PIL Specification

Shawn Ligocki, Chris Berlind, Joseph Schaeffer, Erik Winfree

November 30, 2010

PIL (Pepper Intermediate Language) is a language used by the Pepper compiler framework to describe a DNA design specification. It is derived from the .DES design specification language created by Joe Zadeh, Brian Wolfe, Niles Pierce, et al for the NUPACK multiobjective designer and has been extended to include kinetic and other information about DNA designs. A PIL design contains information about every sequence (domain), strand, structure (multi-stranded complex) and kinetic reaction in a design. It is meant to provide a common framework for sequence designers, including those used for pseudoknotted structures such as DNA tiles. Existing software may not yet implement or utilize all features of PIL, as discussed in section **??**.

1 Specification

A specification is a list of statements. Each statement ends in a line-break (newline character) but contains no line-breaks, so each non-blank line corresponds to exactly one statement.

1.1 General lexical conventions

- Integers are nonempty sequences of digits (regex: [0-9]+).
- Floating point numbers will have the standard syntax used by C, Python, etc. (e.g. .1, -24.551, 1e6, 4.2E12, 1.1e-05, etc.).
- Names are nonempty sequences of characters which are alphanumeric, underscore (_), **or hyphen (-)**. The first character must **not** be a digit (regex: [a-zA-Z_-][a-zA-Z0-9_-]*). Names are identifiers for different objects in the specification, thus each object must have a unique name. Names **are** case sensitive.

1.2 Comments and Whitespace

Extraneous whitespace will be ignored. That is, wherever a sequence of at least one whitespace character (i.e. space or tab, but not newline) appears in the specification, the sequence will be treated as a single space character.

Everything on one line following a **#** is a comment and will be ignored. The comment ends at the end of the line. There are no multiline comments.

Any line which contains only whitespace and/or a comment is considered blank and is ignored.

Examples of comments:

```
# This is a comment
### This is a comment too
sequence toe_x = NNNNNN : 6 # This comment could talk about the sequence
```

1.3 Sequences

A sequence is an indivisible segment of a nucleotide sequence. They are often called domains. Each sequence must be declared with a name, a length, and constraints on the allowable nucleotides.

Syntax:

```
sequence <Name> = <Constraints> : <Length>
```

`<Constraints>` is a sequence of characters describing what base may be assigned to each nucleotide. Each character is one of ACGTRYWSMKBDHVN. The most common are N for any base, S for a strong base (C or G), W for a weak base (A or T), and A, T, C or G for a specific base. For a complete list see Table **??**. Whitespace is allowed between symbols.

`<Length>` is the integer length of the sequence. While explicitly including the length is redundant (as it can be inferred from the constraints), it is required as an error-checking step and to improve ease of readability.

Example sequence statements:

```
sequence toe_x = NNNNNN : 6
sequence regulator = TCGGACT : 7
sequence __Anon-435 = SWABBBBBSS : 10
sequence Translate-And-Gate-data_x = SNNNNNNNNNNNNNNNNNNS : 20
```

1.4 Supersequences

A supersequence is a named ordered collection of sequences (or other supersequences) grouped together for convenience. Like sequences, each supersequence must be declared with a length matching the sum of the lengths of its constituent sequences.

Syntax:

```
supersequence <Name> = <List of (sequence | supersequence)> : <Length>
```

The `<List of (sequence | supersequence)>` is a space separated list. Each element in the list is the name of a sequence or supersequence optionally followed by an asterisk (`*`) meaning the Watson-Crick complement of the named sequence (`toe_x*` means complement of sequence `toe_x`).

Symbol	A	C	G	T	R	Y	W	S	M	K	B	D	H	V	N
Allowed bases	A	C	G	T	AG	CT	AT	CG	AC	GT	CGT	AGT	ACT	ACG	ACGT

Table 1: Nucleotide symbols

While it would be possible to allow such groups to be declared using sequence statements, this would lead to ambiguity whenever a sequence is given a name consisting entirely of characters allowable for nucleotide constraints. For example, the string `TAG` could be used as a name or as sequence constraints, and in certain contexts, these two usages would be indistinguishable. To avoid this problem, the supersequence statement was introduced.

Examples of supersequence statements:

```
supersequence x = toe_x data_x : 21
supersequence input = x y* x : 60
```

1.5 Equal

A sequence or supersequence can be forced to be exactly equal to another sequence, supersequence, or it complement. This is particularly useful for linking together PIL code fragments that define two DNA components that must interact by shared sequence domains.

Example equal statement:

```
equal anon435 y* toehold
```

1.6 Strands

A strand statement represents an individual nucleic acid strand and its sequence constraints. Strand statements follow a syntactic pattern similar to supersequence statements, but while a supersequence is a logical grouping of sequences, a strand represents a physical molecule of DNA or RNA.

