Stony Brook University

Qualifying Paper

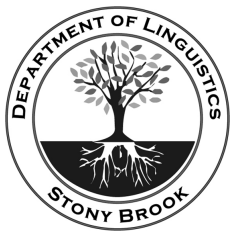
Department of Linguistics

`kist` package: documentation

Author:
Alëna Aksënova

Advisor:
Thomas Graf

April 24, 2018



Contents

Abstract

In this qualifying paper, I present a `kist` toolkit that allows one to work with formal language classes of the subregular hierarchy. From one side, we already know that these classes accommodate most natural language patterns. From the other side, it is shown that learning these classes can be done in polynomial time and data. However, there had been no implementation of the subregular tools yet, therefore development of this toolkit is a necessary and natural step.

1 Introduction

For a long time researchers working with the subregular language classes were forced to perform such basic operations as grammar extraction, sample generation, and well-formedness detection manually. The package `kist` (`k`ist `i`mplements a `s`ubregular `t`oolkit) that I am currently developing aims to simplify their life by providing these functions for different language classes in one place. It is implemented in Python (a language that is widely used in the scientific community), and the project is fully open source. That makes it available to anybody who faces the need in subregular tools.

The formal classes of *subregular* languages are widely discussed in formal language and linguistic literature nowadays. The idea of splitting the class of regular languages into the subregular hierarchy is not new, and was introduced by McNaughton and Papert (1971) several decades ago. However, not until recently this class was shown to be a good fit for natural language phenomena. Heinz (2011a,b) showed that several classes within the subregular hierarchy can accommodate natural language phonology. Follow-up by other researchers extended this view to other parts of language, such as morphology (Aksënova et al., 2016), syntax (Graf and Heinz, 2016), and even semantics (De Santo et al., 2017). Apart from linguistics, these classes found applications in the areas of neural learning (Avcu et al., 2017) and robotics (Rawal et al., 2011) as well.

These results encouraged a large amount of research on formal properties of subregular languages and their possible applications. However, manually designing grammars and languages is very difficult and risky, because even a minor mistake can lead to a completely different result. Moreover, it is impossible to generate large samples of languages and grammars by hand. Therefore, there should be an open-source toolkit that will supply researchers

with necessary instruments to simplify their work with subregular languages. `kist` toolkit provides these tools.

The next sections discuss the following points:

2. relevance of the package in different areas of research;
3. theoretical background behind this package;
4. installation guidelines and simple examples of use;
5. details on the architecture and implementation of the package;
6. conclusions and following steps in the development of the package.

2 Relevance

In this section, I show the importance of the subregular languages in different areas of research. The main ones are analysis of linguistic patterns and exploring limitations on their cognitive complexity, conducting controlled experiments with neural networks, and robotics.

Each of these areas requires generating lots of data or grammars of certain types, or analysis of extensive data samples. While it is achievable to manually go over a small data sample and create a subregular account for it, it becomes nearly impossible when larger samples are involved.

2.1 Linguistics

Phonology Subregular languages are widely applied to different parts of linguistics. Most of the phonological patterns can be captured strictly locally (Chandlee and Heinz, 2018). Heinz et al. (2011) show that numerous long-distance phenomena such as harmonies or long-distance dissimilations can be modeled using TSL and/or SP grammars (see (Heinz, 2015; De Santo and Graf, 2017; Aksënova and Deshmukh, 2018) for surveys and extensions). For the discussion of learnability of different types of harmonic patterns, see (Gainor et al., 2012). Additionally, Rogers (2018) shows that the waste majority of stress patterns are subregular as well.

Morphology There are also multiple application of subregular languages to morphotactics and morphology. For example, Aksënova et al. (2016) claim that morphotactics is at most TSL, and also show typological gaps that are found if the complexity of the pattern is beyond regular. Chandlee (2017) concludes that morphology can be captured via subregular functions, and

Aksënova and De Santo (in press) show that morphological derivations are in fact strictly local, even if overt positioning of morphemes seems to be more complicated.

Semantics Subregular approach found its applications even in semantics. According to De Santo et al. (2017), most generalized quantifiers seem to be subregular as well. Additionally, monomorphemic quantifiers are at most TSL (Graf, 2017).

Syntax The subregular approach is currently being extended to accommodate syntax as well, see (Graf and Heinz, 2016) for tree-based approaches.

2.2 Neural networks

The well-studied classes of subregular languages can also be used for conducting controlled experiments with neural networks. Avcu et al. (2017) did so by using the language classes at the very bottom of the subregular hierarchy, namely strictly local and strictly piecewise. The results of learning using samples from formal languages tell more than when natural language data is used: the nature, as well as the complexity of target patterns, can be controlled.

2.3 Robotics

Rawal et al. (2011) show that the subregular languages can be used to model robotic behavior. The formal class determines the properties and complexity of the system, and also predicts which computational tools it requires. For such tasks as well, manual extraction of patterns from the given data can be unnecessarily complicated.

3 Theoretical background

The subregular hierarchy (cf. Figure 1) is formed by limiting the power of regular languages in different ways. For example, the class of strictly local (SL) grammars evaluates substrings of a certain length, tier-based strictly local (TSL) grammars capture local dependencies among certain symbols while ignoring the rest, locally testable (LT) grammars can check whether an n-gram was used in a string or not, and strictly piecewise (SP) grammars operate with subsequences of a limited length.

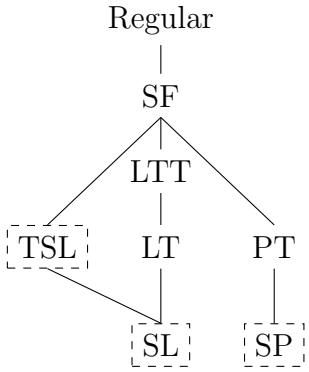


Figure 1: The subregular hierarchy

The `kist` toolkit currently implements strictly local, tier-based strictly local and strictly piecewise grammars that are highlighted on the Figure 1. In this section, I discuss these three subregular classes.

3.1 Strictly local

The class of strictly local (SL) languages is not different from n-gram models that are widely used in NLP. The core idea behind this class is to evaluate a string based on the n-grams that are contained in the given string. The positive SL grammars list allowed substrings of a language, whereas the negative ones contain the prohibited substrings.

For the alphabet $\Sigma = \{a, b\}$, consider the following language: *ab*, *abab*, *ababab*, etc. It is easy to notice, that the "rules" of this language require the well-formed words to start with a, end with b, and to alternate the symbols. Exactly the same generalization can be formulated using strictly 2-local grammar. The locality window $k = 2$ means that well-formedness of a symbol in each position can be decided based on information about the preceding symbol only. The following 2-SL grammar $G_{PSL} = (\rtimes a, ab, ba, b \ltimes)$[1] describes the language above. Strings such as *ab* or *abab* are allowed in this language, whereas **abb* or **bab* are not: the former one contains the illicit **bb* substring, and the latter one starts with a b, i.e. contains *$\rtimes b$. The

[1]The delimiters $\rtimes$ and $\ltimes$ are used to indicate the beginning and the end of a string, respectively. The empty string is then denoted as $\rtimes \ltimes$.

corresponding negative grammar lists all prohibited strings of the same language. $G_{NSL} = (*\rtimes b, *aa, *bb, *a\ltimes)$ is a negative SL grammar that defines the same language as above.

