Funding Scheme: THEME [ICT-2007.8.0] [FET Open]

Paving the Way for Future Emerging DNA-based Technologies: Computer-Aided Design and Manufacturing of DNA libraries

Grant Agreement number: 265505

Project acronym: CADMAD

Deliverable number: D1.2

Deliverable name: DNApl and vDNApl (GUI) dialects requirements, specification and extended parsing algorithms

Contractual Date[1] of Delivery to the CEC: M12
Actual Date of Delivery to the CEC: M12
Author(s)[2]: Prof. Natalio Krasnogor
Participant(s)[3]: UNOTT (lead), WEIZMANN, ETH, UEVE, UKB
Work Package: WP1
Security[4]: Pub
Nature[5]: R
Version[6]: 0.0
Total number of pages: 10

1 As specified in Annex I
2 i.e. name of the person(s) responsible for the preparation of the document
3 Short name of partner(s) responsible for the deliverable
4 The Technical Annex of the project provides a list of deliverables to be submitted, with the following classification level:
Pub - Public document; No restrictions on access; may be given freely to any interested party or published openly on the web, provided the author and source are mentioned and the content is not altered.
Rest - Restricted circulation list (including Commission Project Officer). This circulation list will be designated in agreement with the source project. May not be given to persons or bodies not listed.
Int - Internal circulation within project (and Commission Project Officer). The deliverable cannot be disclosed to any third party outside the project.
5 **R (Report):** the deliverables consists in a document reporting the results of interest.
P (Prototype): the deliverable is actually consisting in a physical prototype**,** whose location and functionalities are described in the submitted document (however, the actual deliverable must be available for inspection and/or audit in the indicated place)
D (Demonstrator): the deliverable is a software program, a device or a physical set-up aimed to demonstrate a concept and described in the submitted document (however, the actual deliverable must be available for inspection and/or audit in the indicated place)
O (**Other):** the deliverable described in the submitted document can not be classified as one of the above (e.g. specification, tools, tests, etc.)
6 Two digits separated by a dot:
The first digit is 0 for draft, 1 for project approved document, 2 or more for further revisions (e.g. in case of non acceptance by the Commission) requiring explicit approval by the project itself;
The second digit is a number indicating minor changes to the document not requiring an explicit approval by the project.

Abstract

The goal of this WP is to specify the requirements for a DNA programming language (DNApl) and associated software tools that will enable the formulation of a library of output DNA molecules as a function of combinatorial assemblages of the input and intermediate sequences DNA sequences. Moreover, it aims at creating the computational tools necessary, e.g. a parser, interpreter, debugger, Graphical User Interface (GUI), etc. for designing combinatorial DNA libraries programmatically.

Keywords[7]:

DNA Library Design language, DNALD, vDNALD, Requirements, Specification

1. Introduction

a. Aim / Objectives

This work package is concerned with the computer-aided specification of large combinatorial DNA libraries: namely the programmatic specification, interpretation, visual representation, and management of sets of DNA sequences that are defined recursively as functions of input, intermediate and output sequences. **This specific deliverable (D.1.2) is concerned with producing the *application* dependent specification for the functionality of a DNA Library Designer language (DNALD for short) and Graphical User Interface (GUI) tools.**

b. State of the Art

Currently, there is no - outside CADMAD - tools for combinatorial DNA library programming. Thus the state of the art is pretty much the work we ourselves have conducted during the first year of operation. In the previous deliverable (D1.1.) we collected a series of 11 core requirements for DNALD, 7 of which are related to the *language and editor* and 4 of them are for *visual* tools. These requirements form the starting point for the innovations & implementations described below. In what follows we use "DNALD" instead of DNApl for the name of the language and tools investigated.

c. Innovation

The proposed technological platform will be built on solid formal language theory [1],[2],[3],[4],[5],[6] and will create a DNA programming language (to be called DNALD) that will permit the seamless specification of large combinatorial molecular libraries. DNALD will have a major impact on the way scientists specify DNA libraries and we plan to expand its influence through the creation of a visual programming counterpart, vDNALD, providing an illustrative means of reasoning about and communicating DNA library combinatorics. Furthermore, we will enable DNALD and vDNALD to be a tailored to particular biological problems and their differing constraints through the collaborative development of multiple dialects of DNALD that more closely reflect the nature of the sequences required for experimentation, for example in protein crystallization, post-transcriptional regulation through RNA secondary structure or DNA-based nanotechnology. In order to bootstrap library specification in these fields we will develop novel algorithms that can reverse engineer DNALD programs from existing sequences. The specification of DNA libraries in DNALD and vDNALD will be supported by a state-of-the-art graphical user interface that is friendly, fast, robust and portable, i.e. accessible through multiple web browsers, operating systems and mobile devices , for a more ergonomic, scalable and reusable DNA library design experience than exists today.

2. Implementation

DNALD is a new programming language for the specification of combinatorial DNA libraries that has a short but important history. DNALD is a refinement and extension of DNAPL prototype language developed by Yair Mazor, Uri Shabi and Ehud Shapiro at the Weizmann Institute. DNALD inherits the structure and intentions of DNApl whilst rationalising the syntax and semantics of the language to make it more consistent, efficient to parse and interpret, and amenable to future extension. To evince the extent of our contribution, and as a specification for the core of DNALD, we present a brief overview of the DNApl syntax and library structure.

