prototyping 24
6.6 Built-in Functions 24

Chapter 1

Introduction to the Eugene Language

1.1 The Motivation for Eugene from a Synthetic Biology Perspective

first we provide some bio yadi yada about Eugene

combinatorial design paradigm ⇒ rule-based design paradigm ⇒ the specification of synthetic biological designs based on biological constraints and rules

- A built-in **Library Management System (LMS)** to organize biological components which can be used in the design of synthetic biological systems. The LMS provided by Eugene is also called "*Sparrow*".
- Manual specification of the biological components and their characteristics ("Properties").
- Support of **Data Exchange** standards, making it possible to automatically import existing biological data from data repositories (such as iGEM partsregistry) into the Sparrow LMS. The support of data exchange standards also enables to export design that are specified and enumerated in Eugene to data standard file formats, such as SBOL or GenBank.
- Rules to
  - **Query** biological components from the LMS
  - **Compose** biological components into more complex components
- **Conditional Branches** and **Loops** to control the flow of the design specification process based on certain events and conditions.
- **Function Prototyping** to group an ordered set of control-flow statements, enabling to specify complex control-flows in a more modular and reusable fashion.

1.2 The Language Characteristics of Eugene v2.0 from a Computer Science Perspective

here we provide the real important stuff of Eugene ⇒ CS rulez!

- multi-paradigm language ⇒ combines declarative and imperative language features
- declarative features ⇒ the user specifies *WHAT* the design problem is based on rules and constraints; invokes *built-in functions*
- imperative features ⇒ *HOW* to (1) process the results returned by Eugene's declarative features and/or (2) build the *WHAT* question to ask Eugene
- interpreted language ⇒ a given Eugene script is interpreted, i.e. it is executed "on-the-fly". That is every statement is executed immediately. The Eugene parser is a one-phase parser, which requires that variables, biological facts, and functions must be declared before being used and/or referred to them.
- strongly typed language ⇒ three primitive types (num, txt, bool), two types of arrays (num and txt), language model for typing the biological facts (primitive and composite).
- Scopes of Identifiers and Variables ⇒
- Global Library of Biological Primitives (i.e. Parts) and Composites (i.e. Devices) ⇒
- Global in-memory symbol tables which store identifiers of variables, functions, and biological components ⇒.

In this manual, we describe the syntax of the Eugene language using the Extended-Backus-Naur Form (EBNF).[1]

1.3 How to use and download Eugene v2.0?

- open-source ⇒ Java
- Java ARchive (JAR) file format ⇒
- XML-RPC web service
- Web application ⇒ EugeneLab

Chapter 2

Language Basics

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

2.1 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
*/
```

2.2 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.

2.3 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`

2.4 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

2.5 Variables

In Eugene v2.0, variables MUST have a type and a unique name (see Sections 2.2) and a unique identifier (see Section 2.3). 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 2.7).

2.5.1 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;`

2.5.2 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.

2.6 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
```

2.7 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 4.2.2 and Chapter 6 we explain the support for boolean expressions respectively relational expressions and logical expressions of Eugene v2.0.

2.7.1 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.

2.7.2 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 6.6).

- 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);
```

2.8 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 3

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 5. 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 6.6.

3.1 Declaration of Biological 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.

3.1.1 Properties of Biological Facts

Properties represent attributes and characteristics of primitive biological components, such as parts. Each property is defined with the `Property` keyword, a unique identi-

fier (see Section 2.3), 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[]);
```

3.1.2 Types of Components — Part Types

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 3.1.1).

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`.

3.1.3 Instantiations of Types

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 3.1.2). 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.[2]

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"));
```

3.1.4 Composite Biological Facts

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);
  ```

3.1.5 Relations and Interactions among Biological Facts

3.2 Containers for Biological Facts and Rules

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

- **Collections** are unordered groups of biological data.
- **Arrays** are ordered groups of biological data.

Eugene v2.0 also supports nested container types. That is, collections can contain collections and arrays, as well as arrays can contain other arrays and collections.

