派業歷

PyElly User's Manual

For Release v2.2.1

9 September 2021

Clinton P. Mah

Walnut Creek, CA 94595

Table of Contents

1. Introduction

PyElly is an open-source software toolkit for creating computer scripts to analyze and rewrite English and other natural language text. This processing will of course fall far short of the talking robot fantasies of Hollywood, but with a little effort, you can still quickly put together many nontrivial applications to do useful linguistic work. In particular, PyElly can serve as a preprocessor to clean up the many pesky low-level details of natural language that often burden deeper analyses of text data.

PyElly can also give you some firsthand experience with the nuts and bolts of computational linguistics. You can quickly write scripts to conjugate French verbs, rephrase information requests into a formal query language, compress messages for texting, extract names and other entities from a text stream, or even re-create the storied Doctor simulation of Rogerian psychoanalysis.

Such natural language applications have been built since computers were only a millionth as powerful as they are today. Natural language processing (NLP) remains quite challenging, however, and most toolkits to support NLP require heavy lifting to develop the logic and interpretive frameworks to accomplish even something simple. PyElly can help here through a broad array of ready-made tools and resources, all integrated within a single free open-source package.

Why do we need yet another NLP toolkit? To begin with, a complete natural language solution is still far off, and so we can gain from a diversity of reliable methods to break down and manipulate text data. Also, though PyElly is all new code, its core is really a legacy system of components dating back more than 40 years. This sounds ancient, but language changes slowly, and mature linguistic software tools and resources can be of service even with today's text data.

The impetus for PyElly and its predecessors came from observing that many different natural language applications face similar issues. For example, information retrieval and machine learning with text data can both work better when we can reduce the words in target text to their roots. So, instead of contending with variants like RELATION, RELATIONAL, RELATIVELY, and RELATING, you can map them all to RELATE. This is the familiar stemming problem, but available free resources to recognize such word variations are often disappointing.

Stemmers are fairly straightforward to build, but it takes time and commitment to do a good job, and no one really wants to repeat this work from scratch in every new project. That is true of many other low-level language processing capabilities as well. One should at least like to pull together some kind of NLP software library of basic resources, but even better would be to integrate them all more closely. PyElly does that.

The current implementation of PyElly is intended primarily for educational use and so was written entirely in Python, currently a favored first programming language in high schools. This should allow students to adapt and incorporate PyElly code into class projects that have to be completed fairly quickly. PyElly can be of broader interest,

though, because of its range of natural language support, including stemming, tokenizing, entity extraction, vocabulary management, sentence recognition, idiomatic interpretation, syntax-driven analysis, and ambiguity handling.

The operation of PyElly revolves around classic rule-based computational linguistics. It will require some language expertise because you need to define all the details of the processing that you want, but if you are working with English input, many of the basics here have been prebuilt in PyElly. The standard PyElly distribution also includes the language definition rules for ten different nontrivial example applications to get you started in constructing your own.

The current PyElly package consists of a set of Python modules in sixty-seven source files. The code should run on any computer platform with a Python 3.8 interpreter, including Windows 8 and 10, Linux, Mac OS X and other flavors of Unix, IOS for iPhone and iPad, and Android. The PyElly source is downloadable from GitHub under a standard BSD license; you may freely modify and extend it as needed. Though intended mainly for education, commercial or government usage of PyElly is unrestricted.

For recognizing just a few dozen sentences, PyElly is probably overkill; you could handle them directly by writing custom code in any popular programming language. More often, however, the possible input sentences will be too many to list out fully, and so you must characterize them more generally through rules describing how the words you expect to see are formed, how they combine in text, and how they are to be interpreted. PyElly lets you manage such details of language definition in a systematic way.

PyElly is a kind of translator: it reads in plain text, analyzes it, and writes out a transformation according to the rules that you supply. So an English sentence like "She goes slowly" might be rewritten in French as "Elle va lentement" or in Chinese as 她走慢慢地. Or you might reduce the original sentence to just "slow" by stripping out suffixes and words of low content. Or you may want to rephrase the sentence as a question like "Does she go slowly?" All this can be done with suitable rules to guide PyElly.

PyElly rules encode linguistic knowledge of many types. The main ones will define a grammar and vocabulary for the sentences of an input language plus associated semantic procedures for rewriting those sentences to produce target output. Creating such rules requires some trial and error, but usually should be no more difficult than setting up macros in a word processor. PyElly will get you started quickly and also will provide debugging aids to help track down the problems inevitably arising.

Many natural language systems, especially those in academic research, aim at the most thorny problems in language interpretation. These are opportunities for impressive processing gymnastics and often lead to dense theoretical papers without necessarily producing anything for everyday use. PyElly aims instead to be simple and down to earth. In response to well-known tough sentences like "Time flies like an arrow," it is quite all right for PyElly or any other practical NLP system to respond just with "Huh?"

PyElly is compact enough to run on mobile devices with no cloud connection. Excluding the Python environment, compiled PyElly code along with encoded rules and other data for an application should typically require less that 500 Kbytes of storage, depending on the number of rules actually defined. A major project may involve hundreds of grammar rules and tens of thousands of vocabulary elements, but some useful text analyses require just a few dozen rules and little explicit domain vocabulary.

What is a grammar, and what is a vocabulary? A vocabulary establishes the range of words you want to recognize; a grammar defines how those words can be put into meaningful sentences. You may also idiomatically rephrase particular input word sequences prior to analysis as well as define patterns for making sense of various kinds of entities not normally listed in a vocabulary: 800 telephone numbers, or Russian surnames ending in -OV or -OVA, or generic drug names ending in -MAB or -VIR.

This manual will explain how to do all of this and also introduce some essentials of language and language processing that any NLP practitioner should know. To take advantage of PyElly, you should be able to create and edit text files for the platform of your choice and set up file directories there so that PyElly can find your rules. PyElly currently has to run from a command line. For advanced work, you should also be comfortable with Python coding. In an ideal world, an interactive development environment (IDE) could make everything easier here, but that is yet to be.

Currently, PyElly is set up to read and process ASCII plus Unicode characters from the Latin-1 Supplement, Latin Extender A, and parts of Latin Extender B plus extra punctuation and spaces. These include most of the letters with diacritical marks used in European languages written with Latin alphabets. For example, PyElly knows that é is a vowel, that ß is a letter, that Œ is the uppercase form of œ, and that ² is a digit. This can be helpful even for nominally English text, where we often encounter terms with foreign spellings like NÉE or RÉSUMÉ or ÆGIS.

PyElly as of version v1.6 can also process non-alphabetic Unicode text input. Currently, this has been done only for text consisting only of Chinese characters (字) plus punctuation and Arabic numerals. The Chinese processing option will ignore all whitespace. If you want to recognize Chinese multi-character words (詞) as single PyElly tokens, they will have to be predefined in a PyElly vocabulary rule table.

As any beginning student of a foreign language soon learns, its rules are inevitably messy. Irregularities will trip up anyone or anything trying to speak or write mechanically from a simple grammar. PyElly users will face the same challenge, but by working generally at first and then adding special rules to deal with exceptions as they turn up, we can evolve our linguistic chops to reach some useful level of parlance. There is no royal road to NLP, but persistence can lead to progress over time. PyElly will help you keep on track in making a sustained effort in the necessary rule definition.

You need not be an experienced linguist or computer programmer to develop a PyElly application; and this manual aims at being understandable by non-experts. The only requirements for users are basic computer literacy as expected of 21st-Century high

school graduates, linguistic knowledge as might be gained from a first course in any foreign language, and willingness to learn. You can start out with simple kinds of PyElly processing and move on to more complex translations as you gain experience.

In addition to this introduction, the PyElly User's Manual consists of fourteen other major sections plus seven appendices. Sections 2 through 7 should be read in sequence as they provide a tutorial on how get started in using PyElly. Sections 8 through 11 deal with advanced features for developing complex applications. Section 12 explains PyElly parsing, Section 13 lays out some practical strategies and tips for PyElly application development, and Section 14 describes some current and future PyElly applications. Section 15 wraps everything up.

PyElly ("Python Elly") was inspired by the Eliza system created by Joseph Weizenbaum over 50 years ago for modeling natural language conversation, but PyElly code has a different genesis. Its implementation in Python is the latest in a series of successive natural language toolkits going back four generations: Jelly (Java, 1999), nlf (C, 1984 and 1994), the Adaptive Query Facility (FORTRAN, 1981), and PARLEZ (PDP-11 assembly language, 1977). The basic PyElly parsing algorithm and the ideas of cognitive and generative semantics come from Vaughan Pratt's LINGOL (LISP, 1973). Frederick Thompson's REL system (1972) also influenced the design of PyElly.

The main PyElly website is at https://sites.google.com/site/pyellynaturallanguage/ . This introduces PyElly and shows some actual translations with language definition rules from different PyElly example applications. The latest PyElly source code, language definition rule files for example applications, test data and procedures, and documentation are on GitHub. There are two versions. The older one written in Python 2.7 is at https://github.com/prohippo/pyelly.git . The newer one written in Python 3.8 is at https://github.com/prohippo/pyellytoo.git .

The PDF of this user's manual can be downloaded from the pyellytoo GitHub repository. It is different from the user manuals for PyElly up to release v1.6.*; you can get the latest of those manuals from the older pyelly GitHub repository. From now on, please use the code and documentation in the pyellytoo repository. PyElly from release v2.0 onward will be written in Python 3.*. That new code is incompatible with the Python 2.7 interpreter employed previously by PyElly.

2. The Syntax of a Language

A language is as a way of stringing together words or other symbols to form sentences that other people can make sense of in some context. In general, not all combinations of symbols will make a meaningful sentence; for example, "Cat the when" is nonsense as English. To define a language that you want PyElly to process, you must first identify those combinations of symbols that do make sense and then assign suitable interpretations to them.

If a language is small enough, such as the repertory of obscene gestures, we can simply list all its possible "sentences" and say in expletives what each of them means. Most nontrivial natural languages, though, have so many possible sentences that this approach is impractical. Instead one must try to find the regular structures of a language so that a computational linguist can then formally characterize the language much more concisely than by listing all possible sentences.

The structural description of a language is called a grammar. It establishes what the building blocks of a language are and how they form simple structures, which in turn combine into more complex structures. Almost everyone has studied grammar in school, but formal grammars have to go into much greater detail. They commonly are organized around sets of syntactic rules describing all the particular kinds of language structure to be expected in sentences. Such syntactic rules will provide a basis for both generating and recognizing the sentences of a language with a computer.

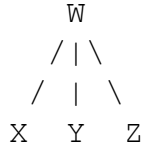
In linguistics, a grammar expresses how one or more structures can come together to produce a new composite structure. This is described through syntactic rules, commonly written with an arrow notation as follows:

```
W->X Y Z
```

This rule states that a `W`-structure can be composed of an `X`-structure followed by a `Y`-structure followed by a `Z`-structure; for example, a noun phrase in English can consist of a number, followed by an adjective, and followed by a noun:

```
NOUNPHRASE->NUMBER ADJECTIVE NOUN
```

There is nothing mysterious here; it is like the kind of sentence diagramming once taught in junior high school and now coming back into vogue. We could draw the following equivalent diagram on a blackboard for the syntactic rule above.



where a `W` can in turn be part of a higher-level structure and `X`, `Y`, and `Z` can also split out further into various substructures. Using a tree to describe syntactic structure is fine, but the arrow notation will be more compact and easier to enter from a keyboard, especially as sentence syntax grows more complex.

The structure-types `W`, `X`, `Y`, `Z` , and so forth roughly correspond to the parts of speech from our school days, but will be interpreted more generally in PyElly. In this user manual, we shall refer to them as syntactic types or syntactic categories. The ones that you need for a PyElly grammar will depend on your particular target application. Some applications may require up to a hundred different syntactic categories, going far beyond `NOUN` and `VERB`.

Syntactic rules for linguistic analysis can be much more complicated than `W->X Y Z`, but PyElly will work with just three forms of rules:

```
X->token
X->Y
X->Y Z
```

where `X`, `Y`, and `Z` are syntactic types and `token` is a word or some other kind of vocabulary element like a number or some arbitrary alphanumeric identifier.

These three types are enough to re-express a more complex rule like

```
R->A B C
```

This is done by dividing the more complex rule into multiple more simple rules

```
R->A T
T->B C
```

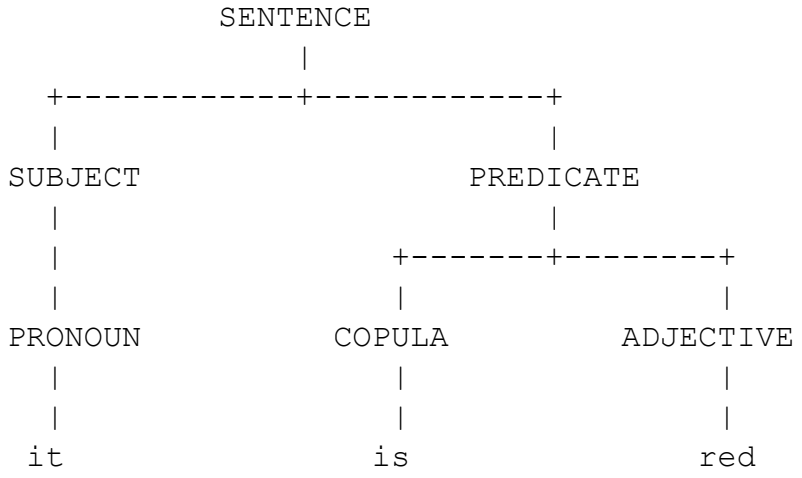
where `T` is a unique intermediate syntactic structure introduced solely to stand for a `B` followed by a `C`. It is not really a part of speech.

Here is a set of PyElly grammar rules that might be employed to describe the structure of the sentence "It is red."

```
SENTENCE->SUBJECT PREDICATE
SUBJECT->PRONOUN
PRONOUN->it
PREDICATE->COPULA ADJECTIVE
COPULA->is
ADJECTIVE->red
```

This shows all three forms of PyElly restricted syntax rules. The ordering of the rules does not matter. The names of syntactic types like `SUBJECT` or `COPULA` are arbitrary except for a few reserved names to be explained below.

The structure of a sentence as implied by these rules can be expressed graphically as a labeled tree diagram, where the root type is `SENTENCE` and where branching corresponds to splitting into constituent substructures. By convention, the tree is always shown upside down, and so the lowest part of a tree will show the individual words of the sentence. For example, the sentence "It is red" would have the following diagram:



The derivation of such a diagram for a sentence from a given set of syntactic rules is called "parsing." The diagram itself is called a "parse tree," and its labeled parts are called the "phrase nodes" of the parse tree; for example, the `PREDICATE` phrase node in the tree above encompasses the actual sentence fragment "is red." Given our syntax rules above, PyElly can automatically parse our sentence; and the resulting tree diagram can then be the starting point for interpreting the sentence.

Our simple grammar above so far can describe only a single sentence, but we can extend its coverage by adding rules for other kinds of structures and vocabulary. For example, the added syntax rules

```
SUBJECT->DETERMINER NOUN
DETERMINER->an
NOUN->apple
```

will let PyElly analyze the sentence "An apple is red." We can continue to build up our grammar here by adding other rules; for example,

```
PREDICATE->VERB
VERB->falls
```

will also put "It falls" and "An apple falls" into the sentences PyElly can recognize. We can continue in this way to encompass still more types of grammatical structure and still more vocabulary. The new rules can be added in any order you want.

The key idea here is that a few rules can be combined in various ways to describe many different sentences. There is still the problem of choosing the proper mix of rules to describe a language in the most natural and efficient way, but we do fairly well by simply adding one or two rules at a time as done above. In some complex applications, we may eventually need hundreds of such rules, but these can still be worked out in small steps.

Technically speaking, PyElly grammar rules shown above define a "context-free language." Such a grammar mainly describes the relationships between adjacent subtrees for a sentence; and it is harder to correlate the possibilities for the structures far apart in a parse tree even though they might be close together as words in an actual sentence. For example, consider the example of a context-free rule with the parallel structures `SUBJECT` and `PREDICATE`.

```
SENTENCE->SUBJECT PREDICATE
```

In languages like English, subjects and predicates have to agree with each other according to the attributes of person and number: "We fall" versus "He falls". When grammatically acceptable `SUBJECT` and `PREDICATE` structures have to be formed in correlated ways, context-free rules here force us to work harder to restrict a `SENTENCE` to have only certain combinations of subjects and predicates in agreement. We might have to write multiple explicitly correlated rules like

```
SENTENCE->SUBJECT1 PREDICATE1
SENTENCE->SUBJECT2 PREDICATE2
SENTENCE->SUBJECT3 PREDICATE3
```

where `SUBJECT`*i* would always agree properly with `PREDICATE`*i*, but this has the disadvantage of greatly multiplying the number of rules we have to define. A good natural language toolkit should make our job easier than this.

Even though English and other natural languages are not context-free in a theoretical sense, we still want to treat them that way for practical reasons. By doing so, we can take advantage of the many sophisticated techniques developed for parsing artificial computer languages like C++ or Swift, which do tend to be context-free. This is the approach taken in PyElly and its predecessors.

For convenience, PyElly also incorporates semantic checking of intermediate results in parsing and allows some shortcuts to make grammars more concise (see Section 8). These extensions can go on top of a plain context-free parser to give it some context-sensitive capabilities, although some kinds of sentences still cannot be handled by PyElly. (The classic problematic context-sensitive examples are the parallel subjects and predicates, in sentences like "He and she got cologne and perfume, respectively.")

The syntax of natural language can get quite complicated in general; but we usually can break this down in terms of simpler structures. The challenge of defining a PyElly grammar is to capture enough of such simpler structures in grammar rules to support a proper analysis of any input sentence that we are likely to see.

A PyElly sentence analysis can actually go far beyond the examples above. For example, the leaves of a parse tree may be more than single words. PyElly vocabulary rules can combine multiple input words into a single parse tree leaf token or split off parts of an input word into separate tokens. PyElly can also control when particular PyElly syntax rules apply by subdividing syntactic categories. Furthermore, PyElly has to handle the unavoidable ambiguity in analyses when working with a natural language. This user manual will go into detail on these and other matters in later subsections, but for simple applications, the basic framework described in this section should be enough.

You must be able to understand most of the discussion in this section in order to proceed further with PyElly. A good text for those interested in learning more about language and formal grammars is John Lyon's book Introduction to Theoretical Linguistics (Cambridge University Press, 1968). This is written for college-level readers, but covers to the basics that you will need to know.

3. The Semantics of a Language

The notion of meaning has always been difficult to talk about. It can be complicated even for individual sentences in a language, because meaning involves not only their grammatical structure, but also where and when it is used and who is using it. A simple expression like "Thank you" can take on different significance, depending on whether the speaker is a thug collecting extortion money, the senior correspondent at a White House press conference, or a disaster victim after an arduous rescue.

Practical computer natural language applications cannot deal with all the potential meanings of sentences, since this would require modeling almost everything in a person's view of world and self. A more realistic approach is to ask what meanings will actually be appropriate for a computer to understand in a particular application. If the role of a system in a user organization is to provide, say, only information about employee benefits from a policy manual, then it probably has no reason to figure out references to subjects like sex, golf, or the current weather.

Within PyElly, the scope of semantics will be limited even more drastically: we will deal with the meaning of sentences only to the extent of being able to translate them into some other language and to evaluate our options when we have more than one possible translation. This has the advantage of making semantics less mysterious while allowing us still to implement useful kinds of language processing.

For example, the meaning of the English sentence "I love you" could be expressed in French as "Je t'aime." Or we might translate the English "How much does John earn?" into a data base query language "SELECT SALARY FROM PAYROLL WHERE EMPLOYEE=JOHN." Or we could convert the statement "I feel hot" into an IoT command line like

```
set thermostat /relative /fahrenheit:-5
```

In a sense, we have cheated here, avoiding the problem of meaning in one language by passing it off to another language. Such a translation, however, can be quite useful if there is a processor that understands the second language, but not the first. This is definitely a modest approach to semantics; but it beats talking endlessly about the philosophical meaning of meaning without ever accomplishing anything in actual code.

As noted before, the large number of possible sentences in a natural language prevents us from compiling a table to map every input into its corresponding output. Instead, we must break the problem down by first taking the semantics of the various constituent structures defined by a grammar and then combine their individual interpretations to derive the overall meaning of a given sentence.

With PyElly, we define the semantics of a sentence structure as procedures associated with each of the grammatical rules describing the structure. There will actually be two different kinds of semantic procedures here: those for writing out translations will be called "generative," while those for evaluating alternative translations will be called

"cognitive." At this stage, however, we shall focus on the generative, and leave the cognitive until Section 10, since these two aspects of semantics operate quite differently.

A successful PyElly sentence analysis will produce a parse tree describing its syntactic structure. Each phrase node of that tree will be due to a particular grammatical rule, and associated with that rule will be a generative semantic procedure defining its meaning. You will have to supply such a procedure for each of your grammar rules, though often you can just take the defaults defined by PyElly for its three types of grammar rules.