[7] Keywords that would serve as search label for information retrieval

We then analyze its features and explain how these have informed, based on the specification, the prototyping of DNALD.

DNApl

DNApl syntax

The primary innovation of DNApl was simply to replace long, cumbersome DNA sequences with symbolic names by which they can be referred to. Symbolic names are associated with DNA sequences using the definition operator (`:=`). For example the sequence CTCGAG is assigned to the name XhoI: `XhoI := CTCGAG`;

A semi-colon is required to end the line and the definition. (In what follows only the expression on the left hand side of the definition is shown and without a terminating semi-colon.)

Subsequences can then be obtained using symbolic names, the subsequence operator (`[]`) and an integer index: `XhoI[2]` returns `T` the second letter of the sequence referred to by XhoI.

Longer subsequences can be obtained using a range of indices denoted by a colon (:) separated pair of integers: `XhoI[2:6]` returns `TCGAG`, the sequence from the second to the sixth letter.

The last index symbol end can also be used as a stop index to obtain everything up to the 3' end: `XhoI[2:end]`, which also returns `TCGAG`.

The concatenation operator (`.`) joins two (sub)sequences: `XhoI[1].XhoI[5:6]` returns `CAG`.

Mutations can be made by using the mutation operator (=) in conjunction with the subsequence operator: `XhoI[1=A]` returns `ATCGAG`. Multiple mutations are separated by a comma (`,`) so that: `XhoI[1=A, 3:4=GT]` returns `ATGTAG`.

Sequences can be repeated using the repetition operator (`*`): `XhoI*2` returns `CTCGAGCTCGAG`.

Sometimes it is more appropriate to work with sequences of amino acids rather than nucleotides. To do this DNApl provides the amino acid function (a_a), e.g.:

```
     gene := a_a(VVPSTQPVTTPPATTPVTTPTIPPS);
```

DNApl also has three combinatorial operators: or (|), all in separate wells (+), and all in the same well (++). How sequences defined as amino acids sequences or by combinatorial operators are treated when combined with other sequences is unclear from the explanation of DNApl given at http://www.cadmad.eu/general-explanation. Some of these operations appear to have been intentions without implementations.

Library structure

A DNApl library is a single file containing a set of definitions. These definitions are divided between three sections to form three disjoint subsets: *inputs, outputs and intermediates*. The input and output sections are each delineated by an eponymous keyword and end. (Keywords and symbolic names are case-insensitive.) Intermediates are any definitions between the end of the input section and beginning of the output section. Each section has a slightly different meaning which in turn places constraints on the content of the definition expressions permitted:

- Inputs are existing fragments of DNA that the library designer can provide to the manufacturer, such as sequenced plasmids. Therefore input definitions cannot contain mutation operations, etc. only a single sequence.
- Intermediates are definitions that are used in outputs (or other intermediates) but are not directly required as an output of the library. Intermediates are either variations on inputs, outputs, other intermediates or entirely new sequences which may need to be obtained by traditional *de novo* synthesis. Intermediates can be used to construct outputs in a parsimonious manner by factoring out reusable subsequences derived from combinations of DNAPL operations.
- Outputs are the target sequences which the library designer wants synthesised. Expressions in output definitions may refer to inputs, intermediates or **other outputs**.

Observations on DNApl with regard to DNALD

From the syntax of DNApl we can make several observations about the nature of the language in comparision to established programming languages like Java or Python, and establish **the position taken in DNALD** on each:

1. Keywords and symbolic names are case-insensitive. This is reasonable as the intended audience are biologists who may not be used to the case-sensitive restrictions of most programming languages and therefore enforcing case-sensitivity, especially of keywords, might seem pedantic. Additionally, by allowing case-insensitive symbolic names, names must be unique words and will therefore be easier to discriminate and unambiguous in conversation. **DNALD will be a case-insensitive language**.
2. Sequence strings are not quoted. This is highly unusual and potentially problematic for parsing. For instance, is `tag` a sequence or a symbolic name? **DNALD will require all sequences to be quoted.**[8]
3. Definitions are terminated by a semi-colon. This is unnecessary as what must follow an expression is either another definition (`symbolic_name :=`) or the section terminator (END). **Semi-colons for terminating definitions are optional in DNALD** to be inclusive of users who are experienced with languages which require semi-colons and might miss them. In the future semi-colons may also be used with subsequence operations or function calls to separate sets of arguments.
4. Subsequence indices start from 1 and go up to the length of the sequence inclusive, whereas array indicies in programming languages are usually 0-based and exclusive. This is a sensible departure from Java as DNA sequences are also indexed starting from 1 and therefore the final index must be inclusive. **DNALD will have 1-based, inclusive indexing of sequences.** Implementors in languages with 0-based sequences must be careful not to confuse the two systems (although this is a simple error to validate and provide instant feedback on in the textual editor GUI).
5. The symbol `end` can be used as a stop index to obtain everything up to the 3' end. This is a useful convenience as the length of a sequence returned from several operations may be difficult to determine. Java has no such syntax while Python allows slices without a stop index to achieve the same end, e.g. sequence[3:]. **DNALD will use the `end` symbol** as it is more explicit than Python, conveying the intentions of the library designer to readers who may be unfamiliar with the language.