3.2.1 Collections

3.2.2 Arrays

Chapter 4

Templates, Rules, and Constraints

4.1 Design Templates

4.2 Rules and Constraints

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

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

4.2.1 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[3]

4.2.2 Relational Expression Constraints

Relational expressions are rules of the following form:

⟨*rel-exp*⟩ ::= ⟨*exp*⟩ ⟨*rel-operator*⟩ ⟨*exp*⟩

⟨*exp*⟩ ::= ⟨*exp-operand*⟩ (⟨*exp-operator*⟩ ⟨*exp-operand*⟩)*

⟨*exp-operand*⟩ ::= ⟨*property*⟩ | ⟨*constant*⟩

⟨*property*⟩ ::= `id` (⟨*index*⟩)* (`.` `id` (⟨*index*⟩)*)* (`.` ⟨*function*⟩)?

⟨*constant*⟩ ::= `number` | `string`

⟨*exp-operator*⟩ ::= + | `-` | `*` | /

⟨*index*⟩ ::= [`number`]

⟨*function*⟩ ::= **size** | **length** | ...

⟨*rel-operator*⟩ ::= < | <= | == | != | > | >=

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

Chapter 5

Data Exchange Capabilities

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

- INCLUDE of Eugene scripts
- IMPORT of Eugene containers
- IMPORT of biological data from the iGEM partsregistry[1]
- IMPORT and EXPORT of biological data from Genbank files[2]
- IMPORT and EXPORT of biological data compliant with the Synthetic Biology Open Language (SBOL) standard[3]
- VISUALIZATION of designs compliant with the SBOL Visual graphical notation[4] using Pigeon[2]

5.1 Inclusion of Eugene Scripts

Eugene v2.0 supports to include Eugene scripts into a Eugene script via the `include` keyword. The syntax is defined as follows:

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

The string argument must specify the relative location of the included Eugene script to the actual Eugene script. If the specified Eugene script does not exist or is not a well-formed Eugene script, then an error is reported.
Examples:

- Select all promoters whose strength is greater than 5:

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

Note! In Eugene v2.0 we do not check for cyclic inclusions of Eugene script. For example, if the Eugene script A includes Eugene script B and Eugene script B includes Eugene script A, then an endless loop is the result.

5.2 Data Import of Eugene Containers

5.3 Data Import from the iGEM Partsregistry

5.4 GenBank Data Import and Export

5.5 SBOL Data Import and Export

5.6 SBOL Visual Compliant Design Visualization

Chapter 6

Imperative Language Features

Imperative languages features can serve for (1) the automated generation of a library of genetic elements (such as randomly generated DNA sequences), (2) the specification of the control-flow of design synthesis based on conditions and repetitions, and (3) to process the automatically enumerated rule-compliant genetic designs. Eugene v2.0 supports to following imperative language features:

- **Branches** control the execution of statements if certain conditions are satisfied (Section 6.2).
- **Loops** enable facilities to (1) iterate over enumerated designs of Eugene's declarative features and (2) to execute a sequence of statements in succession for multiple times while a certain condition is satisfied (Section 6.3).
- **Functions** group a sequence of statements into reusable modules whose execution can be invoked when required (Section 6.5).

6.1 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!

6.2 Conditional Branches

6.3 Loops

Loops enable to control how many times an operation or a sequence of operations is performed in succession. The number of times can be controlled through the specification of conditions.

6.3.1 `WHILE` loops

Eugene v2.0 supports `WHILE` loops that execute a sequence of operations while a condition is satisfied, i.e. `true`. The following example prints 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 while the condition `i<=10` is satisfied.

6.3.2 `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.

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

6.4 Dynamic Naming of Genetic Parts

6.5 Function Prototyping

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

- `println`, `PRINTLN`
  Equivalently to the `print` statement, 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 outputs a line feed, i.e. the NEWLINE character.

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

```
// 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.

- `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 6.4) — and stores the element in the global library.

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

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

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

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

- `query`, `query`

Bibliography

[1] Standard, E. S. S. *http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf*

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

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