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

November 5, 2014

Technical Report No. ECE-2014-NN

Contents

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.

```
// single line comment

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

2.2 Primitive Values

Following the supported primitive types, Eugene v2.0 supports the specification of strings (i.e. character sequences enclosed in double quotas `xxx`) and floating point numbers. In Eugene v2.0, identifiers are unique. That is, no two variables can have the same identifier. Identifiers are also case-sensitive.

```
string ::= " [^"] "
number ::= ( [0-9] )* (. ( [0-9] )* )?
```

2.3 Primitive Types

Eugene v2.0 supports three types of primitives that can be either a floating-point number (**num**), a string of characters (**txt**), or a boolean (**bool**).

```
primitive-type ::= num | txt | bool
```

2.4 Arrays of Primitives

Eugene v2.0 provides two types of arrays, namely arrays of of floating point numbers (**num**[]) and arrays of strings (**txt**[]).

```
primitive-array-type ::= num[] | txt[]
```

2.5 Identifiers

Identifiers are unique in the Eugene v2.0 language. Also, identifiers must be first 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_] )*
```

2.6 Variables

Variables have a a type (see Sections 2.3 and 2.4) and a unique identifier (see Section 2.5). A variable can only take values of its specified type (see Section 2.2) that can either be constants or expressions (see Section 2.7).

⟨*variable-declaration*⟩ ::= ⟨*primitive-type*⟩ ⟨*id*⟩ ;

⟨*primitive-type*⟩ ::= **txt** | **txt**[] | **num** | **num**[] | **bool**

⟨*id*⟩ ::= [a-zA-Z] ([a-zA-Z0-9_])*

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 an array of strings variable with the identifier `txt_array`:
  `txt[] txt_array;`

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 3.3.2 and Chapter 5 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.

2.8 Assignment Statements

Assignment statements assign values to variables and are interpreted from right to left, i.e., the result of the expression specified on the right-hand-side (RHS) is assigned to the variable specified on the left-hand-side (LHS). The LHS and the RHS are separated by the equals operator (`=`).

Values can be assigned to variables at the time of the variable's declaration or at later points during interpretation (see Section 2.8). Values assigned to variables at the time of the variable declaration, can be overwritten and/or re-calculated at a later time. The values of variables will be overwritten if a new assignment is made.

⟨*assignment*⟩ ::= ⟨*lhs*⟩ = ⟨*expression*⟩ ;

⟨*lhs*⟩ ::= ⟨*variable-declaration*⟩ | ⟨*id*⟩

⟨*expression*⟩ ::= ⟨*multiplication*⟩ | ⟨*division*⟩

⟨*multiplication*⟩ ::= ⟨*addition*⟩ | ⟨*subtraction*⟩ (* ⟨*expression*⟩)?

⟨*division*⟩ ::= ⟨*addition*⟩ | ⟨*subtraction*⟩ (/ ⟨*expression*⟩)?

⟨*addition*⟩ ::= ⟨*atom*⟩ (+ ⟨*expression*⟩)?

⟨*subtraction*⟩ ::= ⟨*atom*⟩ (- ⟨*expression*⟩)?

⟨*atom*⟩ ::= ⟨*id*⟩ | ⟨*number*⟩ | ⟨*string*⟩ | (⟨*expression*⟩)