SL grammars capture local dependencies by blocking or allowing substrings of a certain length. As the result, it is not possible to capture a long-distance dependency with a SL grammar: if the distance among two dependent units is unbounded, the locality can never be achieved.

As for the learning result, it is trivial because it is equivalent to memorizing all possible sequences of a certain length, or finding out which ones were never observed in the data.

3.2 Tier-based strictly local

The subregular class of tier-based strictly local (TSL) languages is a proper extension of the SL one. TSL grammar follows the same logic: it evaluates strings by looking for allowed or prohibited substrings. However, instead of evaluating the whole string, TSL grammars operate over tiers, see (Heinz et al., 2011). Every symbol of a string is projected on a tier only if it is present in the *tier alphabet* T.

Consider the following language: *b*, *aaab*, *aaba*, *baa*. Its alphabet is the same as before, $\Sigma = \{a, b\}$. This language can be described as follows: a might be present or not in the word, but there must always be a single b. The idea of ignoring the symbol distribution which is not important for the language is the key idea behind the TSL grammars. For the language that was presented above, the distribution of a is not important, therefore only b is included in the tier alphabet, $T = \{b\}$. Then over the tier, the positive 2-local grammar $G_{PTSL} = (\rtimes b, b\ltimes)$ and the negative one $G_{NTSL} = (*\rtimes\ltimes, *bb)$ describe the desired generalization. The words *aabaaa* or *abaa* are well-formed, whereas **aaaa* or **bab* are not: over the tier, they contain the banned bigrams $*\rtimes\ltimes$ and **bb*, respectively. For a visual example, cf. Figure 2.

The core idea behind TSL languages is to capture long-distance dependencies in a local fashion; namely, by making them local over the tier.

The learning algorithm for TSL languages *kTSLIA* (tier-based strictly k-local inference algorithm) is designed by Jardine and McMullin (2017). It first learns the tier alphabet by detecting the set of symbols that exhibit some sort of dependency. When the tier is learned, the algorithm proceeds to learning SL grammar over the tier images of the input dataset. The algorithm works in the polynomial time from positive data only.

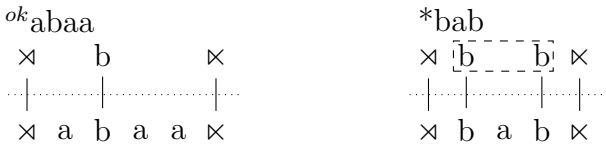


Figure 2: Examples of TSL evaluation

3.3 Strictly piecewise

The class of strictly piecewise (SP) languages differs from SL and TSL by the relation defined for the elements of the grammar. If before the defined relation was *successor*, i.e. immediate adjacency, SP class employs the *precedence* relation, see Fu et al. (2011).

As an example, consider the negative SP grammar $G_{NSP} = \{*bb\}$. Before, in the SL interpretation, this limitation meant *do not have two b adjacent to each other*. However, under the perspective of SP, it means that *b cannot follow another b*. With $\Sigma = \{a, b\}$, examples such as *aaaaabaa* and *aaa* are well-formed, whereas the ones as **aaabbaa* or **baaaaaab* are not. As in the previous example, the window size of this grammar is 2.

The main intuition behind this class is allowing or prohibiting subsequences of a certain length.

As for the learning this class of languages, ultimately it is just the extraction of possible or impossible k-piecewise subsequences. The easy way to represent an acceptor for the SP languages, is to construct a family of automata of the appropriate type, for the procedure and explanations refer to Heinz and Rogers (2013).

4 Quick start guide

In this section, I show how to install the `kist` package, and also provide a short example of its use.

4.1 Installing the package

Python version
Please make sure that you are running Python 3, version 3.6.3 or later.

Downloading the code
Click the following link and download the code, or clone the repository: `https://github.com/loisetoil/slp`.

Checking dependencies
Make sure that you have the following packages installed:

- `itertools` (for permutation-related tools)
- `random` (for random selection)
- `typing` (for variable-type annotations)

Running the toolkit[2]
In the terminal, move the location to the *slp* folder, and run *main.py* file in the interactive mode.

```
$ cd ~/slp
$ python3 -i main.py
```

4.2 Overview of the package

In this package, one can find learners, scanners, and sample generators for positive and negative SL, TSL and SP grammars. The following snippet of the code shows how to initialize grammars of each type.

```
>>> psl = PosSL()      # positive SL grammar
>>> nsl = NegSL()      # negative SL grammar
>>> ptsl = PosSL()     # positive TSL grammar
>>> ntsl = NegSL()     # negative TSL grammar
>>> psp = PosSL()      # positive SP grammar
>>> nsp = NegSL()      # negative SP grammar
```

(T)SL classes

The attributes defined for SL classes are the following:

- `alphabet`: a list of symbols that are used in the language.
  Default value is not given, but can be extracted from the grammar or the data.
- `grammar`: a list of banned/allowed n-grams.
  Can be extracted from the data.
- `k`: locality window of the grammar.
  The default value is 2.

[2]In future, the package will be available via `pip`.

- `data`: well-formed strings on the language.
  Required if the grammar needs to be extracted.
- `edges`: start and end markers.
  The default value is [‘>’, ‘<’].
- `data_sample`: sample that is generated by the grammar.
  Filled after *obj.generate_sample(n, rep)* is called.
- `fsm`: Finite State Machine (FSM) that corresponds to the grammar.
  It is constructed when the grammar is learned.

Basically, the only necessary arguments are `grammar` or `data`, because the latter one is required in order to extract the former one. By default, the locality window is set to 2, but this value can be changed.

The main available methods are the following:

- *obj.learn()*: extracts the grammar based on the data.
  Requires the data to be provided.
- *obj.scan(string)*: scans the given string.
  The string is required.
- *obj.generate_sample(n*=10*, rep=True)*: generates data sample.
  The argument n is the number of examples to be generated, this number is set to 10 by default. Another argument *rep* is the switch that allows or prohibits the generated data items to be repeated By default this value is set to *True*, i.e. it allows for repetitions.
- *obj.clean()*: cleans the grammar.
  Removes “useless” n-grams from the grammar.
- *obj.change_polarity()*: changes the polarity of the grammar.

The TSL classes have all the attributes and methods listed above. The only difference is the additional attribute `tier` that is defined for `PosTSL` and `NegTSL` classes. In `tier`, the tier items are stored. If the grammar is provided, the tier is required as well; and if the grammar is being extracted, the tier is also learned.

SP classes

The attributes defined for SP classes are the following:

- `alphabet`: a list of symbols that are used in the language.
  The default value is not given, but can be extracted from the grammar or the data.
- `grammar`: a list of banned/allowed n-piecewise subsequences.
  Can be extracted from the data.