The top phrase node of a complete PyElly parse tree will always be that of the type `SENT` (="sentence"). The generative procedure for such a phrase node will always be called by PyElly to begin the translation of the original input sentence. This should then set off a cascade of other procedure calls through the various lower-level constituent structures of the sentence to produce a final output. The actual ordering of calls to subconstituent procedures will be determined by the logic of the procedures at each level of the tree.

A PyElly generative semantic procedure basically will operate on text characters in a series of output buffers. This will involve standard text editing operations commonly supported in word processing programs: inserting and deleting, buffer management, searching, substitution, and transfers between buffers. Consistent with PyElly semantics being procedures, there will also be local and global variables, structured programming control structures, subprocedures, simple lookup, and set manipulation.

Communication between different semantic procedures will be through local and global variables as well as by inspecting the current content of buffers. The value of any local or global variable will always be a string of arbitrary Unicode characters, possibly the null string. Global variables will be accessible to all procedures and will remain defined even across the processing of successive sentences, serving as a longer-term memory for PyElly translations.

Local variables will have a limited scope such as seen in programming languages like C or PASCAL. They are defined in the procedure where they are declared and also in those procedures called as subroutines either directly or indirectly. When there are multiple active declarations of a variable with a given name visible to a semantic procedure, the most recent one applies. Upon exit from a procedure, all of its local variables immediately become undefined.

Let us look at some actual semantic procedures to see how a PyElly translation would work. Suppose that we have the five grammar rules

```
SENT->SUBJECT PREDICATE
SUBJECT->PRONOUN
PRONOUN->we
PREDICATE->VERB
VERB->know
```

With these rules, we can implement a simple translator from English into French with the five semantic procedures below, defined respectively for each rule above. For the time being, the commands in the procedures will be expressed in ordinary English. These commands will control the entry of text into some output area, such as a text field in a window of a computer display. Here is our set of procedures:

> For a `SENT` consisting of a `SUBJECT` and `PREDICATE`: first run the procedure for the `SUBJECT`, insert a space into the output being generated, and then run the procedure for the `PREDICATE`.
>
> For a `SUBJECT` consisting of a `PRONOUN`: just run the procedure for the `PRONOUN`.
>
> For the `PRONOUN we`, insert `nous` into the output being generated.
>
> For a `PREDICATE` consisting of a `VERB`, just run the procedure for the `VERB`.
>
> For the `VERB know`, insert `connaisser`.

With this particular set of semantic procedures, the English sentence "we know" will be translated to `nous` followed by a space followed by `connaisser`. You can easily verify this by starting with the semantic procedure for `SENTENCE` and tracing through the cascade of procedure executions.

Each syntactic rule in a grammar must have a semantic procedure, even though the procedure might be quite trivial such as above when a `SUBJECT` is just a `PRONOUN` or a `PREDICATE` is just a `VERB`. This is because we need to make a connection at each level from `SENTENCE` all the way down to individual input words like `we` and `know`. These connections will give us a framework to extend our translation capabilities just by adding more syntactic rules plus their semantic procedures; for example,

```
PRONOUN->they
```

> For the `PRONOUN they`: insert `ils`.

You may have noticed, however, our example translation here is incorrect. Even more so than English, French verbs must agree in person and number with their subject, and so the translation of `know` with the `SUBJECT we` should be `connaissons` (first person plural) instead of `connaisser` (the infinitive). Yet we cannot simply change the `VERB` semantic procedure above to "insert `connaissons`" because this would be wrong if the `PRONOUN` for `SUBJECT` becomes `they` (third person plural).

We really need more elaborate semantic procedures here to get correct agreement. This is where various other PyElly generative semantic commands come in. In particular, we can use local variables to pass information about number and person between the semantic procedures for our syntactic structures to govern their translations. Nevertheless, the overall PyElly framework of semantic procedures attached to each syntactic rule and called recursively will remain the same.

(If you want to look ahead to see how a simple English-to-French translation might be done properly in PyElly, you can peek at Subsection 6.2. Understanding the details of the syntactic rules and their generative semantics will require you to go through Sections 4 and 5 first, however.)

Semantic procedures must always be coded carefully for proper interaction and handling of details in all contexts. We have to anticipate all the ways that constituent structures can come together in a sentence and provide for all the necessary communication between them at the right time. We can make the problem easier here by taking care to have lower-level structures be parts of only a few higher-level structures, but this will require some advance planning.

Writing syntactic rules and their semantic procedures is a kind of programming and will require some programming skills. It will be harder than you first might think when you try to deal with the complexity of natural languages like English or French. PyElly, however, is designed to help you to do this kind of programming in a highly structured way, and it should be easier than trying to accomplish the same kind of translation in a language like Python or even LISP.

The idea of procedural semantics was introduced by Vaughan Pratt in his LINGOL system. Procedures are not the only way of dealing with meaning, but they fit in well with what we do in computational linguistics and especially within a framework where we want to rewrite input text from one language to another. The availability of various programming language features like variables in PyElly semantic procedures also provides a way of modeling the contexts of sentence parts beyond what we can describe with syntactic rules alone.

4. Defining Tables of PyElly Rules

By now, you should understand the idea of grammar rules and their semantic procedures. This section will go into the mechanics of how to define them in text files to be read in by PyElly at startup. To implement different applications such as translating English to French or rewriting natural language questions as structured data base queries, you just need to provide the appropriate files of rules for PyElly to load.

PyElly rules fall into eight main types: (1) grammar, (2) vocabulary, (3) macro substitutions, (4) patterns for determining syntactic types, (5) personal names and name components, (6) various other regular forms of naming, (7) morphology, and (8) punctuation and sentence delineation. The grammar of a language for an application reflects the linguistic capabilities supported by a target system, while a vocabulary is geared toward a particular context of use; macros support particular users of a system, and special patterns and name components tend to be specific to given applications. Having separate tables of rules make it easier to tailor PyElly operation for different environments while allowing parts of a language definition to be reused.

This section will focus on the grammar, special pattern, and macro rule tables, required by most PyElly applications. Some of the finer details of syntactic rules will be postponed to Section 8, "Advanced Programming: Grammar." The creation and use of tables for vocabulary, names, and morphology will be described in Section 9, "Advanced Programming: Vocabulary."

To make PyElly do something, you have to set up an application defined by a specific set of language definition rules organized into tables. The current PyElly package implements each type of rule table as a Python class with an initialization procedure that reads in its rules from an external text source file. The text input files for rules governing a particular application `A` should be named as follows:

`A.g.elly` for syntactic rules and their semantic procedures in a grammar.

`A.m.elly` for macro substitutions.

`A.p.elly` for special patterns.

You may replace the `A` here with whatever name you choose for your application, subject to the file-naming conventions for your computer platform. Only the `A.g.elly` file is mandatory for any PyElly application; the other two may be omitted if you have no use for either macro substitutions or patterns. Section 7 will explain how PyElly will look for various language definition files for an application and read them in.

The rest of this section will describe the required formats of the definitions in the input files `A.g.elly`, `A.m.elly`, and `A.p.elly`. Normally you would create these files with a text editor or a word processor. The NotePad accessory on a Windows PC or TextEdit on a Mac will be quite adequate, although you may have to rename your files afterward because they insist on writing out files only with extensions like `.txt`.

An important element of most language rules will be syntactic structure names, seen in Section 2. We also call them "syntactic types" or "parts of speech," but they will be more general than what we learned in grade school. The current implementation of PyElly in Python can handle up to 112 different syntactic types in its input files. Six of these types, however will be predefined by PyElly with special meanings.

`SENT`	Short for `SENT`ence. Every grammar must have at least one rule of the form `SENT->X` or `SENT->X Y`. PyElly translation will always start by executing the semantic procedure for a `SENT` structure.
`END`	For internal purposes only. Avoid using it.
`UNKN`	Short for `UNKN`own. This structure type is automatically assigned to strings not known to PyElly through its various lookup options. (See Subsection 9.1 for more on this.)
`...`	For an arbitrary sequence of words in a sentence. This is for applications where much of the text input to process is unimportant. (See Section 8 for more details.)
`PUNC`	For punctuation. (See Section 11.)
`SEPR`	For inferred sentence separators. (See Subsection 12.6.)

You will of course have to make up your own names for any other syntactic types needed for a PyElly application. Names may be arbitrarily long in their number of characters but may include only ASCII letters, digits, and periods (`.`); upper and lower case will be the same. You do not have to use traditional grammatical names like `NOUN`, but why be unnecessarily obscure here?

You may want to keep your syntactic type names unique in their first four characters, however. This is because PyElly will truncate names to that many characters in its formatted diagnostic output like parse trees (see Section 12, "PyElly Parsing"). The resulting tree will be confusing if you have syntactic types like `NOUN` and `NOUNPHRASE`. When reading in grammar rules, PyElly will warn you about any such situation.

Here are some trivial, but functional, examples of grammar, macro, and pattern definition files:

```
# PyElly Definition File
# example.g.elly
g:sent->ss        # top-level rule

__
g:ss->unkn

__
g:ss->ss unkn

__
```

```
# PyElly Definition File
# example.m.elly
i'm->i am          # contraction
```

```
# PyElly Definition File
# example.p.elly
0 &# number -1     # simple integer
```

These rules will be explained in separate subsections below. Lines beginning with '`# `', disregarding any leading space characters, will be taken as comments and ignored. A comment can also be at the end of an input line, but this has to start with '` # `'. The extra spaces are necessary because the `#` character is special in PyElly pattern matching.

4.1 Grammar (`A.g.elly`)

An `A.g.elly` text file may have four different types of definitions: (1) syntactic rules with their associated semantic procedures, (2) individual words with their associated semantic procedures, (3) general semantic subprocedures callable from elsewhere, and (4) initializations of global variables at startup. These definitions will be respectively identified in the `A.g.elly` file by special markers at the start of a line: `G:`, `D:`, `P:`, or `I:`. The definitions may appear in any order; markers can be upper or lower case.

4.1.1 Syntactic Rules

These must be entered as text in a strict format of lines. Syntactic rule definitions will follow the general outline as shown here in a monospaced font for easier reading:

```
G:X->Y                  # a marker G: + a syntax rule
_                       # a single <UNDERSCORE>,
                        #     omitted if no semantic
                        #     procedure follows
     .                  #
     .                  # the body of a generative
     .                  #     semantic procedure
                        #
__                      # a double <UNDERSCORE>,
                        #     mandatory definition terminator
```

a. In a `G:` line, PyElly will allow spaces anywhere except after the `G`, within a syntactic structure name, or between the '–' and '>' of a rule.

b. The same formatting applies for a PyElly syntactic rule of the form `X->Y Z`.

c. A generative semantic procedure for a syntactic rule will always appear between the line starting with a single underscore (_) and the line starting with a double underscore (__).

d. A cognitive semantic procedure may appear before the single underscore, but this will be described later in Section 10 ("Logic for PyElly Cognitive Semantics").
e. The actual basic actions for generative semantics will be covered in Section 5 ("Operations for PyElly Generative Semantics").
f. If a semantic procedure is omitted, various defaults apply; see Section 6 ("PyElly Programming Examples"). In this case, the first single underscore delimiter is optional.

4.1.2 Grammar-Defined Words

It is usually a good idea to keep the vocabulary for a PyElly application separate from a grammar as much as possible. For scalability, PyElly will keep its vocabulary mainly in an external data store; and Section 9 will describe how to set this up. Some single-word definitions, however, may also appear alongside the syntactic rules in a grammar definition file. These will be called internal dictionary rules.

In particular, some words like THE, AND, and NOTWITHSTANDING are associated with a language in general instead of any specific content. These are probably best defined as part of a grammar file anyway. In other cases, there may also be so few words in a defined vocabulary for an application that we may as well include them all internally in a grammar rather than externally.

The form of an internal dictionary rule is similar to that for a grammatical rule:

```
D:w<-X                  # a marker D: + a word "w"
                        #    + a structure type X
_                       # a single <UNDERSCORE>
     .                  #
     .                  # a generative semantic procedure
     .                  #
__                      # a double <UNDERSCORE>,
                        #     mandatory definition terminator
```

a. The `D:` is mandatory in order to distinguish a word rule from a grammatical rule of the form `X->Y`.
b. To suggest the familiar form of printed dictionaries, the word `w` being defined appears first, followed by its structure type `X` (i.e., part of speech). Note that the direction of the arrow `<-` is reversed from that of syntax rules.
c. The `w` must be a single word, number, or other text token, possibly hyphenated, or a single punctuation mark, or a short bracketed text segment like `(s)`. Multi-word terms in an application must be defined in PyElly's external vocabulary or stitched together by grammar rules, macro substitution rules, or other mechanisms discussed in Section 9.
d. The single and double underscore separators for semantics are the same as for syntactic rules. A word definition may also have both cognitive and generative semantics to establish its meaning for PyElly.
e. A given word may have multiple internal dictionary rules and so be ambiguous.
f. A given internal dictionary word may also be defined elsewhere in PyElly (see Section 9). These other definitions may differ. PyElly will figure out how to make sense of them.

4.1.3 Generative Semantic Subprocedures

Every PyElly generative semantic procedure for a rule will be written in a special PyElly programming language for text manipulation. This language also allows for named subprocedures, which need not be attached to a specific syntax rule or internal dictionary rule. Such subprocedures may be called in a generative semantic procedure for a PyElly rule or by another subprocedure. Their definitions should in a `*.g.elly` grammar file, but may appear anywhere in that file.

A subprocedure will take no arguments and return no values. All communication between semantic procedures must be through global or local variables or from the text written into PyElly output buffers (see Section 5 for details). Calls to subprocedures may be recursive, but you will be responsible for avoiding infinite recursion here.

A subprocedure definition will have the following form:

```
P:nm                    # a marker + procedure name "nm"
_                       # a single <UNDERSCORE>,
                        #     mandatory
     .                  #
     .                  # generative semantic procedure body
     .                  #
__                      # double <UNDERSCORE> delimiter,
                        #     mandatory
```

a. Note the absence of any arrow, either `->` or `<-`, in the first definition line.
b. A procedure name `n` should be a unique string of alphanumeric characters without any spaces. It should be of non-zero length, because PyElly itself predefines the procedure with no name. The case of letters is unimportant. Duplicate definitions for the same procedure name will be reported as an error.
c. The underscore separators are the same as for syntactic rules and word definitions and will both be mandatory for a subprocedure definition.
d. A subprocedure definition may have only generative semantics. Cognitive semantics will never apply to a subprocedure and will always be ignored if specified.

4.1.4 Global Variable Initializations

PyElly global variables in a generative semantic procedure can be set in various ways (see Subsection 5.6). You may, however, want to preset some important parameters in the loading of a particular application grammar before any generative semantic procedure is run. To initialize a global variable `x` to the string `s` at startup, just put in a `I:` line in your grammar definition file like this:

```
I:x=s
```

You must have one `I:` line for each global variable being initialized. Note that an `I:` line always stands by itself; there is no associated generative semantic procedure as in the case of `G:`, `D:`, and `P:` lines. An `I:` line may appear anywhere in a grammar definition file, but for clarity, it should be before any reference to it in a semantic procedure. For readability, you may freely put spaces around the variable name `x` and after the = sign here. For example,

```
I:iterate   = abcdefghijklm
I:joiner    = svnm
```

In the first initialization above, the `iterate` global variable is set to `abcdefghijklm`. A string value may have embedded space characters, but all leading and trailing spaces will be ignored and multiple consecutive embedded spaces will be collapsed into one.

4.2 Special Patterns For Text Elements (`A.p.elly`)

Many elements of text are too numerous to list out in a dictionary, but are recognizable by their form; for example, Social Security numbers, web addresses, or Icelandic surnames. PyElly allows you to identify such elements in input text by specifying the patterns that they conform to. That is how PyElly now deals with ordinary decimal numbers in input text to be translated.

PyElly special patterns serve to assign a syntactic structure type (part of speech) to a single word or other simple token in its input text. This will supplement any explicit definition in a grammar's internal dictionary (see Subsection 4.1.2) or in its external vocabulary table (see Subsection 9.4). For example, you can make '123' a `NOUN` by a `D:` rule in a grammar table, but PyElly can still infer that it is a type `NUM` from a pattern.

In general, we may have to compare a character string with multiple patterns to identify a particular kind of text element. PyElly coordinates this kind of processing through a finite-state automaton (FSA), which should be familiar to every aspiring computational linguist. This is not a physical machine, but a software algorithm working from a predefined set of rules telling it how to proceed step by step in matching up a series of patterns from left to right against a stream of input text.

The key concept in an FSA is that of a state, which sums up what patterns have already been matched by a text string and what patterns it should look for next. An FSA will typically have multiple states, but one of these will always be the starting state when the FSA is at the front of an input text string with nothing yet matched. The total number of different states must be limited, hence the finiteness of an automaton.

In any given state, an FSA will have a specific list of patterns to check against the input text at its current position. These may have wildcards able to match multiple characters in text; for example, any digit `0-9`. PyElly wildcards are similar to the wildcards used in regular expressions in utilities like `grep`, but are defined specifically for natural language processing (see below). This differs from the usual kind of FSA, which will allow no wildcards in its patterns to trigger transitions.

Each pattern at a state will take a certain action upon any match. Usually, this means moving forward in its input string and going on to a next state according to predefined rules. There may be more than one next state because a string at a given FSA state can match more than one pattern when they have wildcards. This is a complication, but everything is still equivalent to a regular FSA. It just lets our rules be more compact.

Some matches will have no next state in a PyElly FSA, but instead will specify a syntactic structure type. If an FSA has also reached a suitable endpoint for an input token, then a match is complete, and PyElly can create a token of that given syntactic type from the text being matched. At this point, a normal FSA would be done, but PyElly will also examine all the matching possibilities arising from multiple next states for an FSA. PyElly continues until it has checked all reachable states. PyElly then will return a positive match length if any matching token has been found; 0, otherwise. Only the longest of any full match will be kept. See Section 12 for more details.

PyElly requires every FSA state to be identified by a unique non-negative integer, where the initial state is always `0`. The absence of a next state after a match is indicated by `-1` or `-2`. At each state, what to look for next is defined as a pattern of literal characters and wildcards. A `*.p.elly` definition file will consist of separate lines each specifying a possible pattern for a given state, an optional PyElly syntactic structure type associated with any full match, and a next state upon a match with the pattern at a state. These specifications comprise the table of pattern rules that a PyElly FSA will work with.

Here is a simple PyElly file for an FSA with patterns and rules:

```
# simple FSA to recognize syntactic structure types
# example.p.elly
#
# each input record is a 4-tuple
# STATE PATTERN SYNTAX NEXT

0 #,   - 1
0 ##,  - 1
0 ###, - 1
1 ###, - 1
1 ###$ NUM -1
0 &#   -    2
2 .    -    3
2 $    NUM -1
3 &#$  NUM -1
3 $    NUM -1
```

This recognizes numbers like `1024`, `3.1416`, and `1,001,053` as tokens of syntactic type `NUM`. A pattern rule in its most basic form will be a single line with four parts:

```
STATE    PATTERN    SYNTACTIC TYPE    NEXT
```

a. The first part is an integer ≥ 0 indicating a current PyElly finite automaton state.
b. The second part is a `PATTERN`, which may be an arbitrary sequences of letters, numbers, and certain punctuation, including hyphen (`-`), comma (`,`), period (`.`), slash (`/`). If these are present, they must be matched exactly within a word being checked.
c. A `PATTERN` may have explicit characters and also wildcards, which can match various substrings in a token string. PyElly wildcards in a pattern will be as follows:

#	will match a single digit 0 - 9, including some exponents
@	will match a single letter a - z or A - Z, possibly with diacritics
!	will match a single uppercase letter, possibly with diacritics
¡	will match a single lowercase letter, possibly with diacritics
?	will match a single digit or letter
*****	will match an arbitrary sequence of non-blank characters, including a null sequence
&?	will match one or more letters or digits in a sequence
&#	will match one or more digits in a sequence
&@	will match one or more letters in a sequence
^	will match a single vowel
%	will match a single consonant
'	will match an apostrophe appearing either as `'` (ASCII) or `’` (Unicode right single quotation mark) or `′` (Unicode prime)
$	will match the end of a token, but not add to the extent of any matching; a token will end if followed by a space or certain punctuation or non-text character like 中

d. If a character to be matched is also a PyElly wildcard, then you must escape it in a `*.p.elly` pattern with a double backslash; for example, `\\*` to match `*` literally.
e. A `PATTERN` consisting only of `\0` (ASCII NUL) will cause an automaton to move to some **non-final** state without any matching. A double backslash is incorrect here.
f. Brackets `[` and `]` in a pattern will enclose an optional subsequence to match; only one level of bracketing is allowed and only alphanumeric characters are allowed inside the brackets. The pattern `[ab]c` will match the string `abc` or the string `c`.
g. A pattern with a wildcard other than `\0` or `$` by itself must always match at least one character; for example, the pattern `[a]*` will be rejected.
h. The third part of a pattern rule is a syntactic type (part of speech) like `NOUN`, indicating a possible final state. Any non-final state must specify '–' as its associated type.
i. The fourth part is the next state to go to upon matching a specified pattern. This will be an integer ≥ -2. A `-1` here means a final state, which should follow a syntactic type; a `-2` will also be final, but allows for matched token ending in the middle of a normal token.
j. All `-1` final state patterns not ending with the `*` or `$` wildcards will have a wildcard `$` appended automatically. Patterns for all non-final and `-2` final states will be left alone.