Specification of the new DNALD language

Library structure

Reflecting the intention to manufacture, a DNALD library is structured similar to DNApl:
1. optional codon tables that maps amino acids to sets of codons.
2. a set of input definitions of unambiguous DNA sequences. Completely *de novo* libraries may not have inputs, therefore this section should also be optional.
3. an untitled section of intermediate definitions that are used by other definitions but not a manufacturing target.
4. and the required set of output definitions that will be produced.

DNALD semantics

Names are assigned to the results of an expression using the definition operator (`:=`) to allow sequences to referred to in other expressions. We say that the name is defined as being the set of sequences resulting from the expression that followed, and refer to the (name, set of sequences) pair as a "definition".

Expressions

An expression in the DNALD language is a series of one or more operations involving a sets of DNA sequences. The result of an expression is the computed set of DNA sequences resulting from the execution of those operations. In summary, **all DNALD expressions should accept and operate on set of DNA sequences and return a set of DNA sequences**. It is interesting to note that sets need not be defined explicitly, for example the simplest unary expression is a DNA sequence enclosed in quotes (e.g. 'ATG') which actually returns a single member set.

[8] Quoting sequences has the added advantage of providing delimiters between which other non-sequences characters can be used and interpreted without overburdening the grammar of the parser. For instance, number sequences in EMBL or GenBank formats can be copy-pasted between quotes as the interpreter can extract the sequence while preserving the numbering for the user.

Associativity

Operations are either unary or binary, operating on one or two expressions respectively (or one expression and an integer in the case of the repetition operation). Unary operations are either function calls or intra-sequence operations. Intra-sequence operations are an expression followed by a matching pair of brackets (such as `[]` or `{}`) containing one or more lists of arguments and optional qualifiers. Intra-sequence operations operate in a left-associative manner so that the result of the leftmost operation is the input to the next leftmost and so on. Function calls are a function name followed by parentheses containing one or more lists of expressions. Binary operations are two expressions either side of an operator character or keyword. Binary operations can be chained together and operate in a right-associative manner so that the result of the rightmost operation becomes the right operand in the next rightmost and so on.

Directives

Operations producing sets with more than one member are separated-union (`+`), mixed-union (`/`) and one-of-union (`|`, or-groups) and back-translation (see below). The mixed- and separated-union operations return the set of their operands with the directive that they are to be produced in the same or different wells respectively. Directives are heritable in that the results of expressions which make reference those sets inherit the same semantics. For example, `s := 'ATG' + 'TAC'` defines `s` as the the set `{'ATG', 'TAC'}`. Concatenation (`.`) of two sets of sequences: `s.'GCC'` results in the set `{'ATGGCC', 'TACGCC'}`, where the elements of right operand are appended to the elements of the left operand (the joined Cartesian product of the two sets). In this way operations on sets with multiple items act as an implicit for-loop, and therefore operations on multiple multi-item sets act like nested for-loops.

Ambiguous sequences

Input sequences are restricted to the alphabet {A, C, G, T} but sequences used in the expressions of intermediate and output definitions may contain ambiguous nucleotides. Sequences containing ambiguous nucleotides form a one-of-union group, that is a set of sequences from which the library manufacturer may produce one according to manufacturing constraints. It may be the case that the construction plan can be significantly simplified (and reuse optimised) by favoring one alternative over than another, therefore ambiguous sequences and one-of-unions in general are a means by which the library designer can provide degrees of freedom to the manufacturer.

Back-translation

Amino acid sequences can be used in conjunction with a user-specified codon table (or the default based on the standard genetic code of E.coli K12) and "back-translated" to the set of DNA sequences made from all combinations of the codons for each amino acid. As the number of coding sequences increases exponentially with length (being the product of the number of possible codons at each position in the protein sequence) it is necessary to use ambiguous nucleotides to reduce the codon sets. For instance, arginine, leucine and serine are each encoded by six unambiguous codons, but only two ambiguous codons, therefore back-translating to an ambiguous nucleotide sequence significantly reduces the number of sequences that need to be handled when evaluating expressions. To do this we have develop an algorithm to computed the set of ambiguous sequences encoding any set of unambiguous sequences and applied this to computing ambiguous codon tables from any user-defined codon table. The resultant sequences behave the same as a regular set ambiguous sequences, except that the amino acid sequence is retained with the set object.

<u>DNALD syntax</u>

Names given to value of any expression are a definition.
`x := 'ATG'` makes `x 'ATG'`.

Definitions are referenced by name, so that `y := x` makes `y` the value of `x` and therefore `'ATG'` also. Changing `x` to `'TTG'` also changes `y` to `'TTG'`.

Concatenation is simply the joining of two fragments with (`.`) :
`x := 'ATG'.'CC'` defines `x as 'ATGCC'`.

Concatenation operations can be chained[9]:
`x.'TTA'.'GACTAG'` returns `'ATGCCTTAGACTAG'`.

Slicing (`[]`), used to enable reuse of subsequences within the library, requires an index or a range:
`x[2]` returns `'T'` and `x[1:3]` returns `'ATG'`

Slicing also works on non-references to enable copy-pasted sequences to be sliced without needing to previously define them:
`'ATGCCA'[2:5]` returns `'TGCCA'`.