- `k`: window size of the grammar.
  The default value is 2.
- `data`: well-formed strings on the language.
  Required if the grammar needs to be extracted.
- `data_sample`: sample that is generated by the grammar.
  Filled after *obj.generate_sample(n, rep)* is called.
- `fsm`: FSM family that corresponds to the grammar.
  Constructed when the grammar is learned.

Note that there is no attribute `edges`, because SP grammars operate with the *precedence* relation instead of the *successor*, therefore providing the edges will not introduce any new information.

The main available methods are the following:

- *obj.learn()*: extracts the grammar based on the data.
  Requires the data to be provided.
- *obj.scan(string)*: scans the given string.
  The string is a required argument.
- *obj.generate_sample(n=10, rep=False)*: generates data sample.
  The argument n is the number of examples to be generated, this number is set to 10 by default. Another argument *rep* is the switch that allows or prohibits the generated data items to be repeated. By default this value is set to *True*, i.e. it allows for repetitions.
- *obj.change_polarity()*: changes the polarity of the grammar.

4.3 Quick examples

To make sure that all steps indicated in the previous subsection went well, one can run the following examples. After providing the code snippet, I comment on what every line does. More details on the meaning of methods and attributes are given in Section 5.

Learning SL grammar:

```
>>> psl = PosSL()
>>> psl.k = 2
>>> psl.data = ['abab', 'ababab']
>>> psl.learn()
>>> psl.grammar
[('b', 'a'), ('>', 'a'), ('b', '<'), ('a', 'b')]
```

The class of positive SL grammar is being instantiated on the line *1*. Line *2* indicates that the locality window of the desired grammar is 2. The data is provided further on the line *3*. Line *4* calls the *obj.learn()* method that extracts the grammar based on the given data and the k-value. The next line calls the `grammar` attribute, and line number *6* shows the extracted grammar[3].

Scanning with a TSL grammar:

```
>>> ntsl = NegTSL()
>>> ntsl.alphabet = ['a', 'b']
>>> ntsl.grammar = [('b', 'b')]
>>> ntsl.tier = ['b']
>>> ntsl.scan('aabaaba')
False
>>> ntsl.scan('aaaabaa')
True
```

The first line initializes negative TSL grammar. Line *2* defines its alphabet. The `grammar` **bb* is given on the line *3*, and the `tier` is provided afterwards. As the result, if we scan the string *aabaaba*, it is rejected due to the **bb* constraint over the tier. But the string *aaaabaa* is well-formed with respect to the grammar, and therefore accepted.

Generating sample with a SP grammar:

```
>>> nsp = NegSP()
>>> nsp.alphabet = ['H', 'L']
>>> nsp.k = 3
>>> nsp.grammar = [('H', 'L', 'H')]
>>> nsp.generate_sample()
>>> nsp.data_sample
['', 'LLH', 'LLHHL', 'LL', 'HLL', 'LLHH', 'L', 'LHL', 'HL', 'H']
```

A negative SP grammar is initialized on line *1*. Lines *2* and *3* define the alphabet and the locality of the grammar. The restriction **HLH* does not allow L to appear in-between two H.[4] The *obj.generate_sample(n, rep)* is called without any arguments, therefore, by default, 10 examples without

[3]The n-grams are represented as tuples and not simply as strings in order to be able to operate not only with single symbols as the elements of the alphabet. For example, such implementation allows to use morphemes or words as the smallest units.

[4]One might see in this example the familiar pattern of the unbounded tonal plateauing that is a famous example of a SP process in phonology; see Jardine (2016) for the discussion of computational properties of tonal patterns.

repetitions are generated. The `data_sample` attribute is called on the line *6*, and the last line demonstrates the generated data.

4.4 Detailed examples

Here, I provide more examples of `kist` toolkit's applications. In every subsection, one can find the exhaustive list of calls of the methods defined for SL, TSL and SP language classes. Please refer to the next section for the implementation-related details.

4.4.1 SL class

In order to initialize a SL grammar, the user needs to choose the class of the desired polarity. The attributes that can be specified are `alphabet`, `grammar`, `k` (the default value is 2), `data`, and `edges` (the default value is [‘>’, ‘<’]). For example, below I initialize positive SL grammar.

```
>>> psl = PosSL()
>>> psl.data = [‘abab’, ‘ab’]
>>> psl.k = 2
```

To extract the alphabet that is used in the data or in the grammar, the *obj.extract_alphabet()* method can be executed.

```
>>> psl.extract_alphabet()
>>> psl.alphabet
[‘a’, ‘b’]
```

The method *obj.learn()* extracts grammar based on the given data.

```
>>> psl.learn()
>>> psl.grammar
[(‘>’, ‘a’), (‘b’, ‘<’), (‘a’, ‘b’), (‘b’, ‘a’)]
```

In order to decide the well-formedness of a string with regards to the given grammar, *obj.scan(string)* method can be used with a string provided as the argument. For example, *ababab* can be generated by the grammar extracted earlier, but **aba* cannot because it contains the illicit bigram **a⋉*.

```
>>> psl.scan(‘ababab’)
True
>>> psl.scan(‘aba’)
False
```

If the user-provided grammar contains useless n-grams and needs to be cleaned, the *obj.clean()* method can do it. It works by converting the grammar to the corresponding finite state machine (FSM), trimming its transitions, and then converting it back to the list of n-grams. In the grammar provided below, the bigram *bc* is useless, because it cannot lead to the end of the string.

```
>>> psl = PosSL()
>>> psl.grammar = [('>', 'a'), ('b', '<'), ('a', 'b'),
    ('b', 'a'), ('b', 'c')]
>>> psl.clean()
>>> psl.grammar
[('b', '<'), ('a', 'b'), ('>', 'a'), ('b', 'a')]
```

The method *obj.fsmize()* builds the FSM corresponding to the grammar. The resulting FSM and its transitions[5] can be accessed by calling `fsm` and `fsm.transitions` attributes, respectively.

```
>>> psl.grammar = [('>', 'a'), ('b', '<'), ('a', 'b'), ('b', 'a
    ')]
>>> psl.fsmize()
>>> psl.fsm
<fsm.FiniteStateMachine object at 0x10380e470>
>>> psl.fsm.transitions
[(('>',), 'a', ('a',)), (('b',), '<', ('<',)), (('a',), 'b', ('b
    ',)), (('b',), 'a', ('a',))]
```

To change the polarity of the grammar, *obj.change_polarity()* can be used. Consider the following example, where the grammar provided above was converted to negative.

```
>>> psl.change_polarity()
>>> psl.grammar
[('a', 'a'), ('b', 'b'), ('>', '<'), ('a', '<'), ('>', 'b')]
```

Lastly, *obj.generate_sample(n, rep)* generates a sample of data with respect to the given grammar. Its arguments are the number of items to be generated (the default value is 10) and the permission to allow repetitions (the default value is *True*). Generated data can be found under the `data_sample` attribute.