For example, the pattern `###-##-####$` matches Social Security numbers, while the pattern `(###)###-####$` matches a telephone number with an area code, but without separating spaces. See Subsection 9.2.1 for more on possible number patterns.

PyElly also treats lowercase alphabetic characters as semi wild in effect. A lowercase letter in a pattern will match the same letter in text input irrespective of case. An uppercase letter in a pattern, however, will match only the uppercase version of the letter in text input. So, put an uppercase letter into a pattern only if you mean it.

The PyElly FSA will always recognize only a single token with no spaces. You cannot match multiple words separated by spaces like XXX YYY ZZZ. You can, however, recognize special tokens with punctuation characters not normally in a PyElly token; for example, (R-TEXAS). To do this in the PyElly FSA, you should have separate states to match a single punctuation character:

```
0 (      -  1
1 !-!&@  -  2
2 )    XID -1
```

It is possible to define an `A.p.elly` file of rules that reference only the start state `0`. When your patterns have many wildcards, however, having multiple states can make your steps of matching much clearer. Having more states will always work and will also let you share states in different subsets of pattern rules.

4.3 Macro Substitutions (`A.m.elly`)

Macro substitution is a way of automatically replacing specific substrings in an input stream by other substrings. This is a useful capability in a language translator, and so PyElly adds it as yet another integrated tool, adapting code from Kernighan and Plauger, *Software Tools*, Addison-Wesley, 1976.

The main difference in PyElly macro substitution versus *Software Tools* is that substrings to be replaced can be described with wildcards along with explicit characters to match and that the substrings to replace parts of the original string can specify the parts of the original string that matched consecutive sequences of pattern wildcards. This capability was inspired by how the SNOBOL programming language worked.

Macro substitution is convenient for handling idioms, synonyms, abbreviations, and alternative spellings and for doing syntactic transformations awkwardly described by context-free grammar rules. A PyElly macro substitution rule is generally defined as follows in language definition file `A.m.elly`:

```
P_Q_R->A B C D
```

a. Each macro substitution rule must be a single line. Since macros will defined in their own `*.m.elly` file, we need no marker at the beginning of each line.
b. The left side of a substitution will be a pattern containing literal Unicode characters and PyElly wildcards. It may not be empty. The wildcards will include all the ones listed in the preceding subsection (3.2) for special patterns, plus a _ space wildcard and a ~ nonalphanumeric wildcard. The left side may have arbitrarily many parts separated by the _ wildcard. The space wildcard is the only one allowed within pattern brackets `[ ]` for optional components in a match; there will also be semi wild matching of lower alphabetic characters in patterns as in the preceding subsection. A left side not ending in a _ or `*` wildcard is assumed to end with `$`. Note that a ~ wildcard will not match an ampersand, which is interpreted as a stylized writing of the Latin `et`.
c. In macros, a pattern can match multiple words in a macro substitution when there are space wildcards on the left side. This makes the `$` wildcard much less useful than in FSA pattern matching, which are restricted to a single token, which may include punctuation.
d. The right side of a substitution may contain space characters as word separators; or it may be empty. Upper and lower case will be significant on the right side. It will be significant on the left side only for semi wild matching. The right side can be arbitrarily long, although you usually want a substitution to be shorter than an original string.
e. Any and all input text matching the pattern on the left side will be replaced by the right side. This is the actual macro substitution defined by a rule.
f. A `\\1`, `\\2`, `\\3`, and so forth, on the right stands respectively for the first, second, third, and so forth, parts of text matched by wildcard patterns on the left. PyElly allows up to nine bindings for a pattern. Each binding applies to a sequence of contiguous wildcards, except for _ and `'`, which will always be bound to a single character. For example, matching the pattern `#@abc#` will associate the first digit and letter of a match with `\\1` and the last digit with `\\2`. Matching the pattern `#a'_*` with a string will associate the digit before `a` with `\\1` , a space with `\\2`, and any string after the space with `\\3`.
g. Characters used as wildcards can be matched literally by escaping them with `\\`; for example `\\?` matches a question mark. This applies only to the left side of macro rules.
h. When any macro is matched and its substitution done, then all macros will be checked again against the modified result for further possible substitutions. When a macro eliminates its match entirely, though, substitutions will be ended at that position.
i. The order of macro substitution rules is significant. Those first in a definition file will always be tried first, possibly affecting the applicability of those defined afterward. Macro patterns starting with a wildcard will always be checked after all others, however.
j. Macros have no associated semantic procedures because they run outside of PyElly syntax-driven parsing and rewriting.

Macro substitutions will be trickier to manage than grammatical rules because you can accidentally define them to work at cross-purposes. An infinite loop of substitutions can result if you are careless. Nevertheless, macros can greatly simplify a language definition when you use them properly and keep their patterns fairly short.

Substitution rules will be applied to the current PyElly input text buffer at the current position each time before the extraction of the next token to be processed. This can override any tokenization rules in effect and can modify any previous stemming of

words. All of that can add up to substantial overhead if you define many macros because all possible substitutions will always be tried out at each possible token position.

Here are some actual PyElly macro substitution rules from an application trying to compress input text as much as possible while keeping it readable:

```
*'ll -> \\1
percent* -> %
will_not -> wont
data_base -> DB
greater than or equal to -> >=
carry_-ing -> wth
receiv[e]* -> rcv\\1
#*_@[.]m$ -> \\1\\3m
```

The last macro rule above will replace "10 p.m" with "10pm" to save space.

The substitution part of a macro rule may include a `\\s` special character, indicating the `u'\u001E'` Unicode character, which is ASCII `RS` (record separator). For example,

```
,_@@&@ing_that -> ,\\s \\1ing that
```

will insert `RS` into the PyElly text input buffer after finding a comma (`,`) when followed by a participial -ING THAT expression. PyElly automatically recognizes `RS` as the syntactic category `SEPR`, but your grammar rules then must do something with this.

You often can use macro substitutions to handle idioms or other irregular forms that are exceptions to general language rules or to make explicit some dependencies between words that might be hard to capture in PyElly context-free syntax rules. In such cases, just rewrite problematic text into a less unambiguous form that your grammar rules will recognize. This could be an arbitrary string of letters or digits not normally readable by a person; for example, `didn't -> DDNT`.

Macros will always be applied after any inflectional stemming, but also before any morphological stemming (see Subsection 9.1) with an unknown text element. Use them with care; PyElly will warn you when something might be dangerous, but will not stop you from getting into an infinite loop of substitutions.

5. Operations for PyElly Generative Semantics

In Section 3, we saw examples of generative semantic procedures expressed in English. PyElly requires, however, that they be written in a special structured programming language for editing text in a series of output buffers. This language has conditional and iterative control structures, but generally operates at the nitty-gritty level of manipulating a few characters at a time.

Basically, PyElly generative semantics manages buffers and moves around text in them. The semantic procedures for various parts of a PyElly sentence all have to put their contributions for a translation into the right place at the right time. Proper coordination is critical; you have to plan everything out and control the interactions of all procedures.

Every generative semantic procedure will be a sequence of simple commands, each consisting of an operation name possibly followed by arguments separated by blanks. These various operations are described below in separate subsections. For clarity, the operation names are always shown in uppercase here, but lower or mixed case will be fine. Comments below begin with ' # ' and are not part of a command.

5.1 Insertion of Strings

These operations put a literal string at the end of the current PyElly output buffer:

```
APPEND any string        # put "any string" into current buffer

BLANK                    # put a space character into buffer

SPACE                    # same as BLANK

LINEFEED                 # start new line in buffer, add space

UNICODE xxxx             # convert hexadecimal xxxx into char
                         # and put into buffer

OBTAIN                   # copy in the text for the first token
                         # at the sentence position of the
                         # phrase constituent being translated
```

5.2 Subroutine Linkage

For calling procedures of subconstituents for a phrase and returning from such calls:

```
LEFT                     # calls the semantic procedure
                         # for subconstituent structure
                         # Y when a rule is of the form
                         # X->Y or X->Y Z.
```

```
RIGHT                   # calls the semantic procedure
                        # for subconstituent structure
                        # Z for a rule of the form
                        # X->Y Z, but Y for rule X->Y
RETURN                  # returns to caller, not needed at
                        # the end of a procedure

FAIL                    # rejects the current parsing of an
                        # input sentence and returns to the
                        # first place where there is a choice
                        # of different parsings for a part of
                        # an input structure
```

5.3 Buffer Management

Processing starts with a single output text buffer. Spawning other buffers can help to keep the output of different semantic procedures separate for adjustments before joining everything together. You can set aside the current buffer, start working in a new buffer and then return to the old buffer to shift text back and forth between them.

```
SPLIT                   # creates a new buffer and
                        # directs processing to it

BACK                    # redirects processing to end
                        # of previous buffer while
                        # preserving the new buffer

MERGE                   # appends content of a new
                        # buffer to the previous one,
                        # deallocating the new one
```

These allow a semantic procedure to be executed for its side effects without immediately putting anything into the current output buffer. Splitting and merging will work when nested recursively, but for clarity, put a corresponding `SPLIT` and `MERGE` in the same procedure. The `MERGE` operation can also be combined with string substitution:

```
MERGE /string1/string2/  # as above, except that all occurrences
                         # of "string1" in the new buffer will be
                         # be changed to "string2"
                         # (the divider / here may be replaced by
                         #  any char not in string1 or string2)
```

5.4 Local Variable Operations

Local variables can store a Unicode string value. Variable names may have one or more letters or digits. They will be declared within the scope of a semantic procedure and will automatically disappear upon a return from the procedure.

```
VARIABLE xv=string        # declare and set variable xv to
                          # "string"; if the "=string" part
                          # is missing, then initialization
                          # is to a null string ""

SET xv=string             # assigns "string" to the most recent
                          # declaration of local variable xv,
                          # defining xv if necessary; if the
                          # "=string" part is missing, then
                          # assignment is to a null string ""
```

A string may contain any printing characters, but trailing spaces will be dropped. To handle single space characters specified by their ASCII names, you may use the following special forms:

```
VARIABLE xv SP            # define variable xv as single
                          # space char

SET xv SP                 # set variable xv as single
                          # space char, defining xv
                          # if necessary
```

Note the absence of the equal sign (=) here. PyElly will recognize `SP`, `HT`, `LF`, and `CR` as space characters here and nothing else. This form can also be used with the `IF`, `ELIF`, `WHILE`, and `BREAKIF` semantic operations described below. You may write `VAR` as shorthand for `VARIABLE`; they are equivalent.

Some operations have a local variable as their second argument. These support assignment, concatenation of strings, and queuing.

```
ASSIGN  xv=zv             # assigns the strings value of local
                          # variable zv to the local
                          # variable xv in their most
                          # recent declarations

QUEUE   qv=xv             # appends the string stored for local
                          # variable xv to any string stored for
                          # variable xv

UNQUEUE xv=qv n           # removes the first n chars of the
                          # string stored in local variable
                          # qv and assigns them to local
                          # variable xv; if n is unspecified,
                          # the character count defaults to 1;
                          # if qv has fewer than n chars, then
                          # xv is just set to the value of qv
                          # and qv is set to the null string
```

The equal sign (=) and righthand local variable name is required for `UNQUEUE` and `QUEUE`. If a lefthand local variable is undefined here, it will become automatically defined in the scope of the current generative semantic procedure.

5.5 Set-Theoretic Operations with Local Variables

PyElly allows for manipulation of sets of strings, represented as their concatenation into a single string with commas between individual strings. For example, the set {"1","237"," ab","u000"} would be represented as the single string "1,237,ab,u000". When local variables have been set to such list values, you can apply PyElly set-theoretic operations to them.

```
UNITE x<<z              # takes the union of the list values
                        # of local variables x and z
                        # and saves the result in x

INTERSECT x<<z          # intersects the list values
                        # of local variables x and z
                        # and saves the result in x

COMPLEMENT x<<z         # restricts the list values of
                        # of local variable x to those
                        # not in the list value for
                        # local variable z and saves
                        # the result in x
```

5.6 Global Variable Operations

Global variables are permanently allocated in a language definition and are accessible to all semantic procedures through two restricted operations:

```
PUT x y                 # store the value of local
                        # variable x in global
                        # variable y

GET x y                 # the inverse of PUT
```

There is no limit on the total number of global variables. The global variables `gp0`, `gp1`, ... can be defined and set from a PyElly command line (see Section 7); you can define other global variables yourself in semantic procedures by doing a `PUT` or a `GET` with a new global variable name. You can also initialize global variables with the `I:` option in a grammar rule file.

5.7 Control Structures

Only two structures are supported in generative semantics: the IF-ELIF-ELSE conditional and the WHILE loop; they are as follows:

```
IF x=string              # if local variable x has
                         # value string, execute the
                         # following block of code

ELIF x=string            # follows an IF; the test is
                         # made if all preceding
                         # tests failed and will
                         # control execution of
                         # following block of code
                         # (more than one ELIF can
                         # follow an IF)

ELSE                     # the alternative to take
                         # unconditionally after all
                         # preceding tests have failed

WHILE x=string           # the following block of code
                         # is repeatedly executed
                         # while the local variable
                         # x is equal to string

END                      # delimits a block of code and
                         # terminates an IF-ELIF-ELSE
                         # sequence or a WHILE loop
```

An END must terminate every IF-ELIF-ELSE sequence and every WHILE loop. PyElly will report a table definition error if any required END is missing for a control structure.

As in Subsection 5.4, we can check for single space characters here. For example,

```
IF x SP                  # check if local variable x is
                         # a space or a non-breaking
                         # space character

ELIF x SP                #

WHILE x SP               #
```

Instead of SP, you may also have HT, LF, NL , or CR. ; NL is the same as LF. New lines in a PyElly output buffer are always represented as a LF followed by an SP.

A tilde (~) preceding the variable name x reverses the logical sense of comparison in all the checks above.

```
IF ~x=string              # test if x ≠ string
```

The IF and ELIF commands also have a form that allow for the testing a variable against a list of strings. PyElly allows for

```
IF   x=s, t, u            # test if x == s or x == t or x == u
ELIF x=s, t, u            # test if x == s or x == t or x == u
```

The multiple strings to be compared against here must be separated by a comma (,) followed by a space. The space is essential for PyElly to recognize the listing here. The tests here can be negated with a tilde (~) also. The checking of multiple space characters SP, HT, NL, and CR as described above is unsupported in a list.

Within a WHILE loop, you may also have

```
BREAK                     # unconditionally break out
                          # of current WHILE loop

BREAKIF x=string          # if local variable x has
                          # value string, break out of
                          # current WHILE loop
```

The condition for BREAKIF can negated with a preceding tilde (~) as above. You can check for a single space character also.

5.8 Character Manipulation

These work with the current and next output buffers as indicated by < or > in a command; x specifies a source or target local variable to work with.

```
EXTRACT > x n             # drops the last n chars of
                          # the current output buffer and
                          # sets local variable x to the
                          # string of dropped characters

EXTRACT x < n             # drops the first n chars of
                          # the next output buffer and
                          # sets local variable x to the
                          # string of dropped characters

INSERT < x                # insert the chars of local
                          # variable x to the end of the
                          # current output buffer
```

```
INSERT x >              # insert the chars of local
                        # variable x to the start of the
                        # next output buffer

PEEK x <                # get a single char from
                        # start of next output buffer
                        # without removing it
PEEK > x                # gets a single char from
                        # end of current output buffer
                        # without removing it

DELETE n <              # deletes up to n chars from the
                        # start of the next output buffer

DELETE n >              # deletes up to n chars from the
                        # end of the current output
                        # buffer

STORE x k               # save last deletion in a current
                        # procedure in local variable
                        # except for last k chars when
                        # k > 0 or the first k chars
                        # when k < 0; if unspecified,
                        # k defaults to 0

SHIFT n <               # shifts up to n chars from
                        # the start of the next output
                        # buffer to the end of the
                        # current output buffer

SHIFT n >               # shifts up to n chars from
                        # the end of the current output
                        # buffer to the start of the
                        # next output buffer
```

If `n` is omitted for `EXTRACT` operation, it is assumed to be 1; if the < or > are omitted from a `DELETE` or a `SHIFT`, then < is assumed. These commands must always have at least one argument. All deleted text can be recovered by a `STORE` command.

The `DELETE` operation also has two variants

```
DELETE FROM s           # this deletes an indefinite
                        # number of chars starting from
                        # the string s in the current
                        # buffer up to the end
```

```
DELETE TO s              # this deletes an indefinite
                         # number of chars up to and
                         # including the string s
                         # in the next buffer
```

If the argument `s` is omitted for `DELETE FROM` or `DELETE TO`, it is taken to be the string consisting of a single space character. If `s` is not found in the current or the next buffer for `DELETE FROM` or `DELETE TO`, all of that buffer will be deleted. As with the regular `DELETE` operation, any characters removed by this command can be recovered in a local variable by the `STORE` command. Upper and lower case in `s` will matter.

5.9 Insert String From a Table Lookup

This operation that uses the value of a local variable to select a string for appending at the end of the current output buffer. It has the form

```
PICK x table             # select from table according
                         # to the value of x
```

The table argument is a literal string of the form

```
    (v1=s1#v2=s2#v3=s3#...vn=sn#)
```

where the # character must be used to terminate each table option. If the value of given local variable `x` is equal to substring `v`i for a table option, then substring `s`i will be inserted. If there is no match, nothing will be inserted, but when `v`n is null, then `s`n will be inserted if the variable `x` matches no other `v`i.

For example, the particular `PICK` operation

```
PICK x (uu=aaaa#vv=bbbb#ww=cccc#=dddd#)
```

in a generative semantic procedure or in a vocabulary table entry is equivalent to the PyElly code

```
IF   x=uu
    APPEND aaaa
ELIF x=vv
    APPEND bbbb
ELIF x=ww
    APPEND cccc
ELSE
    APPEND dddd
END
```

but the `IF-ELSE` form will take up multiple lines. PyElly will compile a `PICK` operation to use a Python hash object for faster lookup.

The operation

```
PICK x (=dddd#)
```

will append `dddd` for any `x`, including `x` being set to the null string. It is equivalent to offering no choice at al.

5.10 Buffer Searching

There is a general search operation in forward and reverse forms. These assume existence of a current and a new buffer as the result of executing `SPLIT` and `BACK`.

```
FIND s <                  # the contents of the new
                          # buffer will be shifted to the
                          # current buffer up to and including
                          # the first occurrence of string s

FIND s >                  # as above, but the transferring
                          # will be in the other direction
                          # past first occurrence of s
```

The substring `s` may not contain any spaces. If `s` is missing, but either < or > is specified, then `s` will be set to a single space character. If `s` is not found in a buffer scan, the entire contents of the buffer will be shifted. If `s` is specified and a final < or > is omitted, then > will be assumed. Note that `s` can include the characters < or >, which can look confusing, but will have the correct interpretation in what to search for. Note, however, that `FIND <` and `FIND >`, will always search for a single space.

A more specialized search allows you to align your current and new buffers at the start of an output line as marked by a previously executed `LINEFEED` command.

```
ALIGN <                   # shift chars in new buffer to current
                          # until a \n inserted by LINEFEED is
                          # found; the current buffer should
                          # end with \n followed by a space

ALIGN >                   # as above, but transfer will be in the
                          # other direction; any matching \n and
                          # following space will be left in the
                          # current buffer, however
```

`ALIGN` will be like `FIND` in that the entire source buffer be moved if no `\n` is found; but if found, the current buffer will always end up with <nl><sp> as its last two chars.

5.11 Execution Monitoring

To track the execution of semantic procedures when debugging them, you can use the command:

```
TRACE                    # show processing status in tree
```

In the semantic procedure for a phrase, this will print to the standard error stream the starting token position of the phrase in a sentence, its syntax type, the index number of the syntactic rule, the degree of branching of the rule, the generative semantic stack depth, the output buffer count, the number of characters in the current buffer:

```
TRACE @0 type=field rule=127 (1-br) stk=9 buf=1 (2 chars)
```

We see here that PyElly is running the semantics for the 1-branch rule 127 associated with a phrase of type `FIELD` at token position 0; it is executing at the 9th level of calls with a single buffer containing only two characters. If a subprocedure named `pn` (see Subsection 5.13) has called the current generative semantic procedure either directly or indirectly, then this will be identified also. The output above then becomes

```
TRACE @0 type=field rule=127 (1-br) stk=9 in (pn) buf=1 (2 chars)
```

If there are multiple named subprocedures in the chain of calls for the current generative semantic procedure, then only the most recent will be reported.

To show the current string value of a local variable `x`, you can insert this command into a semantic procedure:

```
SHOW x message ....      # show value of local variable x
```

This writes the ID number of the phrase being interpreted, the name of the variable being shown, its current string value, and an optional identifying message to the standard error stream. For example,

```
SHOW @phr 108 : [message ....] VAR x= [012345]
```

The message string is optional here; it may contain spaces.