Indicies are 1-based and inclusive, and must be greater than or equal to 1 and less than or equal the length of all the sequences in the set being sliced. Where sequences in the set are different lengths the `end` keyword can be used to indicate everything up to the length of each sequence in the set.

Mutations are made by using assignment (=) in conjunction with slicing:
`'ATGCCA'[1='T']` returns `'TTGCCA'`.

Multiple mutations are separated by a comma (`,`):
`'ATGCCA'[1='T', 3:4='AT']` returns `'TTATCA'`.
Mutations can be longer or shorter than the specified range, enabling insertion and deletion. Because deletion and insertion change the length of the sequence, within the same mutation operator the original indicies of the sequence will be honoured.

Sequences can be reversed or complemented with functions of the same name: `reverse('ATG')` returns `'GTA'` and `complement('ATG')` returns `'TAC'`. These functions can be used together to obtain the reverse complement: `reverse(complement('ATG'))` returns `'CAT'`.

The combinatorial operations separated-union (+), mixed-union (`/`) and one-of-union (`|`) create sets of more than one sequence with the appropriate directives to produce all in separate wells, all mixed together or one of the set respectively. For example, `'TTA'+'TTG'` returns `'TTA'+'TTG'`.

Concatenating a combination returns the cross-product of the operands:
`'ATG'.( 'TTA'+'TTG')` returns another group of two sequences `'ATGTTA'+ 'ATGTTG'`.
A concatenation involving two groups of sequences of size two and three, e.g.:
`z := ('A'+'CC'+'GGG'). 'ATG'.( 'TTA'+'TTG')`
returns a group of six sequences
`'AATGTTA'+'AATGTTG'+'CCATGTTA'+'CCATGTTG'+'GGGATGTTA'+'GGGATGTTG'`

All operations work on sets so that mutation of z affects all members:
`z[1='T'] returns`
`'TATGTTA'+'TATGTTG'+'TCATGTTA'+'TCATGTTG'+'TGGATGTTA'+'TGGATGTTG'`.

Sequences can be repeated using the repetition operator (`*`):
`'ATG'*2` returns `'ATGATG'`.

Variable length repetitions are also allowed, e.g.:
`'TTA'*(1:3)` returns the one-of-union set `'TTA'|'TTATTA'|'TTATTATTA'`.

Lastly, input definition expressions are constrained to single DNA sequences composed of unambiguous nucleotides only, as these are to be provided by the library designer:

```
    INPUT
            I1 : = 'ATGCTATCGGGGCTAGAAGGTAG'
    END
```

whereas intermediate and output expressions can utilise any of the above operations, in conjunction with ambiguous nucleotides.

[9] Most definitions are chains of concatenation operations, which is because any DNA sequence can be expressed in terms of concatenation of its subsequences.

Evaluation model

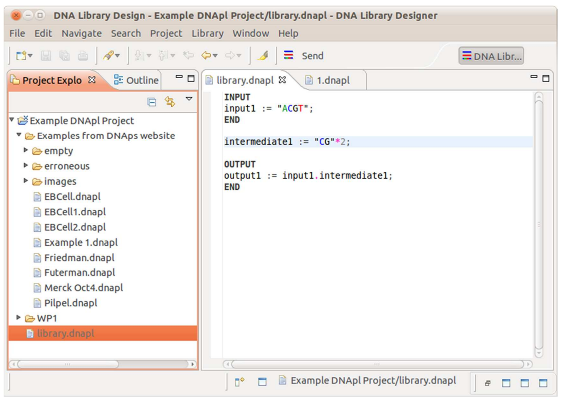
Names in DNALD can only be defined once in a library. This simple constraint leads to an evaluation model reminiscent of a spreadsheet, as changing the expression of a definition impacts all definitions that make reference to it, just as a changing the value or formula of a cell in a spreadsheet can trigger a wholesale update of the cells that reference it via its coordinates. A consequence of this is that definitions may not include circular references, i.e. a definition cannot have an expression that refers to it either directly or indirectly because this would create an infinite regression. Another consequence is that definitions may be evaluated in a non-linear order, to ensure that each definitions which are referred to have previously been evaluated. We have solved these issues in the DNALD interpreter by creating a directed graph from the set of definitions, where an out edge is a reference to another definition, removing cycles (and marking those definitions in cycles as erroneous) and topologically sorting the remaining nodes to obtain a feasible evaluation order.

3. Results

For this deliverable we have progressed beyond what was required and we have prototyped an implementation of the DNALD parser, interpreter and GUI (DNA Library Designer). We detail below the results obtained:

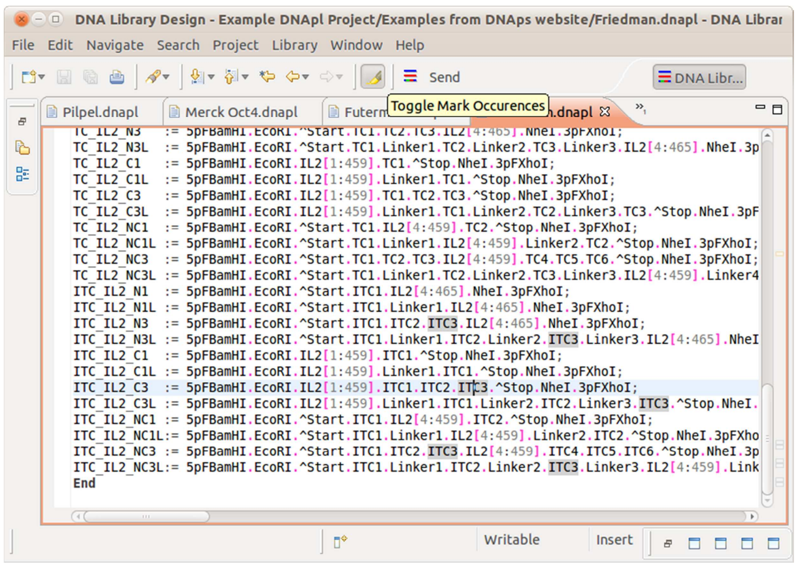
DNA Library Designer is the reference implementation of the DNALD language specified in D1.1 & D1.2, a parser, interpreter and a sophisticated integrated development environment (IDE) that enables biologists to design and explore combinatorial DNA sequence libraries with the support of a real programming editor. The software is developed as a set of Eclipse plugins using the Xtext domain-specific language (DSL) tool chain [7]. It can be installed alongside other plugins in an existing Eclipse installation, and is also available as a standalone product for Windows, Mac and Linux. DNA Library Designer offers syntax/reference highlighting and validation, code completion, source navigation, outline views and rename refactoring; on top of the Eclipse's already excellent find-replace (with regular expressions) and workspace model of project and file management with full text searching. To this GUI we have added semantic validation of DNALD expressions and relevant quick fixes, templates and wizards for DNA libraries including examples, and a rich hover showing the computed sequences of a definition, numbered and formatted appropriately. Versioning of libraries and projects is deferred to other Eclipse plugins such as MercurialEclipse or EGit. The standalone DNA Library Designer product bundles MercurialEclipse [8], giving library designers a fully integrated distributed version control system (DVCS). We envisage DVCS as the primary means of communicating DNALD libraries within working groups and back and forth to the manufacturer as designs are refined in line with what is possible to produce.

The figures 1, 2, 3, 4 and 5 highlight aspects of DNA Library Designer prototype we have developed. The features demonstrated in these figures are implemented by our customized classes which extend default/null implementations provided by the Xtext framework, the provision of which is managed by the Google Guice dependency injection framework. DNALD libraries are currently interpreted in a background thread whenever a change is made in the active editor. Erroneous definitions are skipped so that incomplete libraries can be partially interpreted and the entire library validated and augmented with semantic information. Interpreted sequence sets are then used to update the Definitions, Sequences, Sequence Fragment data and visualization views.



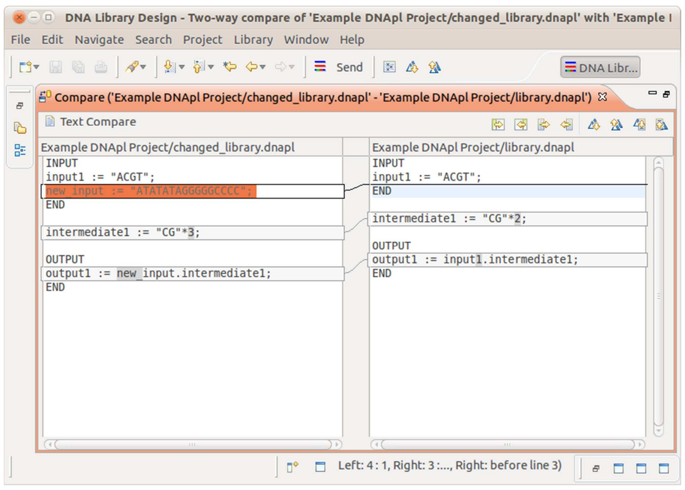
```
INPUT
input1 := "ACGT";
END

intermediate1 := "CG"*2;

OUTPUT
output1 := input1.intermediate1;
END
```

Figure 1: The main DNA Library Designer interface. The interface consists of the editor area (right), docked and open views such as the Project Explorer (left). An editor displays the contents of a file and provides advanced editing facilities. Views display information either derived from the contents of the active editor or pertaining to the file system or other plugins. Projects (top-level folders in The Project Explorer view) are located on disk in a workspace: an Eclipse managed directory of the users choosing (the workspace establishes a root directory from which paths within the application can be resolved and provides a location for persistent Eclipse metadata).