[5]Each transition is presented in the form (q, w, q'), where q is the state with the outgoing arc, q' is the state with the incoming arc, and w is the substring that is being read.

```
>>> psl.generate_sample(3,False)
>>> psl.data_sample
['abababab', 'ab', 'abab']
>>> psl.generate_sample(3,True)
>>> psl.data_sample
['ab', 'abab', 'abab']
```

4.4.2 TSL class

Positive or negative TSL grammars can be initialized in the same fashion as the SL ones before. The crucial difference is that the grammar operates not over the original string anymore, but over its *tier*. Therefore if the TSL grammar is provided, the `tier` attribute needs to be listed as well.

```
>>> ptsl = PosTSL()
>>> ntsl = NegTSL()
>>> ntsl.grammar = [('b', 'b')]
>>> ntsl.tier = ['b']
```

If the data is provided, the tier symbols can be determined using the *obj.learn_tier()* method from the *kTSLIA* algorithm designed by Jardine and McMullin (2017). In the example below, the data shows that although the symbol a can be found anywhere, the appearance of b is restricted, and therefore b is included in the list of the tier symbols.

```
>>> ntsl.data = ['abaa', 'aab', 'ba', 'b']
>>> ntsl.learn_tier()
>>> ntsl.tier
['b']
```

If not only the tier, but the grammar also needs to be extracted, one can use the *obj.learn()* method that does both. Consider the extracted grammar for the dataset provided above. There must be only one b in the string, therefore the resulting negative TSL grammar bans two consecutive b ($*bb$), and the empty tier ($*\rtimes\ltimes$).

```
>>> ntsl.data = ['abaa', 'aab', 'ba', 'b']
>>> ntsl.learn()
>>> ntsl.grammar
[('b', 'b'), ('>', '<')]
>>> ntsl.tier
['b']
```

Just as before, the data sample can be generated by using the method *obj.generate_sample(n, rep)* that takes the same type of arguments as the one for SL grammars. In this case, after the tier is generated, a random amount of non-tier symbols is inserted in different positions of the tier.

```
>>> ntsl.generate_sample(3, True)
>>> ntsl.data_sample
['abaaa', 'b', 'aaba']
```

In the polarity of the grammar needs to be changed, one can call the *obj.change_polarity()* method.

```
>>> ntsl.tier = ['b']
>>> ntsl.grammar = [('>', '<'), ('b', 'b')]
>>> ntsl.change_polarity()
>>> ntsl.grammar
[('b', '<'), ('>', 'b')]
```

4.4.3 SP class

The initialization of the SP class is very similar to the SL one, the only difference is the absence of the `edges` attribute. In the code below, as the `data` attribute, I provide the language that is intended to mean *do not have two or more b in a well-formed string.*

```
>>> nsp = NegSP()
>>> nsp.alphabet = ['a', 'b']
>>> nsp.data = ['aaa', 'aaab', 'aabaaa']
```

Learning the desired pattern can be done by using the *obj.learn()* method. Indeed, the extracted negative grammar reflects the desired pattern.

```
>>> nsp.learn()
>>> nsp.grammar
[('b', 'b')]
```

As before, the method *obj.generate_sample(n, rep)* generates the data sample. The first argument indicates the amount of items to be generated (10 by default), and the second one allows or prohibits repetitions (by default, repetitions are prohibited). The generated data is stored in the `data_sample` attribute.

```
>>> nsp.generate_sample(15)
>>> nsp.data_sample
```

```
[‘’, ‘aba’, ‘ab’, ‘abaa’, ‘abaaa’, ‘aaaa’, ‘aaba’, ‘b’, ‘baaa’,
    ‘a’, ‘baaaa’, ‘aab’, ‘ba’, ‘aaa’, ‘aa’]
```

The method *obj.scan(string)* requires a string as its argument, and returns the boolean value that indicates whether that string can be accepted or not.

```
>>> nsp.scan(‘abaaa’)
True
>>> nsp.scan(‘aaabaaaab’)
False
```

The generator that corresponds to the SP class can be represented as a family of FSMs, see Heinz and Rogers (2013). *obj.fsmize()* builds the corresponding family of FSMs, and the transitions of each of the constructed FSMs can be accessed in the following fashion.

```
>>> nsp.fsmize()
>>> for i in nsp.fsm.family:
        print(i.transitions)
[[0, ‘a’, 1], [0, ‘b’, 0], [1, ‘a’, 1], [1, ‘b’, 1]]
[[0, ‘b’, 1], [0, ‘a’, 0], [1, ‘a’, 1]]
```

Lastly, if the polarity of the grammar needs to be changed, one can apply the *obj.change_polarity()* method.

```
>>> nsp.grammar
[(‘b’, ‘b’)]
>>> nsp.change_polarity()
>>> nsp.grammar
[(‘a’, ‘a’), (‘a’, ‘b’), (‘b’, ‘a’)]
```

5 Implementation

In the current section, I provide a more detailed discussion of attributes and methods that are available in the package, as well as explain the structure of the `kist` toolkit.

5.1 Architecture

In this section, I describe the architecture of the `kist` toolkit. I outline the basic classes that are used in the implementation of the package, and define relations between them. Figure 3 shows the class diagram of the toolkit.