To see up to the last `n` chars of the current and up to the first `n` of the next output buffer at the current point of running generative semantics, you can use a third command

```
 VIEW n                  # show n chars of current + next buffers
```

When executed in a generative semantic procedure, `VIEW 3` will write the following kind of message to the standard error stream:

```
VIEW = 0 @phr 6 : [u'u', u'n', u'>'] | [u'<', u's', u's'']
```

This first gives the index number of the current output buffer and the ID number of the phrase being interpreted. The vertical bar (`|`) separates the list of Unicode characters ending the current buffer from the list starting the next buffer. Set `n` to get as wide a window here as you need; if unspecified, the default for `n` is 5. Run a sequence of `VIEW`s to monitor progress in accumulating rewritten text for PyElly output.

5.12 Capitalization

PyElly has only two commands to handle upper and lower case in output.

```
CAPITALIZE              # capitalize the first char
                        # in the next buffer after a
                        # split and back operation
```

This operates only on the next output buffer. If you fail to do a `SPLIT` and `BACK` operation to create a next output buffer before running this command, you will get a null pointer exception, which will halt PyElly.

```
UNCAPITALIZE            # uncapitalize the first char
                        # in the next buffer after a
                        # split and back operation
```

The restrictions for `CAPITALIZE` apply here also.

5.13 Semantic Subprocedure Invocation

If `DO` is the name of a semantic subprocedure defined with `P:` in a PyElly grammar table, then it can be called from the generative semantic procedure for a grammar rule or from another subprocedure. This is done with a command that puts the name of the procedure into parentheses:

```
(DO)                    # call the procedure called DO
```

The subprocedure name should be defined somewhere in a PyElly `A.g.elly` file. This definition need not come before the call in the file. PyElly will always check whether a definition exists for every call, but will raise an error only when a call to a missing subprocedure is actually executed.

A subprocedure will run in the same environment as the code that it is called from and will see the same local and global variables. It can also define its own local variables, but when it finishes running, execution will return to just after where it was called. Any local variables in the subprocedure will then become undefined.

A subprocedure call will take no arguments. If you want to pass parameters, you must do so through local or global variables or in an output buffer. Similarly, results from a subprocedure can be returned only by putting them into an output buffer or passing them in a global or an external local variable still defined after a return.

The null subprocedure call `()` with no name is always defined; it is equivalent to a generative semantic procedure consisting of just a `RETURN`. This is normally used only for PyElly vocabulary definitions with no associated generative semantics (see Subsection 9.4).

If you know how to use command line tools like `grep`, you can find all subprocedure definitions and calls in your language definition files. A typo in a subprocedure name can produce a bad call, which will be treated by PyElly as equivalent to a `FAIL` operation. Try to make your subprocedure names as distinct as possible.

As of version 2.2, PyElly allows the definition of subprocedures in Python code, possibly in their own source files. The PyElly extendedContext.py module can be edited to link internal and external Python code to a subprocedure name, which can then be invoked in parentheses like subprocedure names defined in a regular PyElly grammar rule file.

Any subprocedure could be implemented within extendedContext.py, but making changes to them will require rebuilding PyElly and rerunning acceptance tests. In normal application development, one should stick to editing PyElly rule files. Use external subprocedures only if PyElly functionality has to be extended in a major way.

Currently, extendedContext.py defines a single subprocedure xngm. This breaks a word up into n-grams, n-letter word fragments employed to index text with only a finite set of keys. The command `(xngm)` reads input from an auxiliary buffer and writes output to the current main buffer. External module wordFragment.py does all the breakup.

6. Simple PyElly Rewriting Examples

We are now ready to look at some simple examples of semantic procedures for PyElly syntax rules, employing the mechanisms and operations defined in the preceding sections. Sections 8 and 9 will discuss more advanced capabilities.

6.1 Default Semantic Procedures

The notes in the Section 4.1.1 of this manual mentioned that omitting the semantic procedure for a syntax rule would result in a default procedure being assigned to it. Now we can finally define those default procedures. A rule of the form `G:X->Y Z` will have the default

```
_
  LEFT
  RIGHT
__
```

Note that a `RETURN` command is unnecessary here as it is implicit upon reaching the end of the procedure. You can always put one in yourself, however.

A rule of the form `G:X->Y` has the default semantic procedure

```
_
  LEFT
__
```

A rule of the form `D:w<-X` has the default

```
_
  OBTAIN
__
```

These are automatically defined by PyElly as subprocedures without names. They do nothing except to implement the calls and returns needed minimally to maintain communication between the semantic procedures for the syntactic rules associated with the structure of a sentence derived by a PyElly analysis.

In the first example of a default semantic procedure above, a call to the procedure for the left constituent structure `X` comes first, followed immediately by a call to the procedure for the right constituent `Y`. If you wanted instead to call the right constituent first, then you would have to supply your own explicit semantic procedure, writing

```
_
  RIGHT
  LEFT
__
```

In the second example above of a default semantic procedure, there is only one constituent in the syntactic rule, and this can be called as a left constituent or a right constituent; that is, a `RIGHT` call here will be interpreted as the same as `LEFT`.

In the third example of a default semantic procedure above, which defines a grammatical word, there is neither a left nor a right constituent; and so we can execute only an `OBTAIN`. Either a `LEFT` or a `RIGHT` command here would result in an error.

6.2 A Simple Grammar with Semantics

We now give an example of a nontrivial PyElly grammar. The problem of making subjects and predicates agree in French came up previously in Section 3. Here we make a start at a solution by handling *elison* and the present tense of first conjugation verbs in French and of the irregular verb AVOIR "to have." For the relationship between a subject and a predicate in the simplest possible sentence, we have the following syntactic rule plus semantic procedure.

```
G:SENT->SUBJ PRED

_
  VAR PERSON=3      # can be 1, 2, or 3
  VAR NUMBER=s      # singular or plural
  LEFT              # for subject
  SPLIT
  RIGHT             # for predicate
  BACK
  IF PERSON=1
    IF NUMBER=s
      EXTRACT X <   # letter at start of predicate
      IF X=a, e, è, é, i, o, u
        DELETE 1    # elison j'
        APPEND '    #
      ELSE
        BLANK       # otherwise, predicate is separate
        END
      INSERT < X    # put predicate letter back
      END
    ELSE
      BLANK         # predicate is separate
      END
  ELSE
    BLANK           # predicate is separate
    END
  MERGE             # combine subject and predicate
  APPEND !

__
```

The two local variables `NUMBER` and `PERSON` are for communication between the semantic procedures for `SUBJ` and `PRED`; they are set by default to "singular" and "third

person". The semantic procedure for SUBJ is called first with LEFT; then the semantic procedure for PRED is called with RIGHT, but with its output in a separate buffer. This lets us adjust the results of the two procedures before we actually merge them; here the commands in the conditional IF-ELSE clauses are to handle a special case of *elison* in French when the subject is first person singular and the verb begins with a vowel.

```
G:SUBJ->PRON
__
```

The above rule allows a subject to be a pronoun. The default semantic procedure for a syntactic rule of the form X->Y as described above applies here, since none is supplied explicitly.

```
D:i<-PRON
_
  APPEND je
  SET PERSON=1
__
D:you<-PRON
_
  APPEND vous
  SET PERSON=2
  SET NUMBER=p
__
D:it<-PRON
_
  APPEND il
__
D:we<-PRON
_
  APPEND nous
  SET PERSON=1
  SET NUMBER=p
__
D:they<-PRON
_
  APPEND ils
  SET NUMBER=p
__
```

These internal dictionary grammar rules define a few of the personal pronouns in English for translation. The semantic procedure for each rule appends the French equivalent of a pronoun and sets the PERSON and NUMBER local variables appropriately. Note that, if the defaults values for these variables apply, we can omit an explicit SET.

Continuing, we fill out the syntactic rules for our grammar.

```
G:PRED->VERB
__
```

This defines a single `VERB` as a possible `PRED`; the default semantic procedure applies again, since no procedure is supplied explicitly here.

Now we are going to define two subprocedures needed for the semantic procedures of our selection of French verbs.

```
P:plural

_
  PICK PERSON (1=ons#2=ez#3=ent#)

__
P:1cnjg

_
  IF NUMBER=s
    PICK PERSON (1=e#2=es#3=e#)
  ELSE
    (plural)
    END

__
```

Semantic subprocedures `plural` and `1cnjg` choose an inflectional ending for the present tense of French verbs. The first applies to most verbs; the second, to first conjugation verbs only. We need to call them in several places below and so define the subprocedures just once for economy and clarity.

```
D:sing<-VERB

_
  APPEND chant    # root of verb
  (1cnjg)         # for first conjugation inflection

__
D:have<-VERB

_
  IF NUMBER=s
    PICK PERSON (1=ai#2=ais#3=a#)
  ELSE
    IF PERSON=3
      APPEND ont  # 3rd person plural is irregular
    ELSE
      APPEND av   # 1st and 2nd person are regular
      (plural)
      END
    END

__
```

We are defining only two verbs to translate here. Other regular French verbs of the first conjugation can be added by following the example above for "sing". Their semantic procedures will all append their respective French roots to the current output buffer and call the subprocedure `1cnjg`.

The verb AVOIR is more difficult to handle because it is irregular in most of its present tense forms, and so its semantic procedure must check for many special cases. Every irregular verb must have its own special semantic procedure, but there are usually only a few dozen such verbs in any natural language.

Here is how PyElly will actually process input text with this simple grammar. The English text typed in for translation is shown in uppercase on one line, and the PyElly translation in French is shown in lowercase on the next line.

```
YOU SING
vous chantez!

THEY SING
ils chantent!

I HAVE
j'ai!

WE HAVE
nous avons!

THEY HAVE
ils ont!
```

The example of course is extremely limited as translations go, but the results are correct French, unlike those in Section 3. For more comprehensive processing, we would also take English inflectional stemming into account, use macro substitutions to take care of irregularities on the English side like `has`, and handle other subtleties. We also have to deal with various tenses other than present as well as aspect, mood, and so forth. You should, however, be able to envision now what a full PyElly grammar should look like for rewriting English as French; it will take much more work just to make the rules fairly complete, but the steps would be similar to what we already have seen above.

7. Running PyElly From a Command Line

We have so far described how to set up text files to define the various tables to guide PyElly operation. This section will show you how to run PyElly for actual language analysis, but first we must take care of some preliminaries. Everything should be fairly straightforward, but computer novices may want to get extra technical help here.

To begin with, PyElly as of release v2.0 was written entirely in Python 3.8, which has become the industry standard for the programming language. This is incompatible with Python 2.* required for releases of PyElly up to v1.6.*. So make sure you have the right version of Python. It is free software, and downloadable from the Web. The details for doing so will depend on your computing platform.

Once you have Python 3.8, you can download the full PyElly package from GitHub. This is open-source software under a BSD license, which means that you can do anything you want with PyElly as long as you identify in your own documentation where you got it. All PyElly Python source code is free, but still under copyright.

The Python code making up PyElly currently consists of 67 source files altogether with about 11,000 source code lines. A beginning PyElly user really needs to be familiar with only three of the modules.

`ellyConfiguration.py` - defines the default environment for PyElly processing. Edit this file to customize PyElly to your own needs. Most of the time, you can leave this module alone.

`ellyBase.py` - sets up and manages the processing of individual sentences from standard input. You can run this for testing or make it a part of a programming interface if you want to embed PyElly in a larger application.

`ellyMain.py` - runs PyElly from a command line. This is built on top of EllyBase and is set up to extract individual sentences from continuous text in standard input.

The ellyBase module reads in `*.*.elly` language definition files to generate the various tables to guide PyElly analysis of input data. Section 4 introduced three of them. For a given application `A`, these will be `A.g.elly`, `A.p.elly`, and `A.m.elly`, with only the `A.g.elly` file mandatory. Later sections of this user manual will introduce the other `*.*.elly` definition files.

The PyElly tables created for an application `A` will be automatically saved in two files: `A.grammar.elly.bin` and `A.vocabulary.elly.bin`. The first is a Python pickled file, which is not really binary since you can look at it with a text editor, but this will be hard for people to read. The second is an actual binary database file produced by SQLite from definitions in a given `A.v.elly`; this is the main way to define a PyElly vocabulary (see Subsection 9.4 for an explanation).

If the `*.elly.bin` files already exist, ellyBase will compare their creation dates with the modification dates of current corresponding `A.*.elly` definition files and create

new tables if any definition files have changed. If a `*.grammar.elly.bin` is more recent than a `*.vocabulary.elly.bin` file, then the latter will always be recompiled. Otherwise, the existing PyElly language rule tables will just be reloaded from the saved `*.elly.bin` files.

The files `*.grammar.elly.bin` will record the version of PyElly that they were created under. If this does not agree with the current version of PyElly, then PyElly will immediately exit with an error message that the grammar file is inconsistent. To proceed, you should then delete all current `*.elly.bin` files so that they will be regenerated automatically from your latest language definition files.

In most cases, ellyBase will try to substitute a file `default.x.elly` if an `A.x.elly` file is missing. This may not be what you want. You can override this behavior just by creating an empty `A.x.elly` file. The standard PyElly download package includes ten nontrivial examples of definition files for simple applications to show you how to set everything up (see Section 14).

You can see what ellyBase does by running it directly with the command line:

```
python ellyBase.py [name [depth]]
```

The `python` command here should be Python 3.*; on some platforms, you may have to invoke the `python3` command explicitly. This will first generate the PyElly tables for the specified application and then provide a detailed dump of grammar rules, allowing you to check for any problems in a language definition. The default application rules here will be `test` if `name` is unspecified. The resulting tables will be saved as `*.elly.bin` files that PyElly can later load directly to start up faster. If you do run the `test` application here, ellyBase will respond:

```
release  = PyElly v2.2.1
system   = test
  . . .
>
```

(Initialization output has been omitted here, indicated by the ellipsis.)

After initializing, ellyBase will prompt (with **>**) for one sentence in an input line. Its translation will be rewritten text in brackets if processing was successful or just `????` on failure. Output will also show all parse trees built for the syntactic analysis plus a detailed summary of the details of parsing. The optional `depth` argument above will limit how far down the reporting of parse trees will go (see `-d` for ellyMain below).

For an application with batch processing of input sentences not necessarily on separate lines, you normally will invoke ellyMain from a command line. The `ellyMain.py` file will read in arbitrary text and let you specify various options for PyElly processing. Its full command line is as follows in usual Unix or Linux documentation format:

```
python ellyMain.py [ -b][ -d n][ -g v0,v1,v2,…][ -p][ -noLang] [name] < text
```

where `name` is an application identifier like `A` above and `text` is an input source for PyElly to translate. If the identifier is omitted, the application defaults to `test`.

The command line flags here are all optional. They will have the following interpretations in PyElly ellyMain:

`-b`	operate in batch mode with no prompting; PyElly will otherwise run in interactive mode with prompting when its text input comes from a user terminal.
`-d n`	set the maximum depth for showing a PyElly parse tree to an integer n. This can be helpful when input sentences are quite long, and you do not want to see a full PyElly parse tree. Set n = 0 to disable parse trees completely. See Section 12 for more details.
`-g v0,v1,v2,..`	define the PyElly global variables gp0, gp1, gp2, ... for PyElly semantic procedures with the respective specified string values v0, v1, v2, ...
`-p`	show cognitive semantic plausibility scores along with translated output. If semantic concepts are defined, PyElly will also give the contextual concept of the last disambiguation according to the order of interpretation by generative semantics. This is intended mainly for debugging, but may be of use in some applications (see `disambig`, described in Section 14).
`-lang=x`	For input text to be processed, where x currently can be `EN` or `ZH`, for English or Chinese.
`-noLang`	do not assume that input text will be either English or Chinese; the main effect is to turn off English inflectional stemming (See Section 11). PyElly will still expect text in a Latin alphabet.

When ellyMain starts up in interactive mode, you will see a banner in the following form with some diagnostic output replaced by `. . .`:

```
PyElly v2.2.1, Natural Language Filtering
Copyright 2014-2021 under BSD open-source license by C.P. Mah
All rights reserved

reading test definitions
recompiling grammar rules
    . . .
recompiling vocabulary rules
    . . .

Enter text with one or more sentences per line.
End input with E-O-F character on its own line.

>>
```

You may now enter multiline text at the >> prompt. PyElly will process this exactly as it would handle text from a file or a pipe. Sentences can extend over several lines, or a single line can contain several sentences. PyElly will automatically find the sentence boundaries according to its current rules and divide up the text for analysis.

As soon as PyElly accumulates a full sentence, it will start a translation. In interactive mode, this will be after the first linefeed following the sentence because PyElly input will be read in full lines. A single linefeed will NOT break out of the ellyMain processing loop, but consecutive linefeeds will terminate a sentence even without punctuation. End all input with an EOF (control-D on Unix and Linux, control-Z in Windows). A keyboard interrupt (control-C) will break out of ellyMain immediately with no translation.

PyElly `*.*.elly` language definition files should be UTF-8 with arbitrary Unicode except in grammar symbol names. As its UTF-8 text input, however, PyElly will accept only ASCII, Latin-1 Supplement, Latin Extender A, some Extender B characters plus some other Unicode punctuation, and Greek lowercase; all other input characters will be converted to spaces or to underscores. The `chinese` application described in Section 14 will accept definition files with both traditional and simplified Chinese characters; the `zhong` application will also accept Chinese characters in input for translation.

All PyElly translations will be UTF-8 characters written to the standard output stream, which you may redirect to save to a file or pipe to modules outside of PyElly. PyElly parse trees and error messages will also be in UTF-8 and will go to the standard error stream, which you can also redirect. Historically, the predecessors of PyElly have been filters, which in Unix terminology means a program that reads from standard input and writes a translation to standard output.

Here is an example of interactive PyElly translation with a minimal set of language rules (`echo.*.elly`) defining the simple `echo` application with inflectional stemming:

```
>> Who gets the gnocchi?

=[who get -s the gnocchi?]
```

where the second line is actual output from ellyMain. PyElly by default converts upper case to lower, and `echo` strips off English inflectional endings as well as -ER and -EST. You can get stricter echoing by turning off English inflectional stemming and morphological analysis.

PyElly will first look for the definition files for an application in your current working directory. You can change this by editing the value for the symbol baseSource in `ellyConfiguration.py`. The definition files for the various PyElly applications described in Section 14 can be found in the `applcn` subdirectory under the main directory of Python source files, resources, and documentation. Configure here to your particular working environment.

PyElly *.py modules by default should be in your working directory, too. You can change where to look for them, but that involves resetting environment variables for Python itself. PyElly is written as separate Python modules to be found and linked up whenever you start up PyElly. This is in contrast to other programming languages where modules can be prelinked in a few executable files or packaged libraries.

There is a stripped-down implementation of top-level PyElly processing called ellySurvey.py, which ignores sentence boundaries and omits the parsing and rewriting of input text. Instead, this produces a listing of all the tokens found by PyElly along with source tags indicating how each was derived. This is invoked with the command line:

```
python ellySurvey.py [name] < text
```

where name is an application identifier and text is an input source to translate. If the identifier is omitted, the application defaults to test.

The ellySurvey listing of tokens will have the following source tags:

Ee	by entity extraction
Fa	by finite automaton for application
Id	in internal dictionary for application
Pu	by punctuation recognizer
Un	unknown type
Vt	in external vocabulary table for application

A token can have more than one source if multiple language rules define it; for example, a term may be in both an internal grammar dictionary and an external vocabulary table, and it might also be recognized by entity extraction and by the finite state automaton built into PyElly. Here is a token listing generated with the marking application:

```
Id on/On
Ee 09/16/____
Pu ,
Id his
Vt country
Vt take
FaId -ed
Id in
Vt at least
Fa 1500
Vt refugee/refugees
FaId -s
Vt flee/fleeing
FaId -ing
Vt war
Pu .
```

A token is shown in its analyzed form as it would appear in a PyElly parse tree; if this differs from its original input form after possible macro and other transformations, then that form is also given on the same line, separated by a slash (/). The listing makes it easier to find problems in tokenization or vocabulary lookup. Unknown tokens are identified in the listing, since you often will want to define them explicitly.

If you are processing a large text corpus for the first time, you should always run ellySurvey first. This should help identify the problems in that data solvable just by vocabulary definitions, which should then make subsequent grammar definitions easier.

On the whole, PyElly gives you many options for processing natural language input. You must, however, be comfortable with computing at the level of command lines in order to run PyElly in `ellyMain.py` or `ellyBase.py` or `ellySurvey.py`. There is as yet no graphical user interface for PyElly. The current PyElly implementation may be a challenge to computer novices unfamiliar with Python or with command line invocation.

8. Advanced Capabilities: Grammar

As noted above, PyElly language analysis is built around a parser for context-free languages to take advantage of extensive technology developed for parsing computer programming languages. So far, we have stayed strictly context-free except for macro substitution prior to parsing and use of local variables shared by generative semantic procedures to control translation.

You can actually accomplish a great deal with such basics alone, but for more challenging language analysis, PyElly supports other grammatical capabilities. These include extensions to grammar rules like syntactic and semantic features and the special `...` syntactic type mentioned earlier. Other extensions related to vocabularies are covered in the next section.

The handling of sentences and punctuation in continuous text is also normally a topic of grammar, but PyElly breaks this out as a separate level of processing for modularity. The details on this will be discussed in Section 11.