DNA Library Design - Example DNApl Project/Examples from DNAps website/Friedman.dnapl - DNA Librar
File Edit Navigate Search Project Library Window Help
Send
DNA Libr...
Pilpel.dnapl
Merck Oct4.dnapl
Futerm
Toggle Mark Occurences
n.dnapl
Writable
Insert

```
TC_IL2_N3    := 5pFBamHI.EcoRI.^Start.TC1.TC2.TC3.IL2[4:465].NheI.3pFXhoI;
TC_IL2_N3L   := 5pFBamHI.EcoRI.^Start.TC1.Linker1.TC2.Linker2.TC3.Linker3.IL2[4:465].NheI.3p
TC_IL2_C1    := 5pFBamHI.EcoRI.IL2[1:459].TC1.^Stop.NheI.3pFXhoI;
TC_IL2_C1L   := 5pFBamHI.EcoRI.IL2[1:459].Linker1.TC1.^Stop.NheI.3pFXhoI;
TC_IL2_C3    := 5pFBamHI.EcoRI.IL2[1:459].TC1.TC2.TC3.^Stop.NheI.3pFXhoI;
TC_IL2_C3L   := 5pFBamHI.EcoRI.IL2[1:459].Linker1.TC1.Linker2.TC2.Linker3.TC3.^Stop.NheI.3pF
TC_IL2_NC1   := 5pFBamHI.EcoRI.^Start.TC1.IL2[4:459].TC2.^Stop.NheI.3pFXhoI;
TC_IL2_NC1L  := 5pFBamHI.EcoRI.^Start.TC1.Linker1.IL2[4:459].Linker2.TC2.^Stop.NheI.3pFXhoI;
TC_IL2_NC3   := 5pFBamHI.EcoRI.^Start.TC1.TC2.TC3.IL2[4:459].TC4.TC5.TC6.^Stop.NheI.3pFXhoI;
TC_IL2_NC3L  := 5pFBamHI.EcoRI.^Start.TC1.Linker1.TC2.Linker2.TC3.Linker3.IL2[4:459].Linker4
ITC_IL2_N1   := 5pFBamHI.EcoRI.^Start.ITC1.IL2[4:465].NheI.3pFXhoI;
ITC_IL2_N1L  := 5pFBamHI.EcoRI.^Start.ITC1.Linker1.IL2[4:465].NheI.3pFXhoI;
ITC_IL2_N3   := 5pFBamHI.EcoRI.^Start.ITC1.ITC2.ITC3.IL2[4:465].NheI.3pFXhoI;
ITC_IL2_N3L  := 5pFBamHI.EcoRI.^Start.ITC1.Linker1.ITC2.Linker2.ITC3.Linker3.IL2[4:465].NheI
ITC_IL2_C1   := 5pFBamHI.EcoRI.IL2[1:459].ITC1.^Stop.NheI.3pFXhoI;
ITC_IL2_C1L  := 5pFBamHI.EcoRI.IL2[1:459].Linker1.ITC1.^Stop.NheI.3pFXhoI;
ITC_IL2_C3   := 5pFBamHI.EcoRI.IL2[1:459].ITC1.ITC2.ITC3.^Stop.NheI.3pFXhoI;
ITC_IL2_C3L  := 5pFBamHI.EcoRI.IL2[1:459].Linker1.ITC1.Linker2.ITC2.Linker3.ITC3.^Stop.NheI.
ITC_IL2_NC1  := 5pFBamHI.EcoRI.^Start.ITC1.IL2[4:459].ITC2.^Stop.NheI.3pFXhoI;
ITC_IL2_NC1L:= 5pFBamHI.EcoRI.^Start.ITC1.Linker1.IL2[4:459].Linker2.ITC2.^Stop.NheI.3pFXho
ITC_IL2_NC3  := 5pFBamHI.EcoRI.^Start.ITC1.ITC2.ITC3.IL2[4:459].ITC4.ITC5.ITC6.^Stop.NheI.3p
ITC_IL2_NC3L:= 5pFBamHI.EcoRI.^Start.ITC1.Linker1.ITC2.Linker2.ITC3.Linker3.IL2[4:459].Link
End
```

Figure 2: When the Mark Occurrences button is toggled on, positioning the text cursor on a definition reference highlights other references in the editor with a grey background and the definition with a beige background, useful for identifying reuse in densely specified libraries such as the one shown.


DNA Library Design - Two-way compare of 'Example DNApl Project/changed_library.dnapl' with 'Example I
File Edit Navigate Search Project Library Window Help
Send
DNA Libr...
Compare ('Example DNApl Project/changed_library.dnapl' - 'Example DNApl Project/library.dnapl')
Text Compare
Left: 4 : 1, Right: 3 :..., Right: before line 3)

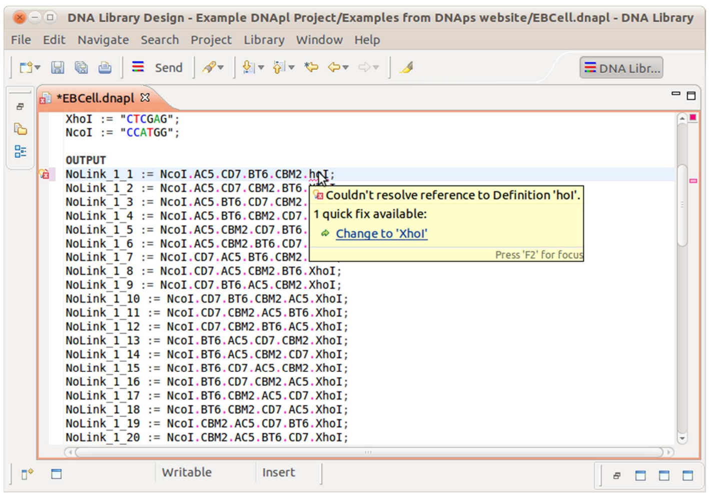
Example DNApl Project/changed_library.dnapl

```
INPUT
input1 := "ACGT";
new_input := "ATATATAGGGGGCCCC";
END

intermediate1 := "CG"*3;

OUTPUT
output1 := new_input.intermediate1;
END
```