2.9 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>
```

2.10 Reserved Words

Eugene v2.0 has the following reserved words which 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

Chapter 3

Declarative Language Features

Eugene v2.0 supports the specification of synthetic biology designs based on biological *Facts* and *Rules*. Additionally, Eugene v2.0 supports the invocation of user-defined *Functions* for the verification and enumeration of rule-compliant synthetic biology designs.

⟨*script*⟩ ::= ⟨*declaration*⟩+

⟨*declaration*⟩ ::= ⟨*fact*⟩ | ⟨*rule*⟩ | ⟨*function*⟩

3.1 Declaration of Biological Facts

Facts can either be specified manually or imported using data exchange standards. Here, we define the Eugene language for the manual specification of facts. We describe Eugene's data exchange facilities in Section 4. Every specification of a fact must be closed with the semi-colon character (;).

⟨*fact*⟩ ::= (⟨*property*⟩ | ⟨*type*⟩ | ⟨*instantiation*⟩ | ⟨*relation*⟩) ;

In the following, we define the syntax for the specification of properties, types, instantiations, and relations in more detail.

3.1.1 Properties of Biological Facts

Properties represent attributes and characteristics of genetic components, either primitive components or composite components. Each property must have an identifier (see Section 2.5) and a primitive type (see Section 2.3).

⟨*property*⟩ ::= **Property** `id` (⟨*primitive-type*⟩)

Example:

3.1.2 Types

Types are abstract representations of the common structure of genetic elements, such as parts. Eugene v2.0 only supports the definition of types of parts. Types define the properties and structure of genetic elements. For example, a type `Promoter` can defines the common characteristics (i.e. properties) of genetic elements, such as DNA sequence, list of operators, or strength.

⟨*type*⟩ ::= ⟨*part-type*⟩

⟨*part-type*⟩ ::= **PartType** `id` `(` ⟨*list-of-ids*⟩ `)`

⟨*list-of-ids*⟩ ::= `id` (`,` ⟨*list-of-id*⟩)?

For the specification of part types, the identifiers in the `<list-of-ids>` must refer to the identifiers of specified properties (see Section 3.1.1). As defined in Section 2.5, properties must have been defined before specifying a type.

3.1.3 Instantiations

In Eugene, types define the common properties and structure of genetic elements. Hence, Eugene also provides facilities to instantiate types with concrete genetic facts. That is, instances of genetic types have a unique name and property values. For examples, a concrete instance of the type `Promoter` can have a specific DNA sequence (e.g. `ATCG`).

⟨*instance*⟩ ::= `id id` `(` ⟨*list-of-property-values*⟩ `)`

⟨*list-of-property-values*⟩ ::= (⟨*dot-notation*⟩ | ⟨*list-notation*⟩)

⟨*dot-notation*⟩ ::= `.` `id` `(` `primitive-value` `)` (`,` ⟨*dot-notation*⟩)

⟨*list-notation*⟩ ::= ⟨*value*⟩ (`,` ⟨*list-notation*⟩)

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. In Eugene, we support two notations of specifying the property values of instances, the dot (**.**) notation and the list notation. Using the list notation necessitates to specify the property values in the same order as specified in the type specification. The dot notation offers the possibility to specify the property name of the property to that the specified value must be assigned.

Eugene does check the types 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`.

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.

3.1.4 Composite Biological Facts

3.1.5 Relations

3.2 Design Templates

3.3 Rules and Constraints

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

3.3.1 Structural Constraints

integrate table from miniEugene paper

3.3.2 Relational Expressions

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

3.4 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.

3.4.1 Collections

3.4.2 Arrays

Chapter 4

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]

4.1 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.

4.2 Data Import of Eugene Containers

4.3 Data Import from the iGEM Partsregistry

4.4 GenBank Data Import and Export

4.5 SBOL Data Import and Export

4.6 SBOL Visual Compliant Design Visualization

Chapter 5

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 5.2).
- **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 5.3).
- **Functions** group a sequence of statements into reusable modules whose execution can be invoked when required (Section 5.5).

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 Conditional Branches

5.3 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.

5.3.1 `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.

5.3.2 `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 example from above looks as follows when using a `FOR` loop.

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

5.4 Dynamic Naming of Genetic Parts

5.5 Function Prototypes

5.6 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/1`, `PRINT/1`
  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/1`, `PRINTLN/1`
  Equivalently to the `print/1` 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/1`, `SIZEOF/1`
  The `sizeof` function takes as input an identifier (`id`) and returns the size of the corresponding element. For example, `sizeof(myarray)` returns the length of a the named element `myarray`.

- `random/2`, `RANDOM/2`
  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/1`, `SAVE/1`
  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 5.4) — and stores the element in the global library.

- `product/1`, `PRODUCT/1`
  describe product() here!

Appendices

Appendix A

Railroad Diagrams

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.