8.1 Syntactic Features

PyElly currently allows for only 112 distinctive syntactic types, including predefined types like `SENT` and `UNKN`. If needed, you get more types by redefining the variable `NMAX` in the PyElly file `symbolTable.py`, but there is a more convenient option here. PyElly also lets you qualify syntactic types through syntactic features, which in effect greatly multiplies the total number of syntactic types available.

Syntactic features are binary tags for syntactic types, allowing them to be divided into subcategories; they are best known from Noam Chomsky's seminal work *Syntactic Structures* (1957). Currently, PyElly allows up to sixteen syntactic features associated with a subset of syntactic types. You can define those subsets and name the features however you want. You can get more than sixteen syntactic features by redefining the variable `FMAX` in `symbolTable.py`, but this is not recommended.

The advantage of syntactic features is that grammar rules can disregard them. For example, a `DEFINITE` syntactic feature would allow definite noun phrases to be identified in a grammar rule without having to introduce a new structural type like `DNP`. Instead, we would have something like `NP[:DEFINITE]`. A grammar syntax rule like `PRED->VP NP` will apply to `NP[:DEFINITE]` as well as to plain `NP`. With a new syntax type like `DNP`, we would also have to add the rule `PRED->VP DNP`.

PyElly syntactic features are expressed by an optional bracketed qualifier appended to a syntactic structural type specified in a rule. The qualifier takes the form

```
[oF1,F2,F3,…,Fn]
```

where "o" is a single-character identifier for a set of feature names for a specific subset of syntactic types and `F1`, ..., `Fn` are the actual names composed of alphanumeric characters, possibly preceded by a prefix '–' or '+'. For clarity, set identifiers should be punctuation characters, but never be from the set { '+' , '–' , '*' , '[' , ']' , ',' }.

Allowing multiple sets of feature names is for convenience only. Each set will have to refer to the same `FMAX` feature bits defined for each phrase node in a PyElly parse tree. When defining multiple name sets, make sure that their usage is consistent. PyElly will reject a syntactic type occurring with syntactic feature names from more than one set because features with the same name in different rules may refer to different bits.

Bracketed syntactic features in language rules must follow a syntactic type name with no intervening space. Spaces may follow a comma in a list of syntactic feature names for easier reading, but any before or after a starting left bracket (`[`) will be seen as an error.

A syntax rule with feature names might appear as follows:

```
G:NP[:DEFINITE,*RIGHT]->THE NP[:-DEFINITE,*RIGHT]
```

This specifies a rule of the form `NP->THE NP`, but with additional restrictions on applicability. The `NP` as specified on the right side of the rule must have the feature `*RIGHT`, but not `DEFINITE`. If the condition is met, then the resulting `NP` structure as specified on the left of the rule is defined with the features `DEFINITE` and `*RIGHT`. The ':' is the feature class identifier here for the `DEFINITE` and `*RIGHT` feature names.

PyElly has no upper limit on the number of different sets, but have as many as you want as long as you can tell them apart. Remember that syntactic features are supposed to simplify grammars and let you define fewer rules. A syntactic feature name should include only ASCII letters and digits, with the case of letters not mattering.

The special feature name `*RIGHT` (or equivalently `*R`) will be defined automatically for all syntactic feature sets. Setting this feature on the left side of a syntactic rule will have the side effect of making that constituent structure inherit any and all syntactic features of its rightmost immediate subconstituent as specified in the rule. This provides a convenient mechanism for passing syntactic features up a parse tree without having to say what exactly they are.

The special feature name `*LEFT` (or equivalently `*L`) will also be defined automatically. This will work like `*RIGHT`, except that inheritance will be from the leftmost immediate subconstituent. It is permissible to inherit both ways. The `*LEFT` and `*RIGHT` syntactic features for a phrase node will be mutually exclusive, however; setting one will turn off the other. With a one-branch rule, `*LEFT` and `*RIGHT` will be the same for inheritance, though remaining distinct as syntactic features.

A third special feature name `*UNIQUE` (or equivalently `*U` or `*X`) will also be in every PyElly syntactic feature set. Its main purpose is to prevent a phrase from matching any other phrase in PyElly ambiguity checking while parsing; it cannot be inherited. It can,

however, be a regular syntactic feature at a leaf node where there is no inheritance. No starred (*) special syntactic features should ever be redefined by a PyElly user.They will always be counted in the total number of syntactic features available for any grammar.

A feature F can be marked with a '–' on the left side of a grammatical rule; for example, X[:*L,-F]—>Y[:F,-G]. This has a different interpretation than that for a feature on the right side of a rule, such as G for the syntactic category Y in the example rule. It serves to turn off a particular feature that might have been inherited, in this case F.

You may also turn off a feature that was just set as in Z[:*R,-*R]—>W[:H]; this makes sense only with the special *L and *R syntactic features, which have side effects when turned on that will persist even after they are turned back off. The result here is that the resulting Z phrase node will inherit the feature H, but will not have *R also turned on in the end. The action is inherent to the PyElly implementation of feature inheritance, but can be helpful when you use the *L and *R features to distinguish leaf phrase nodes.

8.2 The . . . Syntactic Type

When the . . . type shows up in a grammar, PyElly automatically defines a syntax rule that allows phrases to be empty. If you could write it out, the rule would take the form

```
...->
```

This is sometimes called a zero rule, which PyElly will not allow you to specify explicitly in a *.g.elly file for any syntactic type on the left. In strict context-free grammars, any rule having a syntactic structural type going to an empty phrase is forbidden. Such rules are allowed only in so-called type 0 grammars, the most unrestricted of all; but the languages described by such grammars tend to be avoided in text processing because of the difficulty in parsing them.

With . . . as a special syntactic type, however, PyElly achieves much of the power of type 0 grammars without giving up the parsing advantages of context-free grammars. The advantage with using . . . in PyElly is that it allows a grammar to be more compact when this syntactic type is applicable. For example, suppose that we have the rules

```
z->x a
z->x b
z->x c
z->x d
z->a
z->b
z->c
z->c
x->unkn
x->x unkn
```

where `unkn` is the predefined PyElly syntactic category introduced in Section 4 (this will be explained more fully in Section 9.1). Now if `x` is not of interest to us in an eventual translation, then we can replace all the above with just the rules

```
z->... a
z->... b
z->... c
z->... d
...->unkn
...->... unkn
```

The **`...`** type was intended specifically to support keyword parsing, which tries to recognize a limited number of words in input text and more or less ignores anything else. A PyElly grammar to support such parsing can always be written without **`...`**, but may be unwieldy. The `doctor` application for PyElly illustrates how this kind of keyword grammar would be set up; it includes syntax rules like the following:

```
g:ss->x ...

__
g:x[@*right]-> ... key

__
g:...->unkn ...
```

The syntactic type `key` here represents all the various kinds of key phrases to recognize in a psychiatric dialog; for example, "mother" and "dream". We can get away with only one syntactic type here because, with about a dozen syntactic features available for it, we can distinguish between 4095 different kinds of key phrases.

The actual responses of our script will be produced by semantic procedures for the rules defining `x[@*right]` phrases. Note that different responses to the same keyword must be listed as separate rules with the same syntactic category and features. A simplified listing of grammar rules here might be

```
g:sent[@*right]->ss

__
g:x->... key

__
g:key[@ 0,1]->fmly

__
g:ss[.*right]->x[@ 0, 1,-2,-3,-4,-5,-6] ...

_
  append TELL ME MORE ABOUT YOUR FAMILY

__
g:ss[.*right]->x[@ 0, 1,-2,-3,-4,-5,-6] ...

_
  append WHO ELSE IN YOUR FAMILY

__
d:mother <- fmly
```

```
__
g:...->unkn

__
g:...->unkn ...
__
```

This defines two different possible responses for `key[@ 0,1]` in our input. PyElly ambiguity handling will then automatically alternate between them (see Section 10).

The grammar here is incomplete, recognizing only sentences with a single keyword and nothing else. To allow for sentences without a keyword, we also need a rule like

```
g:ss->...
__
```

The **...** syntactic type also has the restriction that you cannot specify syntactic features for it. If you do put something like `...[.F1,F2,F3]` in a PyElly rule, it be treated as just **...**. This is mainly to help out the PyElly parser, which is already working quite hard here; but it also is because PyElly needs to use such features for its own internal purposes (see Subsection 12.3.3).

PyElly will also block you from defining a rule like

```
g:...->...
__
```

or like

```
g:X->... ...
__
```

where `X` is any PyElly syntactic type, including **...**.

The **...** syntactic type can be quite tricky to use effectively in a language description, but it is also tricky for PyElly to handle as an extension to its basic context-free parsing. The various restrictions here are a reasonable compromise to let us do most of what we really need to do. See Subsection 12.3.3 for details on how PyElly parsing actually uses grammar rules containing the syntactic type **...**.

9. Advanced Capabilities: Vocabulary

PyElly operates by reading in, analyzing, and rewriting out sentences. To do this, it requires syntactic and semantic information for every text element that it encounters: words, names, numbers, identifiers, punctuation, and so forth. Certain text elements like punctuation will be fairly limited, but defining all the rest can be a big undertaking even for fairly simple applications.

In all our PyElly examples here so far, we have already seen several ways of defining text elements in a language.

- An explicit `D:` rule in a grammar.
- Assignment of syntactic information through matching of specified patterns.
- Using the predefined `UNKN` syntactic type when a definition is lacking.

These are fine with small vocabularies, but useful natural language applications must deal with hundreds or even tens of thousands of distinct terms. These may not fall into obvious patterns; and stuffing them all into a `*.g.elly` grammar file will demand more keyboard entry than most people care to do. Treating most text elements as `UNKN` is always a fallback option, but this works well only with simple grammars.

There is no perfect solution here. PyElly can only try to provide a user enough vocabulary definition options to make the overall task a little less painful. So, in addition to the methods above, PyElly also incorporates builtin morphological analysis of unknown words to infer a syntactic type, plug-in code for recognizing complex entities like numbers, time, and dates, and vocabulary tables loaded from external databases. These will be described in separate subsections below, but we first should explain better how the `UNKN` syntactic type works.

9.1 More on the UNKN Syntactic Type

We have run across the `UNKN` syntactic type several times already in this manual. Whenever text element `xxxx` in its input cannot be otherwise identified by PyElly, it will be assigned the type `UNKN`. In effect, PyElly generates a temporary rule of the form:

```
D:xxxx <- UNKN
_
  OBTAIN
__
```

Such a rule is in effect only while PyElly is processing the current input sentence.

By itself, `UNKN` solves nothing. It just gives PyElly a handle for working with unknown elements, and you still are responsible for supplying the grammar rules and associated semantics to tell PyElly how to interpret a sentence having `UNKN` as one of its subconstituents. The simplest possibility here is to make some guesses; for example,

```
G:NOUN->UNKN
———
G:VERB->UNKN
——
```

These two rules allow an unknown word to be treated as either a noun or a verb. So, when given a sentence containing unknown `xxxx`, PyElly can try to analyze it as either a noun or a verb. If only one results in a successful parse, then we have managed to get past the problem of having no definition for `xxxx`. If neither works out, we have lost nothing; if both work out, then PyElly can try to figure out which is the more plausible using the cognitive semantic facilities described in Section 10.

9.2 Breaking Down Unknown Words

An unknown word can also be resolved by looking at how it is put together. For example, the word UNREALIZABLE may be missing from a vocabulary, but it could be broken down into UN+ +REAL -IZE -ABLE, allowing us to identify it as an adjective based on the root word REAL, which is more likely to be defined already in a vocabulary. PyElly develops this idea further, and this will be described in the immediately following subsections.

Text document search engines fifty years ago were already using word analysis to reduce the size of their keyword indexes. This was to manage the many variations a search term might take: MYSTERY versus MYSTERIES as well as MYSTIFY, MYSTICISM, and MYSTERIOUS. Since these all revolve around a common concept, many system builders opt to reduce them all to the single term MYSTERY in a search index. This is also helpful for maximizing the number of relevant documents retrieved for a query.

Consequently, many kinds of rule- and table-driven word stemming exist. These can often be rather crude like the Porter algorithm, but we can do it much more reliably if we work long enough at refining the rules for stemming. For English at least, PyElly now has two quite separate PyElly tools for analyzing the structure of words as a way of dealing with unknown terms.

9.2.1 Inflectional Stemming

An inflection is a change in the form of a word reflecting its grammatical use in a sentence. Indo-European languages, including English, tend to be highly inflected; and in instances like Russian, the form of most words can vary greatly to indicate person, number, tense, aspect, mood, and case. Modern English, however, has kept only a few of the inflections of Old English, and so it has been easier to formulate rules to characterize how a particular word can vary.

PyElly inflectional stemming currently recognizes only six endings for English: -S, -ED, -ING, -N, -T, and -'S. These each have their own associated stemming logic and also share additional logic for recovering the uninflected form of a word. This logic is based

on American English spelling rules along with many special cases. It is packaged in two distinct Python modules, one employing binary decision trees and the other running hand-coded Python logic.

The module `inflectionStemmerEN.py` is the oldest part of PyElly, dating back to PARLEZ, its original ancestor written in DEC PDP-11 assembly language. This module loads binary decision trees from external files at run-time. The more recent module `deinflectionMatching.py` was developed in PyElly for use with external dictionary lookup and entity extraction (see Subsections 9.3 and 9.4 below). This module has simple default logic written in Python, which you can supersede.

Most PyElly inflectional stemming is with binary decision trees, If an unknown word ends in -S, -ED, -ING, -N, or -T, PyElly will apply predefined logic for the ending to see whether it is an inflection and, if so, what the uninflected word should be. Though such logic is necessarily incomplete, it has seen over forty years of use in various systems and is quite accurate for American spellings of most English words. For example,

```
winnings ==> win -ing -s
placed   ==> place -ed
judging  ==> judge -ing
cities   ==> city -s
bring    ==> bring
sworn    ==> swear -n
meant    ==> mean -t
```

PyElly stemming will automatically prepend a hyphen (-) on any split off word ending so that it can be recognized. The original word in the PyElly input stream is then replaced by the uninflected word followed by the removed endings as shown. Each ending will become a separate token in PyElly parsing.

In some applications, you may just want to ignore the removed word endings, but these can be quite valuable for figuring out unknown words. The -ED, -ING, -N, and -T endings indicate a verb, and you can provide grammar rules to exploit that syntactic information. For example,

```
D:-ED <- ED

__
D:-T  <- ED

__
D:-N  <- ED

__
G:VERB[|ED]->UNKN ED

__
```

To use English inflectional stemming in PyElly, set the `language` variable in the `ellyConfiguration.py` file to `EN`. To override such stemming just for a particular word, define that word in a vocabulary table entry so that PyElly can recognize its inflected form without analysis.

The logic for an ending `X` is defined by a text file `X.sl` loaded by PyElly at runtime. You can also define your own inflectional stemming logic by editing the current `*.sl` files or by writing new ones. The current files for English are `Stbl.sl`, `EDtbl.sl`, `INGtbl.sl`, `Ntbl.sl`, `Ttbl.sl`, `rest-tbl.sl`, `spec-tbl.sl`, and `undb-tbl.sl`. To do inflectional stemming for a new language `ZZ`, you will have to write the `*.sl` files and a `inflectionStemmerZZ.py`. Use `inflectionStemmerEN.py` as a model here.

Here is a segment of actual logic from `Stbl.sl`, which tells PyElly what to check in identifying a -S inflectional ending when it is preceded by an IE. The literal strings for comparison in the logic below have their characters in reverse order because PyElly will be matching from the end of a word towards its start.

```
IF ei
  IF tros {SU}
  IF koo {SU}
  IF vo {SU}
  IF rola {SU}
  IF ppuy {SU}
  IF re
    IF s
      IF im {SU 2 y}
      END {FA}
    IF to {SU}
    END {SU 2 y}
  IF t
    IS iu {SU 2 y}
    LEN = 6 {SU}
    END
  END {SU 2 y}
```

This approximately translates to

if you see an IE at the current character position, back up and
  if you then see SORT, succeed.
  if you then see OOK, succeed.
  if you then see OV, succeed.
  if you then see ALOR, succeed.
  if you then see YUPP, succeed.
  if you then see ER, back up and
    if you then see S, back up and
      if you then see MI, succeed, but drop the word's last two letters and add Y.
      otherwise fail.
    if you then see OT, then succeed.
    otherwise succeed, but drop the word's last two letters and add Y.
  if you then see T, then back up and
    if you then see a I or a U, then succeed, but drop the word's last two letters and add Y.
    if the word's length is 6 characters, then succeed.
  otherwise succeed, but drop the word's last two letters and add Y.

This stemming logic is equivalent to a finite state automaton (FSA). Its operation should be fairly transparent, although the total number of different rules for English inflections has grown to be quite extensive. You may nevertheless eventually run into a case that is handled incorrectly and that you will want to add to the rules. Make sure, however, to test out every change so that you can avoid making other things worse.

9.2.2 Morphology

Morphology in general is about how words are put together, including processes like BLACK + BIRD ==> BLACKBIRD, EMBODY + -MENT ==> EMBODIMENT, and KOREAN + POP ==> K-POP. PyElly morphological analysis is currently limited to that involving the addition of prefixes or suffixes to a root, which is not necessarily a word.

The morphology component of PyElly started out as simple decision tree logic for just removing common endings from English words, including -S, -ED, and -ING. It has now evolved to focus on non-inflectional endings and to output the actual affixes removed as well as the final root form. Inflectional endings are now handle in a separate module.

Earlier above, we saw "unrealizable" broken down into UN+, +REAL, -IZE, and -ABLE. True morphological analysis here would also tell us that the -IZE suffix changes the word REAL into a verb, the -ABLE suffix changes the verb REALIZE into an adjective again, and the prefix UN+ negates the sense of the adjective REALIZABLE. This is what PyElly can now do, which is useful in figuring out the syntactic type of unknown words.

For an application `A`, PyElly will work with prefixes and suffixes through two language rule tables defined by files `A.ptl.elly` and `A.stl.elly`, respectively. These are akin to the grammar, macro substitution, and word pattern tables already described. We have two separate files here because suffixes tend to be more significant for analyses than prefixes, and it is common to do nothing at all with prefixes.

PyElly morphological analysis will be applied only to words that otherwise would have the `UNKN` syntactic type after all other lookup and pattern matching has been done. The result will be similar to what we see with inflectional stemming; and to take advantage of them, you will also have to add the grammar rules to recognize prefixes and suffixes and incorporate them into an overall analysis of an input sentence.

9.2.2.1 Word Endings (`A.stl.elly`)

PyElly suffix analysis will be done only after removal of any inflectional endings. For application `A`, the `A.stl.elly` file guiding this will contain a series of patterns and actions like the following:

```
abular 2 2 le.
dacy 1 2 te. 1
entry 1 4 .
gual 2 3 . 0a
ilitation 2 6 &,
ion 2 0 .
lenger 2 5 . 0e
oarsen 1 5 .
piracy 1 4 te. 1
santry 1 4
tention 1 3 d.
uriate 2 2 y.
worship 0 0 .
|carriage 0 0 .
|safer 1 5 . 0e
```

Each line of a `*.stl.elly` file defines a single pattern plus actions to be taken upon matching. The format of a rule is as follows from left to right:

- A word ending to look for. This does not have to correspond exactly to an actual morphological suffix and its context; the actions associated with a rule will define that suffix. The vertical bar (`|`) at the start of a pattern string matches the start of a word.
- A single digit specifying a contextual condition for an ending to match: 0= avoid any stemming for this match, 1= no conditions apply, 2= the ending must be preceded by a consonant, and 3= the ending must be preceded by a consonant or U.
- A number specifying how many of the characters of the matched characters to keep as a part of a word after removal of a morphological suffix. A starting vertical bar (`|`) in a listed ending will count as one character here.
- A string specifying what letters to add to a word after removal of a morphological suffix. An `&` in this string is conditional addition of e in English words, applying a method defined in English inflectional stemming.
- A period (`.`) indicates that no further morphological analysis be applied to the result of matching a suffix rule and carrying out the associated actions; a comma (`,`) here means to continue morphological analysis recursively. This is mandatory.
- A number indicating how many of the starting characters of the unkept part of a matching ending to drop to get a morphological suffix to be reported in an analysis.
- A string specifying what letters to add to the front of the reduced unkept part of a matching ending in order to make a full morphological suffix.

In applying such pattern rules to analyze a word, PyElly will always take the longest match. For example, if the end of a word matches the LENGER pattern above, then PyElly will ignore the shorter matches of a ENGER pattern or a GER pattern.

In the LENGER rule above, PyElly will accept a match at the end of word only if preceded by a consonant in the word. On a match, the rule specifies to keep 5 of the matched characters in the resulting root word. From the rest of a matched ending, PyElly will drop no characters, but add an E in front to get the actual suffix removed.

So the word CHALLENGER will be analyzed as follows according to the suffix patterns above:

CHAL **LENGER**	(split off matched ending and check preceding letter)
CHALLENGE **R**	(move five characters of matched ending to resulting word)
CHALLENGE **ER**	(add E to remaining matched ending to get actual suffix -ER)

The period (`.`) in the action for LENGER specifies no further morphological analysis. With a comma (`,`), PyElly would continue, possibly producing a sequence of different suffixes by reapplying its rules to the word resulting from preceding analyses. This can continue indefinitely, with the only restriction being that PyElly will stop trying to remove endings when a word is shorter than three letters.