Example DNApl Project/library.dnapl

```
INPUT
input1 := "ACGT";
END

intermediate1 := "CG"*2;

OUTPUT
output1 := input1.intermediate1;
END
```

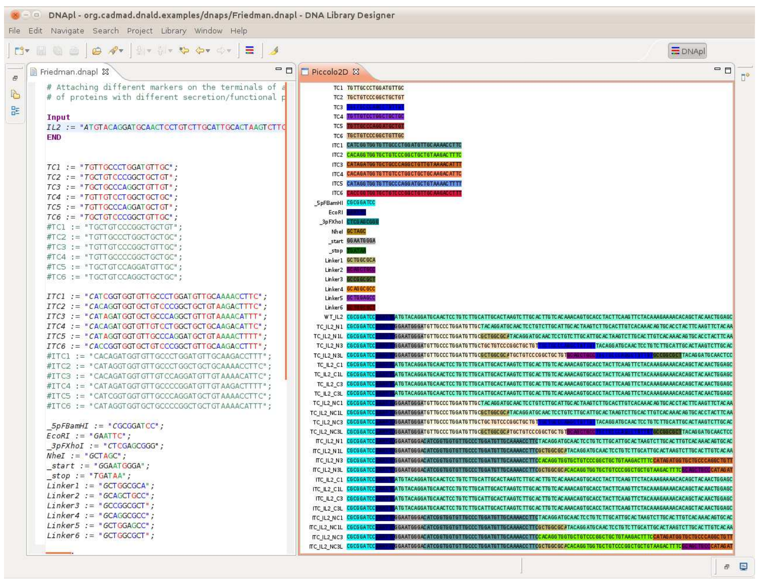
Figure 3: The Compare editor highlights the differences between two libraries (currently based on character differences rather than semantics) and provides actions for navigating between and copying differences; useful for manual partial merging of library versions.


DNA Library Design - Example DNApl Project/Examples from DNAps website/EBCell.dnapl - DNA Library
File Edit Navigate Search Project Library Window Help
Send
DNA Libr...
*EBCell.dnapl
Writable
Insert

```
XhoI := "CTCGAG";
NcoI := "CCATGG";

OUTPUT
NoLink_1_1 := NcoI.AC5.CD7.BT6.CBM2.hoI;
NoLink_1_2 := NcoI.AC5.CD7.CBM2.BT6.
NoLink_1_3 := NcoI.AC5.BT6.CD7.CBM2.
NoLink_1_4 := NcoI.AC5.BT6.CBM2.CD7.
NoLink_1_5 := NcoI.AC5.CBM2.CD7.BT6.
NoLink_1_6 := NcoI.AC5.CBM2.BT6.CD7.
NoLink_1_7 := NcoI.CD7.AC5.BT6.CBM2.
NoLink_1_8 := NcoI.CD7.AC5.CBM2.BT6.XhoI;
NoLink_1_9 := NcoI.CD7.BT6.AC5.CBM2.XhoI;
NoLink_1_10 := NcoI.CD7.BT6.CBM2.AC5.XhoI;
NoLink_1_11 := NcoI.CD7.CBM2.AC5.BT6.XhoI;
NoLink_1_12 := NcoI.CD7.CBM2.BT6.AC5.XhoI;
NoLink_1_13 := NcoI.BT6.AC5.CD7.CBM2.XhoI;
NoLink_1_14 := NcoI.BT6.AC5.CBM2.CD7.XhoI;
NoLink_1_15 := NcoI.BT6.CD7.AC5.CBM2.XhoI;
NoLink_1_16 := NcoI.BT6.CD7.CBM2.AC5.XhoI;
NoLink_1_17 := NcoI.BT6.CBM2.AC5.CD7.XhoI;
NoLink_1_18 := NcoI.BT6.CBM2.CD7.AC5.XhoI;
NoLink_1_19 := NcoI.CBM2.AC5.CD7.BT6.XhoI;
NoLink_1_20 := NcoI.CBM2.AC5.BT6.CD7.XhoI;
```

```
Couldn't resolve reference to Definition 'hoI'.
1 quick fix available:
Change to 'XhoI'
Press 'F2' for focus
```

Figure 4: Syntactic validation, colouring and quick fix auto-sugestions. Simple syntax errors such as missing quotation marks for sequences are highlighted in the editor. Errors are highlighted on the left hand side of the editor by a red symbol and by a red squiggle underline starting from the position of the first unexpected character. Hovering over either error marker brings up a tooltip explaining the source of the error. Certain errors can be autocorrected with a small user intervention called a quick fix (similar to autocompletion). Syntax colouring is fully customisable. By default definition names and references are black, operators are bright pink and nucleotide bases in sequences are coloured according the established convention (adenine is coloured green, cytosine blue, guanine black and thymine/uracil red).