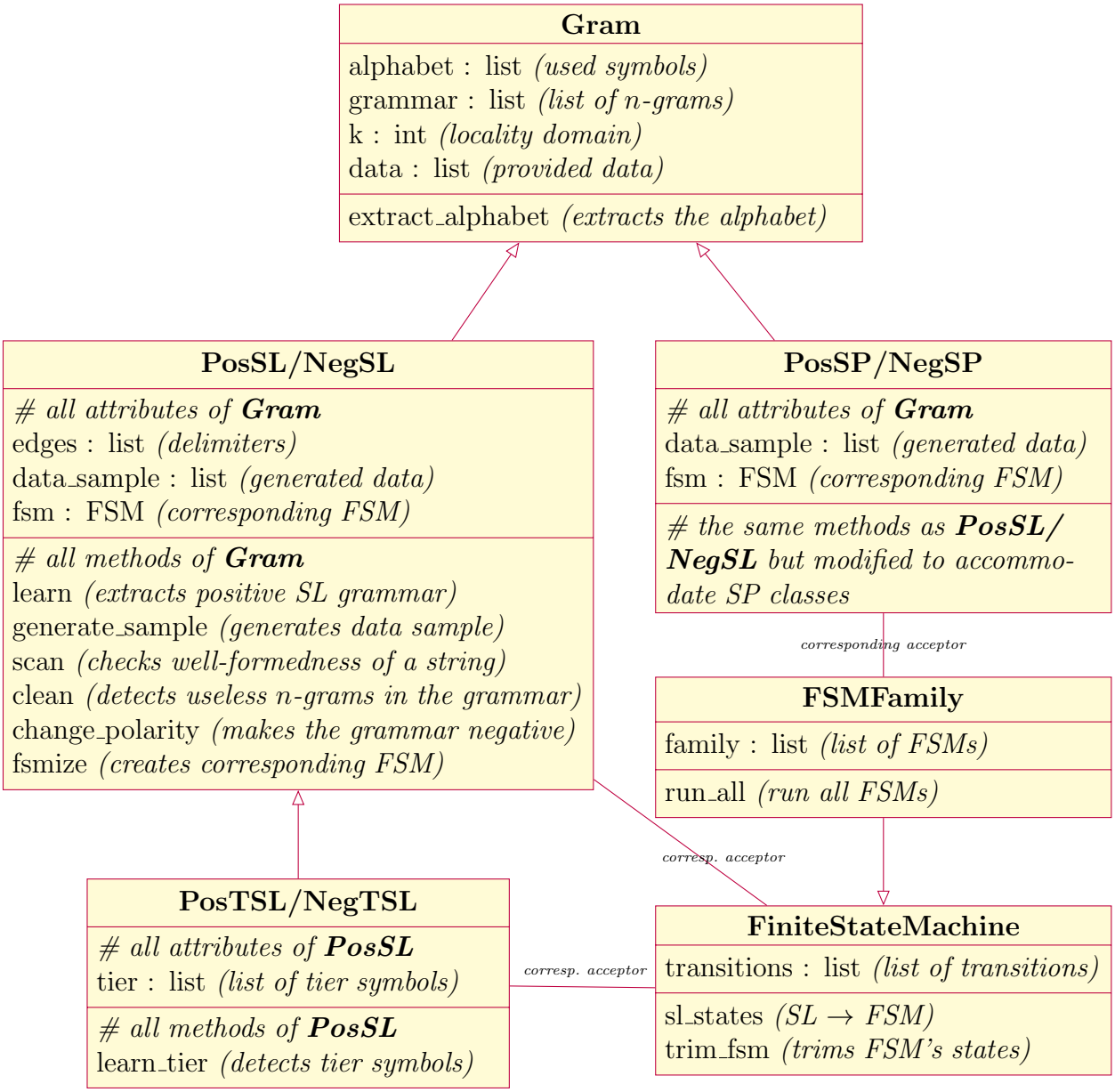


Figure 3: `kist` class diagram

Gram

The abstract class in this package is `Gram`. It defines the basic properties of the grammar at a level that is not different for (T)SL or SP gram-

mars. Such properties include `alphabet`, `grammar`, `k` and `data` attributes and *obj.extract_alphabet()* method, see the next sections for details.

PosSL/NegSL
The two classes of positive and negative SL grammars (`PosSL` and `NegSL`) inherit everything from the general `Gram` class, and add attributes and methods that are specific to the local approach. The acceptor corresponding to these grammars is implemented in `FiniteStateMachine` that is used for scanning a string or generating a data sample.

PosTSL/NegTSL
The classes of positive and negative TSL grammars (`PosTSL` and `NegTSL`) are based on SL classes of the corresponding polarity, but introduce `tier` as one of the specific attributes. The corresponding acceptor is also a finite states machine that operates over the tier representation of a given string.

PosSP/NegSP
The classes of positive and negative SP grammars (`PosSP` and `NegSP`) have different basic relations – precedence instead of successor – and therefore are not related to the local classes. `PosSP` and `NegSP` also inherit basic information from the `Gram` class. But to decide well-formedness of a string, now we need to use a family of finite states machines: `FSMFamily`. The SP grammar corresponds to a family of FSMs, and the string is well-formed with respect to that grammar if and only if it is successfully accepted by every automaton in the family[6].

FiniteStateMachine
This class implements a basic representation of a finite state machine with only one attribute – a list of transitions.

FSMFamily
`FSMFamily` represents a collection of `FiniteStateMachine` objects, and has a method that allows to run all members of the family at once: this is needed to decide well-formedness of a string with respect to a SP grammar.

5.2 Attributes

In this section, I define attributes of language-related and generator-related classes. The table below lists the attributes of classes related to the grammar, i.e. `Gram`, `PosSL/NegSL`, `PosTSL/NegTSL` and `PosSP/NegSP`.

Attributes of language-related classes:			
#	**attribute**	**description**	**relevant classes**
1	*alphabet*	Alphabet is a finite collection of symbols that the grammar is operating with.	Gram, PosSL/NegSL, PosTSL/NegTSL, PosSP/NegSP

Table 1: Attributes of the language-related classes.

Attributes of language-related classes:			
#	**attribute**	**description**	**relevant classes**
2	*grammar*	Grammar refers to a list of banned or allowed substrings or subsequences.	Gram, PosSL/NegSL, PosTSL/NegTSL, PosSP/NegSP
3	*k*	The locality window k refers to the longest substring or subsequence in a grammar. *Default value:* 2.	Gram, PosSL/NegSL, PosTSL/NegTSL, PosSP/NegSP
4	*data*	This attribute stores the data based on which kist extracts the grammar of intended type.	Gram, PosSL/NegSL, PosTSL/NegTSL, PosSP/NegSP
5	*edges*	For grammars operating with the notion of locality, edge symbols are necessary in order to determine initial or final elements of a string. *Default value:* [‘>’, ‘<’].	PosSL/NegSL, PosTSL/NegTSL
6	*data_sample*	The data sample generated by `kist` is being stored in this attribute.	PosSL/NegSL, PosTSL/NegTSL, PosSP/NegSP
7	*fsm*	Generating device that corresponds to the grammar is stored in this attribute.	PosSL/NegSL, PosTSL/NegTSL, PosSP/NegSP
8	*tier*	In TSL classes, tier contains the list of tier symbols.	PosTSL/NegTSL

Table 2: Attributes of the language-related classes.

By default, the value of the attribute `k` is 2, and the edges that are use by `kist` are ['>', '<'] unless redefined. In order to perform the grammar extraction, `data` is the required attribute, and in the case of scanning or sample generation, the `grammar` attribute must not be empty.

Another table below describes attributes of the generator-related classes, i.e. `FiniteStateMachine` and `FSMFamily`. A single FSM corresponds to SL and TSL languages, and a family of ones is required to accept SP languages.

Attributes of generator-related classes:			
#	**attribute**	**description**	**relevant classes**
9	*transitions*	List of transitions of a finite state machine.	FiniteStateMachine
10	*family*	A collection of finite state machines.	FSMFamily

Table 3: Attributes of the generator-related classes.

5.3 Methods

In this section, I present methods of each one of the classes discussed above more elaborately, and explain the logic behind their implementations. Each method is either *user-friendly*, i.e. meant to be used by a user; or *internal*, i.e. written to outsource certain tasks to a sub-function.