To recognize the stripped off morphological suffixes in a grammar, you should define rules like

```
D:-ion <- SUFFIX[:NOUN]
__
```

and then add `G:` grammar rules for dealing with these syntactic types as in the case of inflections. For example,

```
G:NOUN->UNKN SUFFIX[:NOUN]
__
```

A full grammar would of course have to be ready to deal with many different morphological suffixes.

The PyElly file `default.stl.elly` is a fairly comprehensive compilation of English word endings evolving over the past fifty years and covering most of the non-foreign irregular forms listed in WordNet exception files. If there is more than one possible analysis, PyElly cannot handle; for example RENT is not reduced to REND. If you want to do something like this, then you have to supply your own grammar rules.

The `default.stl.elly` file also includes transformations of English irregular inflectional forms, which actually involve no suffix removal. For example, DUG becomes DIG -ED. This cannot be handled by PyElly inflectional stemming logic. Altogether, there are over 2,600 morphological rules defined.

9.2.2.2 Word Beginnings (`A.ptl.elly`)

For prefixes, PyElly works with patterns exactly as with suffixes, except that they are matched from the beginning of a word. For example

```
contra 1 0 .
hydro 1 0 .
non 2 0.
noness 1 3.    # for nonessential
pseudo 1 0 .
quasi 1 0 .
retro 1 0 .
tele 1 0 .
trans 1 0 .
under 1 0 .
```

The format for patterns and actions here is the same as for word endings. As with endings, PyElly will take the action for the longest pattern matched at the beginning of a word being analyzed.

Prefixes will be matched after suffixes and inflections have been removed. Removing a prefix must leave at least three characters in the remaining word. Actions associated with the match of a prefix will typically be much simpler than those for suffixes, and rules for prefixes will tend to be as simple as those in the example above.

PyElly removal of prefixes will be slightly different from for suffixes. With suffixes, the word STANDING becomes analyzed as STAND -ING, but with the prefix rules above, UNDERSTAND would become UNDER+ +STAND. Note that a trailing + is used to mark a removed prefix instead of a leading – for suffixes. Unlike with suffixes, only a single prefix rule at a time can be applied to a word being looked up.

In the overall scheme of PyElly processing of an unknown word, inflections are checked first, then suffixes, and finally prefixes. If there is any overlap between the suffixes and the prefixes here, then inflections and suffixes takes priority. Prefix rules tend to be more unreliable than suffix rules except in special domains like chemical names.

For example, NONFUNCTIONING becomes NON+ +FUNCT -ION -ING with the morphology rules above. A grammar would then have to stitch these parts back together in an analysis.

To work with prefixes in a grammar, you will need a dictionary rule like

```
D:non+ <- PREFIX[+NEG]
__
```

and you should by now know how to supply the required grammar rules yourself.

9.2.3 N-Gram Indexing

PyElly has been extended to support word-fragment analysis of text. This involves breaking up a word into a set of 2-, 3-, 4-, and 5-letter fragments so as to provide a way of indexing the content of text with only a finite set of keys. With only finitely many keys, a word in text will often be missing from an index set; but then, we can resort to noting the major n-letter fragments or n-grams that the word contains.

N-gram indexing is noisy, but in many text processing applications, it can suffice. The PyElly paradigm is limited to rewriting input text as output text, but this now extends to the analysis of any unknown word into predefined set of n-grams. This capability is theoretically possible with PyElly generative semantics alone, but for practice use, it is easier to give this task to an external procedure.

PyElly 2.2 allows external code for complex algorithms to be call as a defined generative semantic subprocedure. This is hardwired into PyElly in its extendedContext.py module. Here is an example of n-grams extracted for a word by the subprocedure call `(xngm)` with a word DEMONSTRATE in the current auxiliary buffer for PyElly output.

```
demonstrate
demo
  mon
   onst
    nstr
     stra
       rate
```

The n -grams are allowed to overlap. The details of their derivation are defined in the Python module wordFragment.py. N-gram indexing is peripheral to PyElly functionality, but PyElly expertise with inflectional and morphological endings in other areas can greatly help to get more information from n-gram indices.

9.3 Entity Extraction

In computational linguistics, an entity is some phrase in text that stands for something specific that we can talk about. This is often a name like George R. R. Martin or North Carolina or a title like POTUS or the Bambino; but it also can be insubstantial like Flight VX 84, 888-CAR-TALK, 2.718281828, NASDAQ APPL, or orotidine 5'-phosphate.

The main problem with entities is that almost none of them will normally be found in a predefined vocabulary. People seem to handle them in stride while reading text, however, even when they are unsure what a given entity means exactly. This is in fact the purpose of much text that we read: to inform us about something we might be unfamiliar with. A fully competent natural language system must be able to function in this kind of situation.

At the beginning of the 21st Century, systems for automatic entity extraction from text were all the rage for a short while. Various commercial products with impressive capabilities came on the market, but unfortunately, just identifying entities is insufficient to build a compelling application, and so entity extraction systems mostly fell by the wayside in the commercial marketplace. In a tool like PyElly, however, some builtin entity extraction support can be quite valuable.

9.3.1 Numbers

PyElly no longer has a predefined `NUM` syntactic type. The PyElly predecessor written in C did have compiled code for number recognition, but this covered only a few possible formats and was dropped later in Jelly and PyElly for a more flexible solution. If you want PyElly to recognize literal numbers in text input, you must make use of special patterns in files `*.p.elly` as described in Section 4.2.

PyElly, however, also has gone further here. It also has some builtin capabilities for automatic normalizations of number references so that you need fewer patterns to recognize them. In particular,

- Automatic stripping out of commas in numbers as an alternative to doing this with special pattern matching:

  1,000,000 ==> 1000000.

- Automatic mapping of spelled out numbers to a numerical form:

  one hundred forty-third ==> 143rd

  fifteen hundred and eight ==> 1508

Here you still need patterns to recognize the rewritten numbers so that PyElly can process them. You can disable all such number rewriting by setting the variable `ellyConfiguration.rewriteNumbers` to False.

9.3.2 Dates and Times

Dates and Times could be handled as PyElly patterns, but their forms can vary so much that this would take an extremely complicated finite-state automaton. For example, here are just two of many possible kinds of dates:

```
the Fourth of July, 1776
2001/9/11
```

To recognize such entities, the PyElly module `extractionProcedure.py` defines some date and time extraction methods written in Python that can be called automatically when processing input text.

To make such methods available to PyElly, they just have to be listed in the `ellyConfiguration.py` module. Here is some actual Python code to do so:

```
import extractionProcedure

extractors = [  # list out extraction procedures
  [ extractionProcedure.date , 'date'] ,
  [ extractionProcedure.time , 'time']
]
```

You can disable date or time extraction by just removing its method name from the extractors list. The second element in each listed entry is a syntax specification string, generally indicating a syntactic category plus syntactic features to be given to a successfully extracted entity; these should be coordinated with other PyElly grammar rules. An optional third element is a semantic feature specification string, which can be '–' for no features. An optional fourth element is an integer plausibility scoring.

The date and time methods above are part of the standard PyElly distribution. These will do some normalization of text before trying to recognize dates and times. Dates will be rewritten in the form

mm/dd/yyyyXX

For example, 09/11/2001ad. Times will be converted to a 24-hour notation

hh:mm:ssZZZ

For example, 15:22:17est. If date or time extraction is turned on, then your grammar rules should expect to expect to see these forms when a generative semantic procedure executes an `OBTAIN` command. The XX epoch indicator in a date and the ZZZ zone indicator in a time may be omitted in PyElly input.

9.3.3 Names of Persons (`A.n.elly`)

Natural language text often contains the names of persons and of things. These can be handled in various ways within PyElly, but names generally will present unique problems for both syntactic and semantic analysis. For example, entirely new names or old names with unusual spellings often show up in text, but it is hard to anticipate them in a vocabulary table or even a pattern table. Also, a name can appear in multiple forms in the same text: Joanne Rowling, J.K. Rowling, Rowling, Ms. Rowling.

To help out here, PyElly incorporates a capability for heuristically recognizing personal names and their variations in natural language text. This will automatically be configured into PyElly whenever a user runs an application `A` that includes a `A.n.elly` rule file. Name recognition will run independently of other PyElly language analysis, but will create parse tree leaf nodes with the syntactic type `NAME` for any names that it is able to identify.

Each rule in an `A.n.elly` file will be in one of two forms:

```
X : T

=PPPP
```

The first form associates a type with a specified name component; it consists of a pattern `X` for a component followed by a colon (`:`) with optional spaces around it and followed by a name component type `T`. The second form lists a phonetic pattern `PPPP` (see below) that will be used to validate inferred component types that are otherwise unknown; the pattern must be preceded by an equal sign (=) with no space after it.

9.3.3.1 Explicit Name Component Patterns and Types

PyElly predefines 12 component types for name recognition; these are not syntactic categories. Currently these are indicated by three-letter identifiers as follows:

`REJ`	reject name with this component
`STP`	stop any scan for a name
`TTL`	a title like "Captain"
`HON`	an honorific like "The Honorable"
`PNM`	a personal name
`SNM`	a surname
`XNM`	a personal name or surname
`SNG`	possible single name
`INI`	an initial like "C."
`REL`	a relation like "von"
`CNJ`	a conjunction like 'y'
`GEN`	a generation tag like 'Jr.'

One of these types must be the `T` part of a `X : T` rule in an `A.n.elly`. Anything else will cause an error exception during table generation. The case of an identifier here will be unimportant.

The `X` pattern part of a `X : T` rule must be a string of ASCII letters possibly including spaces; a string without spaces can also optionally start or end with a + or −. The possibilities here are

`abc de`	matches the exact string "abc de"
`abc-`	matches a string of letters starting with "abc"
`abc+`	matches a string of letters starting with "abc", but the rest of the string must also match another table entry
`-abc`	matches a string of letters ending with "abc"
`+abc`	matches a string of letters ending with "abc", but the rest of the string must also match another table entry

The `X` part of a type rule will always delineate a single name component, although this might have multiple parts like `de la`.as in `George de la Tour`. The basic idea here is to provide the various possible parts of a name, which will then be combined by hard-coded PyElly logic to report actual names and name fragments in input text within the existing framework of PyElly entity extraction.

Here some name rules in a PyElly `*.n.elly` file:

```
# simple name table definition
# example.n.elly

John  : PNM
Smith : SNM
Kelly : XNM
Mr.   : TTL
Sir   : TTL
III   : GEN
de la : REL
Fitz- : SNM
+son  : SNM
-aux  : SNM
y     : CNJ
prince: SNG
university : REJ
```

With these rules, "Fitzgerald" and "FitzABBA" will be recognized as surname components, while "Peterson" will be recognized as a surname only if "Peter" is also recognized as a name component. Upper and lower case will not matter in the rules here, nor will the ordering of the rules. Comments for documentation take the same form as in other PyElly rule files, a line starting with '`#`' or the rest of a line after ' `#` '.

9.3.3.2 Implicit Name Components

In any PyElly application that has to recognize personal names, the most reliable approach is to maintain lists of the most commonly expected name components. These are fairly easy to compile with the resources available on the Worldwide Web, but no listing here will ever be complete. Various rules of thumb can help us to find unknown names, but this is guessing, and we really have to make only a few mistakes here.

For example, if every capitalized word is a possible name component, then we can get text items like ABC, The, University, Gminor. and Ltd. Such results will diminish the value of the true names that we do find. So, we have to be quite strict about the criteria for judging a string to be a possible name component:

1. The string is alphabetic, with at least four letters. Anything shorter can be listed explicitly in a name table to eliminate guessing.
2. Its first letter is capitalized. An explicitly known name component can leave off capitals, but any inference of a name must have as much support as possible.
3. Its adjacent digraphs (e.g. `ab`, `bc`, and `cd` in the string `abcd`) are all in common digraphs for first names in the 2010 U.S. census when a candidate string has six or fewer characters. It may have all but one of its digraphs be common when a string has seven or more characters.
4. It occurs along with at least one explicitly known name component. That is, a name cannot consist completely of inferred name components.
5. If the first three conditions above are met, and its phonetic signature matches the signature for common name components explicitly known, then an inferred name component can also be used to corroborate another inferred component with respect to condition 4.

A PyElly phonetic signature is based on a kind of Soundex encoding of a name component. This is a method of approximating the pronunciation of names in English by mapping its consonants to phonological equivalence classes. That is a big mouthful, but in classical Soundex, its six equivalence classes are actually understandable:

- { B , F , **P** , V }
- { C , G , J , **K** , Q , S , X , Z }
- { D , **T** }
- { **L** }
- { **M** , N }
- { **R** }

In Soundex, all consonants of an equivalence class map into its representative letter, shown **boldface** above. All other letters are ignored, and two consecutive letters going to the same class will have only a single representative: "Brandt" becomes **BRNT**.

Soundex also prepends the first letter of a name to get the complete code b**BRNT**, but PyElly simplifies that scheme by prepending an 'a' only if the first letter is a true vowel Otherwise, no extra letter is added.

To be more phonetic, PyElly will split the biggest Soundex equivalence class so that hard-C and hard-G are together with in a new class with representative **K** and soft-C and soft-G are together in another new class with representative **S**. This complicates the mapping of consonants to equivalence classes, but is still fairly easy to implement. So "Eugene" becomes a**SN** and Garibaldi becomes **KRPLT**.

PyElly will also encode the letters 'h', 'w', or 'y' when used as semi-consonants as **H**, **W**, or **Y**. So both "Rowen" and "Rowan" become **RWM**, while "Foyer" becomes **FYR**, and "Ayer" becomes a**YR**. The three extra semi-consonant equivalence classes here allow for finer phonetic distinctions than with plain Soundex.

Finally, PyElly will transform certain letter combinations the spelling of names in English to get phonetic signatures better representing their pronunciation. For example, "Alex" becomes a**LKS**, "Eustacia" becomes **YSTS**, and "Wright" becomes **RT**.

The necessary phonetic signatures for supporting inferred name component will have to be listed in the definition file for a PyElly name table. They should appear one per line starting with an equal sign (=) to distinguish them from the listing of explicit name components and their types. For example,

```
=aSN
=KRPLT
```

After getting known or inferred name components, PyElly will string together as many as possible to make a complete name. This will be done under the following constraints:

a. Any particular name component type may occur only once, unless they are consecutive.

b. A `TTL` name component will always start the accumulation of the next name; a `GEN` will always end any name being accumulated.

c. A `CNJ` or `REL` cannot be at the end of a name.

d. There must be at least one instance of `PNM`, `SNM`, `XNM`, or `SNG`, or a `TTL` and an `INI`.

e. A name with a single component must be a `SNG` or locally known (see below).

When any complete name is accepted, all of its individual components will be remembered in a non-persistent local PyElly table. This will be kept only until the end of the current PyElly session.

Name recognition is implemented as part of PyElly entity extraction. When the ellyBase module sees a `*.n.elly` definition file to load, it will automatically put the `nameRecognition.scan` on its list of extractors with the `NAME` syntactic type. Any recognized name will then enter PyElly sentence analysis just like any other kind of entity. In particular, any longer text element found at the sentence position for a recognized name will supersede the name.

9.3.4 Compound Entities Defined by Templates (`A.t.elly`)

PyElly will let you define various kinds of multiword text entities by applying user-provided templates to sequences of tokens in an input stream. An element in a template may be specified explicitly or by referencing a class of words, either predefined by PyElly itself or listed out by a user in a rule file `A.t.elly` for an application `A`. Matching of word variants with some inflectional endings is automatically supported for English.

A template is specified as a list of literal words or word classes; for example, `%s of %c` is a three-part template. A `%` followed by a single letter like `s` or `c` indicates a word class, which can match any one word in that class. User-defined classes also allow you to set additional restrictions on capitalization sequences (see below). The predefined word classes for any PyElly application currently are

`%n`	a sequence of digits like "01234"
`%c`	a sequence of letters with at least the first letter capitalized
`%u`	a sequence of letters with no first capitalization
`%x`	an uncapitalized letter sequence, but not including certain common words like "the" and "of"
`%b`	[is , am , are , was , were , be , being , been]
`%p`	[in , of , on , for]

So, if a user-defined class `%s` has the words [`school` , `institute` , `college` , `academy`], then the template `%s of %c` will match `School of Rock`, `Academy of Sciences`, or `Colleges of Dentistry`. This capability will be helpful for more efficient parsing if an application has to recognize many entities of a similar form.

Limited recognition of inflectional stemming is built into the matching of input text with words in a user-defined class. We see this above where the regular plural `colleges` is matched by a template expecting `college`. This stemming will apply only to regular forms of `-s, -ed`, and `-ing` and will be less comprehensive than PyElly standalone inflectional stemming for English. It can, however, match `reaches` with `reach`, `dimmed` with `dim`, and `dying` with `die`.

Such simplified stemming will sometimes make a mistake. To get around any resulting bad match, you can always list out inflected forms explicitly in a template, instance by

instance. For irregular inflected forms like `shook` matched with `shake`, this will be the only option.

There is a special type of class that is specified by a template element of the form `%*xyz`. This will match an uncapitalized sequence of letters ending in some `xyz` as specified by a user. It will have no implied inflectional matching. For example, `%*ly` will match `only` or `euphemistically`. Note, however, that there is no check on the semantic sense of a final `-ly`. The `%*ly` class will match `smelly`, `unruly`, and `rally`.

A `*.t.elly` file will have two kinds of rules:

```
CLASS := WORD,WORD,WORD,WORD,WORD

TEMPLATE : SYNTAX-SPECIFICATION
```

A `CLASS` will be indicated by `%`x, where x is a single PyElly letter, possibly with diacritical marks to get us beyond only 26 possible word classes. Class rules may be anywhere in a definition file. If a template specifies any undefined class, loading of a definition file will halt with an error message. The " := " separator is mandatory in an explicitly specified class. You may specify any number of words separated by commas in a class rule.

A rule defining a template to match will have two parts separated by a mandatory " : " separator. The left part will be a list of at least two elements: words, word classes, or embedded punctation to be matched, all separated by spaces. The right part will be any PyElly syntactic specification to be assigned to text matching a template. For example,

```
put %p : VERB
%d - %d : ADJ[%cnj]
```

A word in a template must be alphanumeric only, except possibly for a completely embedded comma. Hyphens, periods, and apostrophes all have to be matched explicitly, as shown in the second example above, which will recognize `01234-56789` . Uppercase letters in a template will always be converted to lowercase. The only way you can do any uppercase matching in a template is with the `%c` word class.

To match a template, a sequence of words in input text must match every element of the template, including any implied spaces between the components. More than one template can be matched by input text, but in line with other kinds of PyElly vocabulary lookup, only the longest matches will kept.

On the whole, template rules are intended for entities of at least two components. A simple application can dispense with them entirely by relying instead on literal vocabulary tables as described below in Subsection 9.4. Templates can, however, be useful for processing text like biomedical literature, which can have quite complex nomenclature that may be hard to parse out syntactically.

9.3.5 Coding Your Own Entity Extractors in Python

You can write your own entity extraction methods in Python and add them to the extractors list for PyElly in `ellyConfiguration.py`. This should be done as follows:

1. The name of a method can be anything legal in Python for such names.
2. The method should be defined at the level of a module, outside of any Python class. This should be in a separate Python source file, which can then be imported into `ellyConfiguration.py`.
3. The method takes a single argument, a list of individual Unicode characters taken from the current text being analyzed. PyElly will prepare that list. The method may alter the list as a side effect, but you will have be careful in how you do this if you want the changes to persist after returning from the method. That is because Python always passes arguments to a method by value.
4. The method returns the count of characters found for an entity or 0 if nothing is found. The count will always be from the start of an input list after any rewriting. If no entity is at the current position input text, return 0.
5. If a non-zero character count is returned, these characters are used to generate a parse tree leaf node of a syntactic type specified in the `ellyConfiguration.py` extractors list.
6. PyElly will always apply entity extraction methods in the order that they appear in the extractors list. Note that any rewriting of input by a method will affect what a subsequent method will see. All extractor methods will be tried.
7. An extraction method will usually do additional checks beyond simple pattern matching. Otherwise, you may as well just use PyElly finite-state automatons described in Section 4.2.
8. Install a new method by editing the extractors list in the PyElly module `ellyConfiguration.py` to append a method and a syntax specification to the list. You will have to import the actual module containing your method. For example, to add a method called `yourModule.yourExtractor` to the default list of PyElly entity extractors.import yourModule

```
extractors = [   # entity extraction procedures to use
    [ extractionProcedure.date , 'date' ] ,
    [ extractionProcedure.time , 'time' ] ,
    [ yourModule.yourExtractor , 'type' ]
]
```

The module `extractionProcedure.py` defines the method `stateZIP`, which looks for a U.S. state name followed by a five- or nine-digit postal ZipCode. This will give you a model for writing your own extraction methods; it is currently not installed.