Syntax:

```
strand <Name> = <List of (sequence | supersequence)> : <Length>
```

or

```
strand [<Substrate type>] <Name> = <List of (sequence | supersequence)> : <Length>
```

As defined for supersequences, `<List of (sequence | supersequence)>` is a space-separated list of sequence or supersequence names. `<Substrate type>` can be either `DNA` or `RNA` to indicate the nucleic acid type of the strand.

Examples of strand statements:

```
strand X = x : 21
strand Base = x* data_y data_z* : 51
```

1.7 Structures

A structure statement represents a single- or multi-stranded ordered complex and its exact secondary structure (pseudoknotted or unpseudoknotted) at the base-pair level.

Syntax:

```
structure <Name> = <List of strands> : <Secondary structure>
```

or

```
structure [<Parameters>] <Name> = <List of strands> : <Secondary structure>
```

The `<List of strands>` is a list of strand names separated by plus signs (`+`). A strand **may** appear twice in a single structure.

`<Secondary structure>` is specified using dot-bracket notation. To allow for pseudoknotted structures, future versions will support any bracket type (i.e. `( )`, `[ ]`, { }, `< >`) or labeled brackets (i.e. `(0 0)`, `(1 1)`, etc.) may represent paired bases, a dot (`.`) represents unpaired bases, and a plus sign represents a strand break. Whitespace is allowed between symbols and must appear after number-labeled left brackets and before number-labeled right brackets.

`<Parameters>` uses the following syntax:

```
<List of targets> @ <List of conditions>
```

where `<List of targets>` and `<List of conditions>` are comma-separated lists. `<List of conditions>` should contain a set of conditions under which the set of targets in `<List of targets>` should be satisfied. Possible target elements include `no-opt` (meaning no optimization) and `<Float>nt` (meaning within `<Float>` nucleotides of the target structure, e.g. on average), and `<Low> kcal/mol < dG < <High> kcal/mol` (meaning the free energy must be within the range). `<Low>` and `<High>` may be `-inf` or `inf` to indicate no bounds are given. Possible condition elements include `temperature=<Float>C` (temperature in degrees Celsius) and `Na=<Float>M` (sodium ion concentration) and `Mg=<Float>M` (magnesium ion concentration). Units `K` for Kelvin, and `mM`, `uM`, `nM` are also supported.

Examples of structure statements:

```
structure Gate = Out + Base : (((((((((((...............+))))))))))).......
structure [no-opt] In_Waste = In + Base : ((((((((((((((+...))))))))))))))
structure [1nt @ temperature=37.5C, Na=1.0M] IN = In : ...................
structure pknot = s1 + s2 : ((((((((....[[[[[[[[....))))))))+....]]]]]]]]....
```

1.8 Kinetics

Kinetic statements express the reactions you want to take place between structures. Kinetic reactions must preserve strands, thus each strand in an input structure must appear in an output structure. Also, a strand may not appear more than once in a kinetic reaction.

Syntax:

```
kinetic <List of inputs> -> <List of outputs>
```

or

```
kinetic [<Parameters>] <List of inputs> -> <List of outputs>
```

`<List of inputs>` and `<List of outputs>` are both lists of structure names separated by plus sign (`+`). `<Parameters>` is currently being used to specify target reaction rates. Its syntax is:

```
<Low> /M/s < k < <High> /M/s
<Low> /s < k < <High> /s
```

where the units clearly depend on whether this is a bimolecular or unimolecular reaction. `<Low>` and `<High>` are floating point numbers and `<High>` may be `inf` for no upper bound.

For example:

```
kinetic IN + Gate -> OUT + IN_Waste
kinetic [0 /M/s < k < 1e6 /M/s] Left + SeesawRight -> Right + SeesawLeft
kinetic [1000.1 /s < k < inf /s] Glob -> X + Y
```

1.9 Noninteracting

Noninteracting statements are used to specify specific negative design targets. These are sets of things that we do not want to interact with each other.

Generally, the syntax is:

```
noninteracting [<Type>] <List of things>
```

Where `<List of things>` could be sequences, strands or structures and their interpretation depends up on the `<Type>`. Currently, we have one possible type:

```
noninteracting [kinetic] <List of structures>
```

where `<List of structures>` is a space separated list of structures which should pairwise not react kinetically, and not waste too much time interacting.

Additional noninteracting lists will be used by Spurious_Design: `sticky-ends` for tile sticky ends, `toeholds` for strand displacement toeholds, `branch-migration` for 3-way and 4-way branch migration domains, and `single-stranded` for single-stranded sequences that must not hybridize to each other.

2 Related software tools

2.1 Software

Software that currently (November 2010) reads or writes PIL include: `spurious_design.py` (the wrapper for the `spuriousC` sequence-symmetry designer), `KinD.py` (strand displacement kinetics network evaluator and sequence designer), `random_design.py` (a trivial random sequence designer), `PIL-to-DES.py` (an yet-to-be-written converter for using the NUPACK designer).

2.2 Implementation status

Currently, no software implements the `[<Substrate type>]` qualifier. Currently no software implements the psuedoknotted dot-paren notation. Currently no software implements the structure target criteria specifications.