5.3.1 SL classes

Here I discuss methods defined for positive and negative SL grammar classes.

obj.extract_alphabet() *[user-friendly]*
This methods extracts alphabet from the data or the grammar, depending on what is provided. No arguments are required by this method.

```
>>> a = PosSL()
>>> a.data = ['abab', 'bac']
>>> a.extract_alphabet()
>>> a.alphabet
['a', 'b', 'c']
```

[6] Jeffrey Heinz, p.c.

obj.learn() *[user-friendly]*
This method extracts n-grams from the provided data, and updates the `alphabet` attribute. If the `data` attribute is empty, the error is raised. As for the method-internal arguments, nothing is required.

```
>>> a.learn()
>>> a.grammar
[('>', 'a'), ('b', '<'), ('>', 'b'), ('a', 'c'), ('b', 'a'),
    ('c', '<'), ('a', 'b')]
```

obj.generate_sample(n=10, rep=True) *[user-friendly]*
This method generates a data sample with respect to a given grammar. Two arguments can be provided: n is the number of strings to be generated (by default, this number is 10), and *rep* allows or prohibits repetitions (by default, the repetitions are allowed). As the result, k random strings are generated.

```
>>> b = NegSL()
>>> b.alphabet = ['a', 'b']
>>> b.grammar = [('a', 'a'), ('b', 'b')]
>>> b.generate_sample(n=5, rep=True)
>>> b.data_sample
['ba', 'b', '', 'b', 'baba']
```

If the parameter *rep* is False, then the generation process is repeated until the number of generated unique strings is equal to n.

```
>>> b.generate_sample(n=5, rep=False)
>>> b.data_sample
['', 'a', 'ab', 'ababa', 'b']
```

Generation is performed by constructing a FSM that corresponds to the grammar, and then creating a *state map* that indicates which symbols can serve as a continuation of a $k-1$ prefix, where k is the locality of the grammar. See the *obj.state_map()* method for details.

obj.scan(string) *[user-friendly]*
This methods scans the string that is passed as an argument, and tells whether it is well-formed. The resulting value is determined based on whether the set of n-grams of the string is a subset of the grammar if the grammar is positive. If the grammar is negative, then the string is considered well-formed if none of the elements of the grammar are found in the string.

```
>>> a.grammar = [('>', 'a'), ('b', '<'), ('a', 'b'), ('b', 'a')]
    # positive grammar
>>> b.grammar = [('a', 'a'), ('b', 'b')] # negative grammar
```

```
>>> a.scan('aabb')
False
>>> b.scan('aabb')
False
>>> a.scan('abab')
True
>>> b.scan('abab')
True
```

obj.clean() *[user-friendly]*
This method checks whether all n-grams of the grammar are "useful". By "useful", I mean that they can be somehow used in the grammar. For example, in the positive SL grammar $\{>a,\ ab,\ ba,\ b<,\ ca\}$, the bigram ca is useless: there are no transitions that lead to a state from which the symbol c can be read.

```
>>> a = PosSL()
>>> a.grammar = [('>', 'a'), ('a', 'b'), ('b', 'a'), ('c', 'a'),
    ('b', '<')]
>>> a.clean()
>>> a.grammar
[('b', '<'), ('a', 'b'), ('>', 'a'), ('b', 'a')]
```

If there are no "useless" n-grams in the grammar, the grammar is left unchanged.

```
>>> a.grammar
[('b', '<'), ('a', 'b'), ('>', 'a'), ('b', 'a')]
>>> a.clean()
>>> a.grammar
[('b', '<'), ('a', 'b'), ('>', 'a'), ('b', 'a')]
```

This method works by running the corresponding FSM to the SL grammar from the start to the end and seeing which states can be accessed. Then the automaton is run in the opposite direction, and the states that can be accessed are marked again. The corresponding automaton is then trimmed: only the states that were marked twice remain, see *obj.trim_fsm()* method. The resulting grammar is constructed based on the trimmed version of the automaton.

obj.change_polarity() *[user-friendly]*
This method changes the polarity of the grammar to the opposite one by creating all possible k-grams based on the alphabet of the language, and then taking the relative complement of the original grammar.

```
>>> b.grammar
[('a', 'a'), ('b', 'b')]
>>> b.change_polarity()
>>> b.grammar
[('>', '<'), ('b', 'a'), ('a', '<'), ('a', 'b'), ('b', '<'),
    ('>', 'b'), ('>', 'a')]
```

In order to be consistent, the internal type name of the class is also rewritten to the opposite one.

```
>>> b.__class__
<class '__main__.NegSL'>
>>> b.change_polarity()
>>> b.__class__
<class '__main__.PosSL'>
```

obj.generate_item() *[internal]*
This internal method returns a well-formed word with respect to the provided grammar. It works by constructing a *state map* and then moving through it, see *obj.state_map()* method explained later.

```
>>> b.grammar
[('a', 'a'), ('b', 'b')]    # negative grammar
>>> b.generate_item()
'babab'
```

obj.state_map() *[internal]*
The method *obj.state_map()* creates a dictionary where every prefix of the $k-1$ length is a key, and its value is a list of symbols that can follow this prefix. Basically, this automata represents the list of possible transitions without running the actual automaton.

```
>>> b.grammar
[('a', 'a'), ('b', 'b')]        # negative grammar
>>> b.state_map()
{'a': ['<', 'b'], 'b': ['a', '<'], '>': ['<', 'b', 'a']}
```

This dictionary is constructed by generating all possible $k-1$ sequences with respect to the given grammar, and then passing them though the automaton corresponding to the grammar. The list of possible continuations is created by listing all possible moves from the state that corresponds to the prefix under consideration.

obj.fsmize() *[internal]*
This method constructs a finite state machine based on the SL grammar. Every transition of the FSM corresponds to a n-gram of the positive grammar.

The transitions are of the following shape: $\langle q_0, w, q_1 \rangle$, where q_0 and q_1 are the states corresponding to $k-1$ prefix and suffix of the n-gram respectively, and w is the last symbol of the n-gram.

```
>>> a.grammar = [('b', '<'), ('a', 'b'), ('>', 'a'), ('b', 'a')]
      # positive grammar
>>> a.fsmize()
>>> a.fsm.transitions
[(('b',), '<', ('<',)), (('a',), 'b', ('b',)), (('>',), 'a', ('a
    ',)), (('b',), 'a', ('a',))]
```

For the sake of simplicity while running the automaton, even when the grammar is negative, the FSM corresponds to the positive version of the grammar.

```
>>> b.grammar = [('a', 'a'), ('b', 'b')]   # negative grammar
>>> b.fsmize()
>>> b.fsm.transitions
[(('>',), '<', ('<',)), (('b',), 'a', ('a',)), (('a',), '<',
    ('<',)), (('a',), 'b', ('b',)), (('b',), '<', ('<',)),
    (('>',), 'b', ('b',)), (('>',), 'a', ('a',))]
```

obj.annotate_data(string, k) *[internal]*
In order to process a string in a local fashion, it needs to be annotated with start and end symbols, and this is the task of the *obj.annotate_data(string, k)* method. The argument *string* is the string to be annotated, and k is the locality of the grammar. As the output, the string is annotated with $k-1$ start and end symbols.

```
>>> a.annotate_data('subregular', 2)
'>subregular<'
```

obj.ngramize_data(k, strings) *[internal]*
This method takes a list of strings and a k-value as arguments, and returns a list of k-grams that are used in the given strings.

```
>>> a.ngramize_data(2, ['', 'ab'])
[('>', '<'), ('a', 'b'), ('b', '<'), ('>', 'a')]
```

obj.ngramize_item(string, k) *[internal]*
This method reads a string, and returns a list of k-grams that this string consists of.

```
>>> a.ngramize_item('abcd', 2)
[('b', 'c'), ('a', 'b'), ('c', 'd')]
```

obj.build_ngrams(transitions) *[internal]*
This method takes a list of transitions as input, and returns the corresponding grammar as output. It is useful for cleaning the grammar: after the corresponding FSM is trimmed, this function is called to construct a clean version of the grammar based on the FSA transitions.

```
>>> a.fsm.transitions
[(('b',), '<', ('<',)), (('a',), 'b', ('b',)), (('>',), 'a', ('a
    ',)), (('b',), 'a', ('a',))]
>>> a.build_ngrams(a.fsm.transitions)
[('b', '<'), ('a', 'b'), ('>', 'a'), ('b', 'a')]
```