9.4 PyElly Vocabulary Tables (`A.v.elly`)

PyElly can maintain large vocabulary tables in external files created and managed with the SQLite package, a standard part of Python libraries. PyElly formerly used Berkeley Database for vocabulary, but changes in its open-source licensing made it awkward for unencumbered educational use. For more details here, please refer to Appendix C.

You can run PyElly without vocabulary tables, but these can make life easier for you even when working with only a few hundred different terms. They provide the most convenient way to handle multi-word terms and terms including punctuation. They also can be more easily reused with different grammar tables and generally will be more compact and easier to set up than `D:` rules of a grammar. Without them, PyElly will be limited to fairly simple applications.

PyElly vocabulary table entries will be defined in a `*.v.elly` definition file where each entry can be only a single text line and can have only extremely limited semantics. This is mainly so that one may generate large numbers of entries automatically through scripts. For example, the PyElly distribution file `default.v.elly` has 155,229 entries generated with bash shell scripts from WordNet 3.0 data files.

Each vocabulary entry in a PyElly `*.v.elly` definition file should be a single text line that usually takes one of the following formats:

```
TERM : SYNTAX

TERM : SYNTAX =TRANSLATION

TERM : SYNTAX x=Tx, y=Ty, z=Tz

TERM : SYNTAX (procedure)
```

A ' : ' indicates where a term ends; the spaces before and after the colon (`:`) are mandatory. Other definition formats will be described later in Subsection 10.3.2.

The `TERM : SYNTAX` part is mandatory for any vocabulary entry. A `TERM` can be

Lady Gaga

Lili St. Cyr

Larry O'Doule

"The Robe"

ribulose bisphosphate carboxylase oxygenase

No wildcards are allowed in a `TERM`, and it may start or end with a letter, digit, or a punctuation mark. Upper and lower case in a `TERM` will not matter; PyElly will always ignore case when matching up entries with input text.

An external vocabulary entry must start with a letter or digit or one of the following punctuation marks: period (.) , hyphen-minus (-), left double quotation mark (“) , comma (‘,’) , percent sign (%) , em dash (—) , and hyphen (‐). The last is Unicode and is treated differently by PyElly than the ASCII hyphen-minus character. You can change this list by editing the assignment to initChar in the source code for vocabularyTable.

In general, a vocabulary table entry may containing spaces and arbitrary punctuation. This is not allowed with internal dictionary rules or with FSA-defined patterns.

`SYNTAX` is just the usual PyElly specification of syntactic type plus optional syntactic features; for example, `VERB[^PTCL]`.

The final translation part of a vocabulary entry is optional and can take one of the forms shown above. If the translation is omitted, then the generative semantic procedure for the entry will be just the operation `OBTAIN`. This is equivalent to no translation at all.

An explicit `TRANSLATION` is a literal string to be used in rewriting a vocabulary entry; the ‘=’ is mandatory here. The `x=Tx, y=Ty, z=Tz` alternate form is a generalization of the simpler `TRANSLATION`; it maps to the generative semantic operation

```
PICK lang (x=Tx#y=Ty#z=Tz#)
```

One of the `x` or `y` or `z` in the translation options of a vocabulary entry can be the null string. In this case, the `PICK` operation will treat the corresponding translation as the default to be taken when the value of the `lang` PyElly local variable is undefined or matches none of the other specified options.

If there is no default and `lang` is undefined or not set to one of the `pick` options, then a null translation will be selected. This may sometimes be helpful, but is probably not what you really want.

A `(procedure)` in parentheses is a call to a generative semantic subprocedure defined elsewhere in a `*.g.elly` grammar rule file.

Here are some full examples of possible vocabulary table entries in a `*.v.elly` file:

```
Lady Gaga : noun =Stefani Joanne Angelina Germanotta
Lili St. Cyr : noun[:name] 0
horse : noun FR=cheval, ES=caballo, CN=馬, RU=лошадь
twerk : verb[|intrans]
```

All references to syntactic types, syntactic and semantic features, and procedures will be stored in a vocabulary table as an encoded numerical form according to a symbol table associated with a PyElly grammar.

Syntactic features must be immediately after a syntactic category name with no space in between. Otherwise, PyElly will be unable to differentiate between syntactic and

semantic features in a `*.v.elly` file. Individual syntactic or semantic features inside of brackets may be preceded by a space, however.

Unlike the dictionary definitions of words in a grammar, there are no permanent rules associated with the terms in a vocabulary table. When a term is found in a vocabulary table, PyElly automatically generates a temporary internal dictionary rule to define that term. This rule will persist only for the duration of the current sentence analysis.

Often, a vocabulary table may have overlapping entries like

```
manchester : noun [^city]
manchester united : noun [^pro,soccer,team]
```

PyElly will always take the longest matching entry consistent with the analysis of an input sentence and ignore any shorter matches. Long matches will also supersede that of shorter matches for other PyElly lookup methods.

When comparing a vocabulary entry with text, the entry must match in the text up to a nonalphanumeric delimiter. That is, the entry "jigsaw puzzle" will match the text "jigsaw puzzle?", but not "jigsaw puzzlement". A exception is made here, though, for inflectional endings and the English possessive endings -'S and -S'. This will let "jigsaw puzzle" match "jigsaw puzzles", "jigsaw puzzled", "jigsaw puzzling", "jigsaw puzzle's", and "jigsaw puzzles'". PyElly will take these endings into account automatically, sharing the code employed by PyElly template matching.

You can turn off this English inflectional analysis by using a subclass of the PyElly VocabularyTable that has its doMatchUp() method overridden appropriately. The default inflectional analysis complicates vocabulary lookup, but can be helpful in providing limited world knowledge to facilitate PyElly parsing.

An external vocabulary entry can contain arbitrary characters that PyElly normally would take to mean that a word or other token has ended and that another has started. For example, the vocabulary entry "R&B" can be matched as a whole in text even though PyElly would analyze "R&B" in text as separate "R", "&", and "B" without the vocabulary entry. This capability can be quite useful in many PyElly applications. A vocabulary entry may not match across sentences, however.

For a given application `A`, PyElly first will look for vocabulary definitions in `A.v.elly`. If this is missing, `default.v.elly` is taken, which includes most of the nouns, verbs, adjectives, and adverbs in WordNet 3.0. Always define `A.v.elly` if you do not want a huge vocabulary, which can take a long time to load. PyElly will save a compiled vocabulary data base for an application `A` in the file `A.vocabulary.elly.bin`. If you change `A.v.elly`, PyElly will automatically recompile any `A.vocabulary.elly.bin` at startup. Recompilation will also happen if `A.g.elly` has changed. Otherwise, PyElly will just read in the last saved `A.vocabulary.elly.bin`.

Note that the `A.vocabulary.elly.bin` file created by PyElly must always be paired only with the `A.rules.elly.bin` file it was created with. This is because syntactic

types and features are encoded as numbers in `*.elly.bin` files, which may be inconsistent when they are created at different times. If you want to reuse language rules, always start from the `*.*.elly` files. If PyElly has to recompile `A.rules.elly.bin` at startup, then it will also automatically recompile `A.vocabulary.elly.bin`. It will also recompile if the grammar rule file is more recent that the vocabulary rule file.

10. Logic for PyElly Cognitive Semantics

The generative semantic part of a grammar rule tells PyElly how to translate its input into output, while the cognitive semantic part evaluates the plausibility for any particular translation. Generative semantics is always the final step in PyElly processing; cognitive semantics will run each time a new phrase node is created in PyElly sentence analysis. This is to get ready to resolve any subsequent ambiguities.

With a large grammar, we cannot expect every input sentence to break down in only one way into subconstituents according to the rules of that grammar. In most languages, for example, a particular word might be assigned multiple parts of speech, and each possibility here can result in a different syntactic analysis for an input sentence. All such alternate analyses must be evaluated to find the best interpretation of a sentence.

PyElly takes a wait-and-see approach in ambiguous situations. Multiple interpretations might exist at lower levels of a parse tree, but some could end up not fitting into any final analysis for an entire sentence. In this case, an ambiguity will resolve itself within a bigger context, and so we really want to hold off any decision on interpretation until as long as possible.

PyElly looks at differing interpretations only when it comes across two or more phrase nodes of the same syntactic type with the same syntactic features over the same tokens of an input sentence, and without the `*unique` feature set. At that point, PyElly will compare the cognitive semantic plausibility scores already computed for each alternate phrase node and then go forward only with the highest ranking of them to reach a full sentence analysis.

There is only one exception to this requirement for exact matching of syntactic category and features. At the end of processing a sentence, PyElly may have two or more phrase nodes of type `SENT` over the entire sentence, but with different syntactic features. PyElly will then ignore any differing features here and just choose one interpretation according to its semantic plausibility.

PyElly will see ambiguity only when its language rules actually allow for it. For instance, "I love rock" could be about music or landscaping in normal human understanding, but if the grammar rules of a language definition fail to produce two different syntactic analyses with co-extensive phrase nodes of the same type and same features, PyElly will see only a single interpretation. If you want to find ambiguity here, you must provide PyElly with two separate vocabulary entries for "rock."

Let us consider the following simple set of grammar rules:

```
g:sent[:*r]->x

__
g:sent[:*r]->y

__
d:wwww<-x[:f]

_
  (xgen)

__
d:wwww<-y[:g]

_
  (ygen)

__
```

where `wwww` is an internal dictionary word associated with two different syntactic types `x` and `y`. A sentence consisting only of the word `wwww` will therefore be ambiguous at the lowest level of analysis because the generative semantics for the overall sentence must call either `(xgen)` or `(ygen)` as a subprocedure, but not both.

Two PyElly sentence analyses are possible here, given the rules for inheriting syntactic features through the predefined `*R` syntactic feature described in Subsection 8.1:

```
SENT[:*r,f]       SENT[:*r,g]
     |                 |
   X[:f]             Y[:g]
     |                 |
   wwww              wwww
```

There are no PyElly ambiguities here, however, because none of the constituents in the two alternate analyses of the sentence "`wwww`" have the same syntactic type and the same syntactic features. We do, however, end up with two possible parse trees for the type `SENT` at the end of processing and will then choose one of them from which to produce a translation here regardless of syntactic features.

The choice between alternate sentence analyses or between different interpretations of individual phrases will be through a numerical plausibility score assigned to each phrase node in a parse tree. A score of 0 here will be neutral, increasingly positive will be more plausible, and increasingly negative will be more implausible. Various characteristics of a phrase node can be examined to decide whether to increase or decrease its plausibility. The highest plausibility score wins.

This section of the PyElly User's Manual tells various ways to assign a score to each possible subtree in a PyElly sentence analysis from the rules from a PyElly language definition used to generate an analysis. You can choose to write language descriptions without such scoring, but plausibility is built into the PyElly parsing algorithm and provides another way to control how PyElly runs.

The PyElly plausibility score for an analyzed constituent of a sentence will always be an integer value. The score for a particular phrase node generally will add up the scores of

its immediate phrase subconstituents plus an adjustment from the cognitive semantics of the grammatical rule combining those subconstituents into one resulting phrase.

For example, suppose that we have a constituent described by a grammar rule `A->X Y`. We will expect that plausibility scores were computed already for subconstituent `X` and for subconstituent `Y` in earlier PyElly parsing. So we can then run the cognitive semantic logic for the grammar rule `A->X Y`, producing an adjustment to the summed plausibility scores for `X` and `Y` to get an overall plausibility score for our phrase of type `A`.

With competing analyses, if only one phrase has the top score, PyElly chooses it and is done. If more than one phrase has the highest score, then PyElly will arbitrarily pick one of them. When there are multiple choices and they differ by at most one between the highest score and the next highest, PyElly will also tag the rule for the phrase that was picked. This will then favor a different choice the next time a similar ambiguity arises.

10.1 The Form of Cognitive Semantic Clauses

PyElly allows us to define special logic for determining the contribution of a grammar rule to an overall plausibility score for a phrase that it describes. The logic consists of a series of clauses; each one specifying the conditions under which certain types of scoring will apply. PyElly cognitive semantics will check the clauses in sequence, the first to have all its conditions satisfied will be applied. All other clauses will be disregarded.

In a `*.g.elly` file providing grammar rules for a PyElly language description, the clauses in the cognitive semantic logic for a rule will come just after the `G:` or a `D:` line introducing the rule and will end at the first _ or __ line (see Section 4). Each clause will be a single line separated into two parts by the character sequence '>>', with its left part containing zero or more conditions for applicability and its right part specifying an action to take and an adjustment to apply. For example, here is a rule with three cognitive semantic clauses, but no explicit generative semantics:

```
G:NP->ADJ NOUN
 L[^EXTENS] R[^ABSTRACT]>>*R-
            R[^GENERIC] >>*L+
                        >>*R

__
```

The '>>' is mandatory in any clause even if there are no conditions. If one or more conditions are specified, they must all be satisfied for the action of a clause to be taken. Spaces between the conditions of the left part of a clause are optional.

The conditions here can be of three types: (1) testing the semantic features associated with the constituents to be combined by a grammar rule into a single phrase; (2) checking the starting position of the constituents, the number of characters in a resulting phrase, or the total number of tokens spanned by the constituents combined; and (3) measuring the distance between the semantic concepts associated with each constituent in the case of a 2-branch rule.

Having nothing on the left side of a clause is an always-satisfied condition, and this will always make any following clauses irrelevant. It is possible also that no listed clauses for a rule apply. In that case, a zero plausibility adjustment is assumed for a phrase. That is, just the sum of subconstituent plausibility scores will be reported for the phrase.

Each of the three types of conditions for a clause will have a distinct form (see below). You may freely mix all the different types in the left side of a single clause. The conditions will be independent and may appear in any order. Be careful, however, to avoid contradictory conditions, which can never be satisfied.

A special cognitive semantic clause allows for tracing the execution of logic in which it appears. It will have the fixed form

```
?>>?
```

The `?` condition on the left side is always False, so that right side of the clause will never be run. This clause will have the side-effect, however, of telling PyElly to identify the phrase and generating rule being processed. Here is a sample trace message:

```
tracing phrase 0 : rule= 29 with current bias=0
cog sem at clause 4 of 4
l: phrase 1 @0: type=0 syn[00 00] sem[00 00] : bia=0 use=0
r: phrase 2 @1: type=0 syn[00 00] sem[00 00] : bia=0 use=0
raw plausibility= 0
adjustment= 1 sem[00 00]
3 token(s) spanned @0
```

This specifies the phrase node where the cognitive semantic logic is being executed, the grammar rule generating the node, and the subconstituents involved. If an action is then taken subsequently for the *k*th clause out of *n* in the logic being traced, then this will be reported as in the second line. In general, this will be as follows:

```
cog sem at clause k of n
```

Whether or not the action for any clause is taken, tracing output will then show the total computed plausibility increment plus the resulting semantic features (see Subsection 10.2.3 below) for a phrase.

```
incremental scoring= -2 sem[24 00]
```

Cognitive semantic tracing will show how a PyElly parse tree is being assembled node by node. Generative semantic tracing will show the order of execution for individual semantic procedures and subprocedures after a parse tree is completely built.

A grammar rule may have cognitive semantic clauses even if it has no explicit generative semantic procedure. In this case, the listing of clauses will be terminated by a double underscore (__) line without a preceding single underscore (_).

10.2 Kinds of Cognitive Semantic Conditions

PyElly cognitive semantics mainly shows up in the grammar rule logic for evaluating the plausibility of phrase nodes in an analysis, but can appear elsewhere in PyElly as well. Four different kinds of conditions are currently supported: semantic features, starting token position and token count, and semantic concepts. Cognitive semantic logic can also have no conditions, which allows a grammar rule to have fixed scoring.

10.2.1 Fixed Scoring

The simplest and most common cognitive semantic clause will have no condition. This is always satisfied and will specify fixed positive or negative adjustment for a grammar rule when computing plausibility scores in a phrase analysis. Such clauses may take one of the following forms:

```
>>-
>>+
>>+++
>>———-
>>+5
>>-20
```

The initial + or – signs are mandatory in the scoring. A string of n +'s or –'s is equivalent to +n or -n. Here is an example of use in a grammar rule:

```
G:NP->ADJ UNKNOWN
 >>--    # cognitive  semantics disfavoring this rule by -2
_
  RIGHT  # generative semantics
  LEFT   #
__
```

If no cognitive semantic clauses are specified for a grammar rule, this is equivalent to

```
>>+0
```

a special case of fixed scoring. Note that the "+" is necessary here if you actually want to be explicit here about a zero score. This can be expressed more simply, however, by specifying no scoring increment or decrement, which is a neutral zero plausibility.

10.2.2 Starting Position, Token Count, and Character Count

Each PyElly phrase node in a parse tree records its starting token position and the number of input tokens and the number of text characters that it encompasses. A cognitive semantic scoring for a phrase can be conditional on its initial token position `p`

or its total token count `n` or its character count `c` to some reference value This will happen on the left side of the >> in a clause. For example,

```
p<1 >> -1
n>1>>+1
n<8>>+1
c>4>>-1
n>1n<8  >> +2
n>2 n<7 >> +1
```

A test of starting token position is with `p<` or `p>`, of token count with `n<` or n>. and character count with `c<` or `c>`. Note that the first position in a sentence will be 0. In the last clause above, a phrase will be scored as `+1` only if its token count is 7 > `n` > 2. You may insert spaces in any clause for readability, but no spaces are allowed before or after a < or a > in any comparison.

Normally a leaf node in a parse tree will have a token count of 1, but a leaf node with the `...` syntactic type may have a count of 0. Note also that a multiword vocabulary table entry like `flash flood` is always counted as a single token. Character count will include the embedded spaces in multiword vocabulary, but will count no text spaces between separate tokens.

10.2.3 Semantic Features

Semantic features are similar to syntactic features as defined above in Section 8, but play no role in distinguishing between different grammar rules. They are assigned to particular phase nodes and are specified in the same bracketed notation as syntactic features; for example:

```
[&ANIMATE,ARMORED]
```

The `&` is the feature set identifier, and `ANIMATE` and `ARMORED` are two specific features. Semantic features will have completely separate lookup tables from syntactic features. In particular, a syntactic feature set and a semantic feature set can have the same set identifier without any conflict, but always make the identifiers different just for clarity.

As with syntactic features, you may have up to 16 semantic feature names, with the rules for legal names being the same. Semantic feature names must include only ASCII letters or digits, with the case of letters not mattering, just as with syntactic feature names.

Unlike syntactic features, however, they will have only one predefined feature name: `*CAPITAL` or equivalently `*C`, which indicates that a phrase is capitalized and so is probably a name or a proper noun. This name is reserved in every semantic feature set.

10.2.3.1 Semantic Features in Cognitive Semantic Clauses

A cognitive semantic clause for a 2-branch splitting `G:` grammar rule will have the following general form when semantic features appear:

```
L[oLF1,...,LFn] R[oRF1,...,RFn]>>x[oF1,...,Fn]#
```

Semantic features can appear on both the left and the right side of a clause. The symbol "`o`" is a feature set identifier; "`x`" may be either `*L` or `*R`, and the "`#`" is a fixed scoring action as in Subsection 10.1 above; for example, `+++` or `-3`.

Here is an example:

```
l[!coord]r[$*c]>>*r-2
```

This tells PyElly that a phrase with a left part marked as `coord` and a capitalized right part inherit the semantic features from the right part and will lose two points from its semantic plausibility score.

As with syntactic features, a semantic feature `F` can be preceded by a '`-`' in a clause. On the left side, this means that feature `F` must not be associated with a matching phrase structure. On the right side, this means that any inherited `F` must be turned off in the new phrase being created for a grammar rule.

The prefixes `L` and `R` on the left side of a clause specify the constituent substructures to be tested, respectively left and right descendant. You may test none, one, or both descendants in a 2-branch rule; this is the most common kind of condition when you want something different from fixed scoring.

The "`x`" prefix on the right is optional for specifying inheritance of features. An `*L` means to copy the semantic features of the left subconstituent into the current phrase; `*R` means to copy the right. You cannot have both; a missing "`x`" means no inheritance at all. Any explicit semantic feature appearing in the right part of the clause will indicate any additional features to turn on or off for a phrase node.

A full cognitive semantic clause for a 1-branch extending `G:` grammar rule will have the following general form in its semantic features:

```
L[oLF1,...,LFn]>>x[oF1,...,Fn]#
```

Here is an example:

```
L[^animate]>>*L[^actor]+1
```

A `D:` grammar rule defines a phrase without any constituent substructures. The semantic features in a clause must take the form

```
>>[oF1,...,Fn]#
```

That is, you can set semantic features for a `D:` rule, but may not test or inherit any. Here is an example

```
>>[^animate]+1
```

For both splitting and extending grammar rules, any of the left side of a cognitive semantic clause can be omitted. If all are omitted here, then a clause always applies. It becomes a case of fixed scoring.

10.2.3.2 Semantic Features in Generative Semantics

PyElly also allows a generative semantic procedure to look at the semantic features for the phrase node to which it is attached. This is done in a special form of the IF command where the testing of a local variable is replaced by the checking of semantic features as done in cognitive semantics. For example,

```
IF [&F1,-F2,F4]
  (do-SOMETHING)
  END
```

The testing here is like that on syntactic features to determine the applicability of a 1- or 2-branch grammar rule in PyElly parsing. The `IF` here cannot be negated with `~`. If you want negation, you have to specify it for the individual features.