DNApl - org.cadmad.dnald.examples/dnaps/Friedman.dnapl - DNA Library Designer
File Edit Navigate Search Project Library Window Help
DNApl
Friedman.dnapl
Piccolo2D

```
# Attaching different markers on the terminals of a
# of proteins with different secretion/functional p

Input
IL2 := "ATGTACAGGATGCAACTCCTGTCTTGCATTGCACTAAGTCTTC
END


TC1 := "TGTTGCCCTGGATGTTGC";
TC2 := "TGCTGTCCCGGCTGCTGT";
TC3 := "TGCTGCCCAGGCTGTTGT";
TC4 := "TGTTGTCCTGGCTGCTGC";
TC5 := "TGTTGCCCAGGATGCTGT";
TC6 := "TGCTGTCCCGGCTGTTGC";
#TC1 := "TGCTGTCCCGGCTGCTGT";
#TC2 := "TGTTGCCCTGGCTGCTGC";
#TC3 := "TGTTGTCCCGGCTGTTGC";
#TC4 := "TGTTGCCCCGGCTGCTGC";
#TC5 := "TGCTGTCCAGGATGTTGC";
#TC6 := "TGCTGTCCAGGCTGCTGC";

ITC1 := "CATCGGTGGTGTTGCCCTGGATGTTGCAAAACCTTC";
ITC2 := "CACAGGTGGTGCTGTCCCGGCTGCTGTAAGACTTTC";
ITC3 := "CATAGATGGTGCTGCCCAGGCTGTTGTAAAACATTT";
ITC4 := "CACAGATGGTGTTGTCCTGGCTGCTGCAAGACATTC";
ITC5 := "CATAGGTGGTGTTGCCCAGGATGCTGTAAAACTTTT";
ITC6 := "CACCGGTGGTGCTGTCCCGGCTGTTGCAAGACCTTT";
#ITC1 := "CACAGATGGTGTTGCCCTGGATGTTGCAAGACCTTT";
#ITC2 := "CATAGGTGGTGTTGCCCTGGCTGCTGCAAAACCTTC";
#ITC3 := "CACAGATGGTGTTGTCCAGGATGTTGTAAAACATTC";
#ITC4 := "CATAGATGGTGTTGCCCCGGATGTTGTAAGACTTTT";
#ITC5 := "CATCGGTGGTGTTGCCCAGGATGCTGTAAAACCTTC";
#ITC6 := "CATAGGTGGTGCTGCCCCGGCTGCTGTAAAACATTT";

_5pFBamHI := "CGCGGATCC";
EcoRI := "GAATTC";
_3pFXhoI := "CTCGAGCGGG";
NheI := "GCTAGC";
_start := "GGAATGGGA";
_stop := "TGATAA";
Linker1 := "GCTGGCGCA";
Linker2 := "GCAGCTGCC";
Linker3 := "GCCGGCGCT";
Linker4 := "GCAGGCGCC";
Linker5 := "GCTGGAGCC";
Linker6 := "GCTGGCGCT";
```

Figure 5: Side-by-side textual specification and visualization of interpreted library. The DNALD program (left) is interpreted in the background and the resultant library can be visualised (right) enabling library designs to check the results of their programs, and follow by colour the history of sequence fragments in the library.

4. Conclusions

We have completed the requirements collection and specification of the DNALD and vDNALD and, ahead of time (D1.3), we have started prototyping a visual programming environment for DNALD. Derived from the analysis in D1.1, D1.2 & D5.3 we have taken the technical decision to devolve application-dependent features to "plug-ins" that can be added to the visual environment and then activated/deactivated on a per-need basis. In this way, rather than defining and maintaining multiple, perhaps diverging, versions of the core language syntax, we can standardize the definition of DNALD and create a community of plug-ins developers that could contribute with new functionalities built on top of our core development. We anticipate that, for D1.3 and D1.4 we will be exploring plug-ins for three different families of applications: synthetic biology (parts, systems, devices, etc.), systems biology and structural biology (inhibitors, promoters, degradation tags, genes/proteins, etc.) and nanotechnology (strands displacements, origamies, etc.). These three families of applications, each anticipated to have related plug-ins, cover the spectrum of our user base.

5. References

[1] Yehezkel, T.B., Linshiz, G., Buaron, H., Kaplan, S. Shabi, U., Shapiro, E. De novo DNA synthesis using single molecule PCR. *Nucleic Acids Research*, 36:17 (2008).
[2] Linshiz, G., Yehezkel, T.B., Kaplan, S., Gronau, I., Ravid, S., Adar, R., Shapiro, E., Recursive construction of perfect DNA molecules from imperfect oligonucleotides. *Molecular Systems Biology*, 4:191, (2008).
[3] Cai, Y., et al.,A syntactic model to design and verify synthetic genetic constructs derived from standard biological parts. *Bioinformatics* 23:20 (2007).
[4] Searls,D.B. The language of genes. *Nature,* 420:211–217 (2002).
[5] Brendel,V. et al. Linguistics of nucleotide sequences: morphology and comparison of vocabularies. *J. Biomol. Struct. Dyn.,* 4 (1986).
[6] Loose,C. et al. A linguistic model for the rational design of antimicrobial peptides. *Nature* ,443 (2006).
[7] Xtext http://www.eclipse.org/Xtext/
[8] MercurialEclipse http://javaforge.com/project/HGE

6. Abbreviations

List all abbreviations used in the document arranged alphabetically.

DNALD	DNA Library Designer language
GUI	Graphical User Interface (GUI)
vDNALD	Visual DNA Library Designer language