5.3.2 TSL classes

For the TSL classes, all methods apart from the ones discussed in this section are the same as the ones presented in the previous one (5.3.1 "SL classes").

obj.learn() *[user-friendly]*
This method extracts the list of tier symbols and the tier grammar from the given data. The algorithm based on which the method is designed is based on the one by Jardine and McMullin (2017).

At the first step, the list of tier symbols is being determined, and then the list of n-gramized tier sequences is extracted. If the grammar is positive, this list is the TSL grammar.

```
>>> p.data = ['abaa', 'aab', 'ba', 'b']
>>> p.learn()
>>> p.grammar
[('>', 'b'), ('b', '<')]
```

If the grammar is negative, then the complement of that list is taken.

```
>>> n.data = ['abaa', 'aab', 'ba', 'b']
>>> n.learn()
>>> n.grammar
[('>', '<'), ('b', 'b')]
```

obj.scan(string) *[user-friendly]*
This methods scans the string that needs to be passed as an argument, and tells whether it is well-formed with respect to the provided grammar. The resulting value is determined based on whether the set of n-grams of the tier representation of the string is a subset of the grammar if the grammar is positive. If the grammar is negative, then the string is well-formed if none of the elements of the grammar are found on the tier of the string.

```
>>> p.grammar = [('>', 'b'), ('b', '<')]   # positive grammar
>>> p.scan('abba')
False
>>> p.scan('aba')
True
```

obj.generate_item() *[internal]*
This method first generates a well-formed tier sequence of a word using the parental *obj.generate_item()* method that is inherited from the SL class. Afterwards, non-tier symbols are randomly inserted in-between the generated tier symbols.

```
>>> p.grammar = [('>', 'b'), ('b', '<')]   # positive grammar
>>> p.generate_item()
'aaaba'
```

obj.erasing_function() *[internal]*
Given the provided `data` and `tier` attributes, this method returns the list of tier sequences of the given data.

```
>>> p.data = ['abaa', 'aaa']
>>> p.tier = ['b']
>>> p.erasing_function()
['', 'b']
```

obj.tier_image(string, tier) *[internal]*
Given a string and a list of tier symbols, this method returns the copy of the string with all its non-tier symbols erased.

```
>>> p.tier = ['b']
>>> p.tier_image('ababa', p.tier)
'bb'
>>> p.tier_image('aaaa', p.tier)
''
```

obj.test_insert(symbol, ngrams, ngrams_less) *[internal]*
This method is one of the two tests that a symbol needs to pass in order to be removed from the tier alphabet. Given a symbol and all *k+1*-grams that contain it, one should try to remove the given symbol from them. If all of the resulting k-grams can be found in the data, this symbol might be not a tier-symbol, but if it is not true, the symbol under consideration is definitely a tier symbol, see Jardine and McMullin (2017).

obj.test_remove(symbol, ngrams, ngrams_more) *[internal]*
This method is the second one of the two tests that a symbol needs to pass in order to be removed from the tier alphabet. Given the symbol and all *k-1*-grams of the data, one should try to insert this symbol in all positions of those *k-1*-grams. If all of the resulting *k*-grams can be found in the data, this symbol might be not a tier-symbol, but if it is not true, the symbol under consideration is a tier symbol, see Jardine and McMullin (2017).

5.3.3 SP classes

This section discusses the methods that are defined for SP languages. Methods *obj.generate_sample(n, rep)*, *obj.extract_alphabet()* and *obj.change_polarity()* are the same as discussed before, therefore I do not repeat them here.

obj.learn() *[user-friendly]*
This method extracts all possible or impossible subsequences from the data depending on the polarity of the intended grammar, therefore the data attribute must not be empty. If the grammar is positive, then it simply extracts all *k*-piecewise subsequences from the data.

```
>>> a = PosSP()
>>> a.data = ['aaabaa']
>>> a.learn()
>>> a.grammar
[('a', 'a'), ('a', 'b'), ('b', 'a')]
```

If the grammar is negative, the complement of possible *k*-piecewise subsequences is taken.

```
>>> b = NegSP()
>>> b.data = ['aaabaa']
>>> b.learn()
>>> b.grammar
[('b', 'b')]
```

obj.subsequences(string) *[internal]*
This method extracts all subsequences of the length *k* out of the given string.[7]

```
>>> b.subsequences('subreg')
[['s', 'u'], ['s', 'b'], ['s', 'r'], ['s', 'e'], ['s', 'g'], ['u
    ', 'b'], ['u', 'r'], ['u', 'e'], ['u', 'g'], ['b', 'r'], ['b
    ', 'e'], ['b', 'g'], ['r', 'e'], ['r', 'g'], ['e', 'g']]
```

[7]Every position in a string can be represented as a layer of a graph, and the task is to find all paths within this graph that are going through exactly *k* nodes.

obj.fsmize() *[internal]*
This method creates a FSM family that corresponds to a given SP grammar. First, it makes sure that the grammar is extracted. Second, it creates the template of the constructing family[8]. Third, it runs grammar sequences through the states of each automaton in the family marking the transitions that were taken. Lastly, it creates the copy of the FSM family where all the transitions that were never taken are trimmed away.

The transitions of every machine within the corresponding FSM family can be viewed in the following way:

```
>>> b.fsmize()
>>> for i in b.fsm.family:
        print(i.transitions)
[[0, 'a', 1], [0, 'b', 0], [1, 'a', 1], [1, 'b', 1]]
[[0, 'b', 1], [0, 'a', 0], [1, 'a', 1]]
```

obj.scan(string) *[user-friendly]*
This methods scans the string that is passed as an argument, and says whether it is well-formed. A string is well-formed if and only if every automaton in the FSM family can accept it.

```
>>> b.scan('ababaa')
False
>>> b.scan('abaa')
True
```

obj.generate_item() *[internal]*
This function generates a well-formed item with respect to the given grammar. Until it is randomly selected to stop generating the sequence, it randomly adds symbols to the string that is being generated, and at every step it checks that the resulting string is still well-formed by passing it through the *obj.scan(string)* method.

```
>>> n.generate_item()
'aaaaba'
```

[8] In total, there are $|\Sigma^{k-1}|$ FSAs constructed. See Heinz and Rogers (2013) on the algorithm of the FSM family construction. The current implementation fully relies on this algorithm, and the only modification is that now it is closed under subsequence in order to improve the learning results.

5.3.4 FSM class

In this and the next sections, I do not show code snippets of the method calls, because the discussed functionality is designed to be used within (T)SL or SP classes, and the access to these methods without defining a language-related task is quite complicated. More details on the implementation are given in the doc-strings of the methods.

obj.sl_states(grammar) *[internal]*
This method takes a strictly local grammar as input, and creates an FSM with the list of transitions that corresponds to the given grammar. This function is employed by *obj.fsmize()* of the SL and TSL classes.

obj.sp_template(seq, alphabet, k) *[internal]*
It creates the FSM family template for a given SP grammar's configuration. Given the sequence of symbols to be represented in a current FSM, alphabet, and k, this method creates a FSM with the transitions of the shape $\langle q_0, w, q_1, Bool \rangle$, where q_0 and q_1 are the states, w is one of the symbols from the *seq*, and *Bool* is a boolean value that indicates whether the transition was taken or not. Upon initialization, all *Bool* are set to False. This method is used by *obj.fsmize()* within the SP class.

obj.run_sp(string) *[internal]*
This method runs a string through the SP automaton and returns a boolean value depending on the well-formedness of the string with respect to the machine.

obj.run_learn_sp(string) *[internal]*
It passes the given string through the template automaton, and marks the transitions that were taken by that string. This is required in order to generate an FSM family corresponding to the intended SP grammar.

obj.sp_clean() *[internal]*
This method trims away the transitions that were never taken. It is required in order to construct an FSM family for the intended SP grammar.

obj.trim_fsm(markers=['>', '<']) *[internal]*
This method detects useless states of the FSM and trims them away. A state is considered *useless* if there is either no way to enter this state (i.e. to come from the initial state), or no way to exit from it (i.e. to reach the final state after passing through that state). Calling this method results in trimming useless states and transitions within the current FSM.

obj.__accessible_states(transitions, markers) *[internal]*
This method runs from the initial (or, if the machine is mirrored, final) state of the finite state machine, and detects which states can be reached from there. Other states are being trimmed away.

5.3.5 FSM family class

obj.run_all(string) *[internal]*
This method passes the given string through all the automata within the FSM family simultaneously, and returns True if all of the machines can accept that string, otherwise it returns False, i.e. rejects the string.

6 Conclusion and future work

In this paper, I presented a Python toolkit `kist` that contains instruments for working with subregular languages. Currently, the toolkit includes tools for working with subregular classes of strictly local, tier-based strictly local, and strictly piecewise languages. More precisely, `kist` contains learners, scanners, sample generators, and several other tools.

At the current moment, I am improving the performance of already implemented tools, and adding probabilistic versions to every one of the grammars listed above. Apart from extending the toolkit, I am turning it into a Python package. In the nearest future, I will add tools that allow one to work with subregular transducers (Chandlee and Heinz, 2018).

The importance of such a toolkit can be demonstrated by the wide use and fast development of subregular techniques. Their applications can be found in linguistics, robotics, and neural network research. However, it is difficult to manually generate or analyze large amounts of data and, therefore, this toolkit automates tasks of researchers that work in related areas.

References

Alëna Aksënova and Aniello De Santo. in press. Strict locality in morphological derivations. In *Proceedings of the 53rd meeting of Chicago Linguistics Society (CLS 53)*.

Alëna Aksënova and Sanket Deshmukh. 2018. Formal restrictions on multiple tiers. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2018*, pages 64–73. Association for Computational Linguistics.

Alëna Aksënova, Thomas Graf, and Sedigheh Moradi. 2016. Morphotactics as tier-based strictly local dependencies. In *Proceedings of the 14th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 121–130. Association for Computational Linguistics.

Enes Avcu, Chihiro Shibata, and Jeffrey Heinz. 2017. Subregular complexity and deep learning. In *Proceedings of the Conference on Logic and Machine Learning in Natural Language*.

Jane Chandlee. 2017. Computational locality in morphological maps. *Morphology*, 27:599–641.

Jane Chandlee and Jeffrey Heinz. 2018. Strict locality and phonological maps. *Linguistic Inquiry*, 49(1):23–60.

Aniello De Santo and Thomas Graf. 2017. Structure sensitive tier projection: Applications and formal properties. Manuscript. Stony Brook University.

Aniello De Santo, Thomas Graf, and John Drury. 2017. Evaluating subregular distinctions in the complexity of generalized quantifiers. Poster presented at the ESSLLI Workshop on Quantifiers and Determiners (QUAD 2017), July 17 – 21, University of Toulouse, France.

Jie Fu, Jeffrey Heinz, and Herbert G. Tanner. 2011. An algebraic characterization of strictly piecewise languages. In *Theory and Applications of Models of Computation*, volume 6648 of *Lecture Notes in Computer Science*, pages 252–263. Springer Berlin/Heidelberg.

Brian Gainor, Regine Lai, and Jeffrey Heinz. 2012. Computational characterizations of vowel harmony patterns and pathologies. In *The Proceedings of the 29th West Coast Conference on Formal Linguistics*, pages 63–71.

Thomas Graf. 2017. Subregular morpho-semantics: The expressive limits of monomorphemic quantifiers. Invited talk, December 15, Rutgers University, New Brunswick, NJ.

Thomas Graf and Jeffrey Heinz. 2016. Tier-based strict locality in phonology and syntax. Manuscript. Stony Brook University and University of Delaware.

Jeffrey Heinz. 2011a. Computational phonology part I: Grammars, learning, and the future. *Language and Linguistics Compass*, 5(4):140–152.

Jeffrey Heinz. 2011b. Computational phonology part II: Grammars, learning, and the future. *Language and Linguistics Compass*, 5(4):153–168.

Jeffrey Heinz. 2015. The computational nature of phonological generalizations. Manuscript. University of Delaware.

Jeffrey Heinz, Chetan Rawal, and Herbert G. Tanner. 2011. Tier-based strictly local constraints for phonology. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, pages 58–64, Portland, USA. Association for Computational Linguistics.

Jeffrey Heinz and James Rogers. 2013. Learning subregular classes of languages with factored deterministic automata. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, pages 64–71, Sofia, Bulgaria. Association for Computational Linguistics.

Adam Jardine. 2016. Computationally, tone is different. *Phonology*, 33(2):247–283.

Adam Jardine and Kevin McMullin. 2017. Efficient learning of tier-based strictly k-local languages. *Lecture Notes in Computer Science*, 10168:64–76.

Robert McNaughton and Seymour A. Papert. 1971. *Counter-Free Automata (M.I.T. Research Monograph No. 65)*. The MIT Press.

Chetan Rawal, Herbert G. Tanner, and Jeffrey Heinz. 2011. (Sub)regular robotic languages. In *IEEE Mediterranean Conference on Control and Automation*, pages 321–326.

James Rogers. 2018. On the cognitive complexity of phonotactic constraints. Slides of a Stony Brook Colloquium. March 23.

Department of Linguistics
Stony Brook University
Stony Brook, NY 11794-4376
alena.aksenova@stonybrook.edu