10.2.4 Semantic Concepts

PyElly is a currently being used experimentally in applying conceptual information from WordNet to infer the intended sense of ambiguous words in English text. This now an option for cognitive semantics in PyElly and may appear on both the left and right sides of a clause attached to a grammar rule and in other places.

PyElly allows you to establish a set of concepts each identified by a unique alphanumeric string and related to one other by a conceptual hierarchy defined in the language description for an application. This can be done any way that you want, but WordNet provides a good starting point here since it contains over two hundred thousand different related synonym sets (or synsets) as potential concepts to work with.

(WordNet is produced manually by professional lexicographers affiliated with the Cognitive Science Laboratory at Princeton University and is an evolving linguistic resource now at version 3.1. [George A. Miller (1995). WordNet: A Lexical Database for

English. Communications of the ACM Vol. 38, No. 11: 39-41.] This notice is required by the WordNet license.)

In WordNet, each possible dictionary sense of a term will be represented as a set of synonyms (synset) in a given language. This can be uniquely identifiable as an offset into one of four WordNet data files associated with the main parts of speech—`data.noun`, `data.verb`, `data.adj`, and `data.adv`.

For disambiguation experiments in PyElly, trying to work with all the synsets of WordNet is too cumbersome. So we instead focused on concepts from a small subset of WordNet synsets related to interesting kinds of ambiguity in English. We can identify each such concept as an 8-digit decimal string combining the unique WordNet offset for its corresponding synset plus a single appended letter to indicate its part of speech. For example,

```
13903468n : (=STAR) a plane figure with 5 or more points; often used as an emblem

01218092a : (=LOW) used of sounds and voices; low in pitch or frequency
```

The standard WordNet letter abbreviation for a part of speech is `n` = noun, `v` = verb, `a` = adjective, `r` = adverb.

For any set of such concepts, we can then map selected semantic relations for them from WordNet into a simple PyElly conceptual hierarchy structure, which will be laid out in a PyElly language definition file `A.h.elly`. The current `disambig` example application in the PyElly package has a hierarchy with over 800 such related concepts, all taken from WordNet 3.1.

You can of course also define your own hierarchy of concepts with their special hierarchy of semantic relations. The only restriction here is that each concept name must be an alphanumeric string like aAA0123bcdef00. Upper and lower case will be ignored in letters. Such semantic concepts can be explicitly employed by cognitive semantic clauses on their left side and can be explicitly employed in one way and implicitly employed in two ways on the right side.

10.2.4.1 Concepts in Cognitive Semantic Logic

Semantic concepts serve to provide another aspect to consider when computing plausibility scores to choose between alternate interpretations in case of ambiguity. This will happen in the cognitive semantic logic associated with each syntax rule, and the concepts will have different roles when they are on the left and on the right sides of a cognitive semantic clause.

10.2.4.1.1 Concepts on the Left Side of Cognitive Semantic Clauses

The left half of a clause is for testing its applicability to a particular phrase, and PyElly allows the semantic concepts associated with its subconstituents to be checked out. The syntax here is similar to how you test semantic features of subconstituents, except you will use parentheses `( )` to enclose a concept name instead of the `[ ]` around semantic features. Here is an example of a concept check:

```
L(01218092a) R(13903468n) >>+
```

This checks whether the left subconstituent of a phrase has a concept on a path down from concept `01218092a` in a conceptual hierarchy and whether the right subconstituent has a concept on a path down from concept `13903468n`. The ordering of testing here does not matter, and you may omit either the `L` or the `R` test or both.

You can mix concept testing with semantic feature testing in the conditional part of a cognitive semantic clause. For example,

```
L(01218092a) L[^PERSON] >>++
```

You may also specify more than one concept per test. For example,

```
L(00033319n,08586507n) >>+
```

Here, PyElly will check for either `L(00033319n)` or `L(08586507n)`.

You of course can define more self-descriptive concept names for your own application. You are not limited to WordNet 3.1 synset ID's.

10.2.4.1.2 Concepts on the Right Side of Cognitive Semantic Clauses

A single concept can be explicitly appended on the right side of a clause with a separating space. For example,

```
>>++ CONCEPT
```

This must always come after a plausibility scoring expression. If you want a neutral scoring here, you must specify it explicitly here as

```
>>+0 CONCEPT
```

Normally, this kind of concept reference will be useful only for the cognitive semantics of `D:` dictionary rules of a grammar, but nothing keeps you from trying it out in `G:` rules as well.

Concepts can also be referenced implicitly of the right side of a clause. When a subconstituent of a phrase has an associated concept, the `*L` or `*R` inheritance actions specified by a clause will apply to concepts as well. So, a clause like

```
>> *L++
```

will cause not only the inheritance of semantic features from a left subconstituent, but also the inheritance of any semantic concept from that left subconstituent. That is also true for `*R` with a right subconstituent.

To use semantic concepts on the right side of a clause, you generally must use the `*L` or `*R` mechanism even if you have no semantic features defined. This must be done to pass concepts in a parse tree for later checking. Note that you cannot have both `*L` and `*R` in a cognitive semantic clause.

Semantic concepts also implicitly come into play in two ways when PyElly is computing a plausibility score for a phrase:

1. When a subconstituent has a semantic concept specified, PyElly will check whether it is on a downward path from a concept previously seen in the current or an earlier sentence. PyElly will maintain a record of such previous concepts to check against. If such a path is found, the plausibility score of a phrase will be incremented by one. If a phrase has one subconstituent, the total increment possible here is 0 or 1; if the phrase has two, the total increment could be 0, 1, or 2.

2. If a phrase has two subconstituents with semantic concepts, PyElly will compute a semantic distance between their two concepts in our inverted tree by following the upward paths for each concept until they intersect. The distance here will be the number of levels from the top of the tree to the point of intersection. If the intersection is at the very top, then the distance will be zero. The lower the intersection in the tree, the higher the semantic relatedness. This distance will be added to the plausibility score of a phrase containing the two subconstituents.

If no semantic concepts are specified in the subconstituents of a phrase, then a semantic plausibility score will be computed exactly as before.

10.2.4.2 Language Definition Files for Semantic Concepts

To use semantic concepts, you must define them in a PyElly language definition. For an application `A`, this must happen in the files `A.h.elly`, `A.g.elly`, or `A.v.elly`. They can be omitted entirely if you have no interest in them.

10.2.4.2.1 Conceptual Hierarchy Definition (A.h.elly)

This specifies all the concepts in a language definition and their semantic relationships. You can define everything arbitrarily, but to ensure consistency, start from some

existing language database like WordNet. Here are some entries from `disambig.h.elly`, a conceptual hierarchy definition file based on WordNet 3.1 concepts for a PyElly example application:

```
14831008n > 14842408n
14610438n > 14610949n
00033914n > 13597304n
05274844n > 05274710n
07311046n > 07426451n
07665463n > 07666058n
04345456n > 02818735n
03319968n > 03182015n
08639173n > 08642231n
00431125n > 00507565n
```

The ">" separates two concept names to be interpreted as a link in a conceptual hierarchy, where the left concept is the parent and the right concept is a child. In this particular definition file, each concept name is an offset in a WordNet 3.1 part of speech data file plus a single letter indicating which part of speech (`n`, `v`, `a`, `r`). Both offset and part of speech are necessary to identify any WordNet concept uniquely.

For convenience, a `*.h.elly` file may also have entries of the form

```
=xxxx yyyy
=zzzz wwww
```

These let you to define equivalences of concept names, where the right concept becomes the same as the left. For example, the entries make `yyyy` to be the same as `xxxx` and `wwww` to be the same as `zzzz`. The left concept must occur elsewhere in the hierarchy definition, though. An equivalence can be specified anywhere in a `*.h.elly` file. It specifies only a convenient alias for a concept name without defining a new concept, which can be helpful in documentation of semantic relationships.

10.2.4.2.2 Semantic Concepts in Grammar Rules (A.g.elly)

This was already discussed briefly in Subsection 10.3.2 above. Here is an example of a grammar dictionary rule with cognitive semantics referencing semantic concepts:

```
D:xxxx <- NOUN
 >>+0 CONCEPT

_
  APPEND xxxx-C

__
```

Similarly with a regular syntax rule:

```
G:X -> Y Z
 >>*L+0 CONCEPT

_
  RIGHT
  SPACE
  LEFT

__
```

Note here that the `*L` action will also cause any concept associated with `Y` to be inherited by `X`, but the explicit assignment of `CONCEPT` here will always override any such inheritance.

10.3 Adding Cognitive Semantics to Other PyElly Tables

Earlier sections described various PyElly language definition tables, but without mention of cognitive semantics, since they were as yet undefined. Because cognitive semantics helps in resolving ambiguity, however, we now need to take care of that omission so that we can better define PyElly language elements.

Currently, vocabulary tables, pattern tables, template tables, and entity extractors all optionally allow fixed plausibility scoring and semantic features in definitions. You can also specify semantic concepts in vocabulary tables; but elsewhere, you can bring them in only though special grammatical rules for the syntactic types associated with them.

10.3.1 Cognitive Semantics for Vocabulary Tables

Vocabulary table rules can include all basic cognitive semantics, but no logic. This is because vocabulary can appear only at the bottom of a PyElly parse tree.

10.3.1.1 Semantic Features in Vocabulary Table Entries (A.v.elly)

In addition to the rule forms listed in Subsection 9.4, PyElly also recognizes

```
TERM : SYNTAX SEMANTIC-FEATURES PLAUSIBILITY

TERM : SYNTAX SEMANTIC-FEATURES PLAUSIBILITY =TRANSLATION

TERM : SYNTAX SEMANTIC-FEATURES PLAUSIBILITY x=Tx, y=Ty, z=Tz

TERM : SYNTAX SEMANTIC-FEATURES PLAUSIBILITY (procedure)
```

`SEMANTIC-FEATURES` are the bracketed semantic features for cognitive semantics (see Subsection 10.2); it can be "`0`" or "`-`" if no features are set. `PLAUSIBILITY` is an integer value for scoring a phrase formed from a vocabulary entry; this value may include an attached semantic concept name separated by a "`/`" (see Subsection 10.3). Both

SEMANTIC-FEATURES and PLAUSIBILITY may be omitted, but if either is present, then the other must be specified also.

Here are some expansions of vocabulary definitions from Subsection 9.4:

```
Lady Gaga : noun [^celeb] +2 =Stefani Joanne Angelina Germanotta
Lili St. Cyr : noun[:name] [^celeb] 0
horse : noun [$animate] 0 FR=cheval, ES=caballo, CN=馬, RU=лошадь
twerk : verb[|intrans] [^sexy] -1 (xxxx)
```

Semantic features are helpful in distinguishing words with multiple senses as multiple vocabulary table entries; for example,

```
bank : noun [^institution] 0
bank : noun [^geology] 0
bank : verb [|intrans] - -2
bank : verb [|trans] 0
```

If the word BANK shows up in input text, then all of these entries will be tried out in possible PyElly analyses, with the most plausible taken for final PyElly output. Try to give higher plausibility to the part of speech that is most likely.

10.3.1.2 Semantic Concepts in Vocabulary Table Entries (A.v.elly)

For a vocabulary table entry, we extend the plausibility field in an A.v.elly input file to allow appending a concept name separated by a "/" (See Subsection 9.4 above) with no spaces. Omitting a concept name here will be equivalent to a null concept.

Here are some entries from disambig.v.elly, a vocabulary table definition file making use of the concepts above.

```
finances : noun[:*unique] - 0/13377127n =funds0n
monetary resource : noun[:*unique] - 0/13377127n =funds0n
cash in hand : noun[:*unique] - 0/13377127n =funds0n
pecuniary resource : noun[:*unique] - 0/13377127n =funds0n
assets : noun[:*unique] - 0/13350663n =assets0n
reaction : noun[:*unique] - 0/00860679n =reaction0n
response : noun[:*unique] - 0/00860679n =reaction0n
covering : noun[:*unique] - 0/09280855n =covering0n
natural covering : noun[:*unique] - 0/09280855n =covering0n
cover : noun[:*unique] - 0/09280855n =covering0n
```

The *UNIQUE syntactic feature in each entry is to disable PyElly ambiguity resolution at lower levels of sentence analysis, a requirement for the disambig example application. The translation provided for each entry above is the WordNet sense ID for a particular word sense, which ends with a single letter specifying its part of speech.

You may name the concepts in your own `A.h.elly` hierarchy definitions however you wish, but with two exceptions: the name "`-`" will be reserved to denote a null concept explicitly in grammar rules; and the name "`^`" will be reserved for the top of a hierarchy to which every other concept is linked eventually. You must have "^" somewhere in a `A.h.elly` hierarchy definition file for it to be accepted by `vocabularyTable.py` as a language definition file.

10.3.2 Cognitive Semantics for Pattern Rules

PyElly pattern rules for determining the syntactic type of an arbitrary string were first presented in Section 4.2. With the definition of semantic features and plausibility scores here, we can now also talk about how pattern rules can specify them. This can be done by inserting one or two extra fields into a pattern rule specification. The four-part format defined in Section 4.2 is still valid.

```
STATE    PATTERN    SYNTACTIC TYPE    NEXT
```

PyElly will allow two other formats as well.

```
STATE    PATTERN    SYNTACTIC TYPE  SEMANTIC FEATURES  NEXT

STATE    PATTERN    SYNTACTIC TYPE  SEMANTIC FEATURES  SCORE  NEXT
```

a. The `STATE`, `PATTERN`, `SYNTACTIC TYPE`, and `NEXT` are the same as before.
b. `SEMANTIC FEATURES` is a bracketed list of feature names as seen in a vocabulary rule or in the cognitive semantics of a grammar rule. These features are optional and may be set or left unset.
c. `SCORE` is a possibly signed integer value to be assigned as an initial plausibility for a token matching a pattern in a final state.
d. If a `SCORE` is specified in a pattern rule, then the `SEMANTIC FEATURES` must also be present. As in the case of a vocabulary entry; a simple – place holder can be specified here to indicate no setting of features.

Here some examples of some pattern rules with cognitive semantics:

```
0 &#@$ XID [!nom] -1
3 \#&#$ ORD - 2 -1
3 \#&#@#$ ORD - -1 -1
11 &@mab$ PHRM[:sgl] [$cancer] 1 -1
```

Note that PyElly allows basic cognitive semantics only at a final state of a pattern-matching automaton. A final state is always indicated with a `-1` as the next state of a pattern rule, which will always be the last part of a rule. An error will be flagged for cognitive semantics if there is a next state.

Assigning a base cognitive semantic score allows you to favor or disfavor a syntactic type assigned to a token by pattern matching versus by another possibility like lookup in an internal dictionary or in an external vocabulary table. Semantic features allow pattern-

matched interpretations of a token to be tested by the cognitive semantic logic in rules when ambiguity resolution is required.

10.3.3 Cognitive Semantics for Template Matching

Cognitive semantics for template matching will be the same as for vocabulary table entries. After the syntax specification for a template rule, you can also optionally add semantic features and a plausibility score. The full form of a template rule is

```
TEMPLATE  :  SYNTAX-SPECIFICATION SEMANTIC-FEATURES PLAUSIBILITY
```

If a rules specifies semantic features, it must also specify a plausibility; for example

```
%s of %c : NOUN[:*l] [&cmpd] -1
```

The semantic features can be just a hyphen (–), if none are to be set. This is the same as with vocabulary table entries. A semantic concept name preceded by / can immediately follow the `PLAUSIBILITY` as with a vocabulary table rule.

10.3.4 Cognitive Semantics for Entity Extraction

You can define the cognitive semantics for an entity extractor listed in the PyElly module `ellyConfiguration.py`. This can be done by adding one or two optional elements in a list entry for an extractor to specify semantic features or a plausibility scoring. For example, we can rewrite the listing of Subsection 9.3.2 as

```
import extractionProcedure

extractors = [  # list out extraction procedures
  [ extractionProcedure.date , 'date' , '-' , 0 ] ,
  [ extractionProcedure.time , 'time'] , '[^x]' ]
]
```

Note that semantic features must be specified as a bracketed string in a third list element and a plausibility scoring must be specified as an integer in a fourth list argument. Semantic features may not be omitted if a plausibility scoring is present; but these can be specified as a non-committal '–' as done in vocabulary table entries.

Currently, semantic concepts may not be specified for any entity recognized by procedural extractor. If you need an associated concept for such an entity, then you have to assign one via the cognitive semantics for an X->Y grammatical rule. Entities in general will not be found in resources like WordNet.

Name recognition operates the same way as entity extraction. When turned on by a command line flag, PyElly will insert the method `nameRecognition.scan` into the `extractors` array in the PyElly module `ellyConfiguration.py`, but without specifying any cognitive semantics.

This is because personal names are strictly denotative, which means they are purely identifiers and typically do not convey any meaning that could be captured in a concept. This is unlike other entities, which can be both denotative and connotative like INTERNATIONAL MONETARY FUND and can be treated conceptually.

If you absolutely must have cognitive semantics for name recognition, you can also just use the trick mentioned above to define a grammar rule X->Y with the cognitive semantics to define both semantic features and semantic concepts. This is a highly unlikely eventuality.

11. Sentences and Punctuation

Formal grammars typically describe the structure only of single sentences in a language. PyElly accordingly is set up to analyze one sentence at a time through its ellyBase module. In real-world text, however, sentences are all jumbled together, and we have to divide them up properly before doing anything with them. That task is harder than one might think; for example,

```
I met Mr. J. Smith at 10 p.m. in St. Louis.
```

This sentence contains six periods (`.`), but only the final one should stop the sentence. It is not hard to recognize non-stopping exceptions, but this is yet one more detail to take care of on top of the many other basic tasks of natural language processing.

Furthermore, we have to deal with special instances of punctuation like

```
He said "No."
(Turn to page 6.)
```

where punctuation after the stop (`.`) really should be tacked onto the sentence that it ends. PyElly is prebuilt to handle all of this.

PyElly divides text input into sentences with its ellySentenceReader module, which looks for various patterns of punctuation to detect sentence boundaries in text. While doing this, PyElly also normalizes each sentence to make it easier to process. The algorithm is simple and tends to find too many sentences, but we can always help PyElly out with various special-case logic to give it more smarts .

Currently, the PyElly stopException module lets a user provide a list of patterns to determine whether a particular instance of punctuation like a period (`.`) should actually stop a sentence. The PyElly exoticPunctuation module tries to normalize various kinds of unorthodox punctuation found in informal text. This solution is imperfect, but we can extend or modify it as needed. See Subsection 11.2 below for details.

The approach of PyElly here is to provide sentence recognition a notch or two better than what one can cobble together just using Python regular expressions or the standard sentence recognition methods provided by libraries in a language like Java. If you really need more than this, then there are other resources available; for example, NLTK can be trained on sample data to discover its own stop exceptions. Builtin PyElly sentence recognition should be adequate for most applications, however.

PyElly sentence reading currently operates as a pipeline configured as follows:

```
raw text => ellyCharInputStream => ellySentenceReader => ellyBase
```

where raw text is an input stream of Unicode encoded as UTF-8 and read in line by line with the Python `readline()` method. The ellyCharInputStream module is a filter that removes extra white space, substitutes for Unicode characters not recognized by PyElly,

converts hyphens used as dashes when appropriate, and replaces single new line characters with spaces. The ellyCharInputStream and ellySentenceReader modules both operate at the character level and together will divide input text into individual sentences for subsequent PyElly processing.

A single input line could contain multiple sentences, or a single sentence may extend across multiple input lines. There is also no limit on how long an input line may be; it could be an entire paragraph terminated by a final linefeed as found in many word processing files. PyElly can also read text divided into short lines terminated by linefeeds, carriage returns, or carriage returns plus linefeeds. It currently will not splice back a hyphenated word split across two lines, however.

The ellySentenceReader module currently recognizes five kinds of sentence stopping punctuation: period (`.`), exclamation point (`!`), question mark (`?`), colon (`:`), and semicolon (`;`). By default, any of these followed by whitespace except for a Unicode thin space will indicate the end of a sentence. A blank line consisting of two new line characters together will imply the end of a sentence without any final punctuation.

The ellyMain module, the standard top-level module for PyElly, employs ellySentenceReader. You can run ellyMain interactively from a keyboard, but since it expects general text input, you may have to add an extra `<RETURN>` to get the module to recognize the end of a sentence and start processing.

11.1 Basic PyElly Punctuation in Grammars

The PyElly punctuationRecognizer module automatically defines a small set of single Unicode characters as punctuation for text. These include the stop punctuation already recognized by ellySentenceReader, plus comma (`,`) , bracketing and parentheses (`[]`), apostrophe (`'`), and double-quote (`"`) in ASCII, and a few non-ASCII Unicode characters like (‟) and (″) seen in formatted text. These are defined in the Python source file `punctuationRecognizer.py`.

The punctuationRecognizer module is a builtin extension of the grammar rules in a `X.g.elly` definition file for an PyElly application `X`. It has the effect of automatically creating default internal dictionary entries for single-character punctuation in every PyElly application. This is currently biased toward English, but can be adapted for other languages by changing the basic punctuationRecognizer table and recompiling or by adding explicit internal dictionary rules to override the table.

The punctuationRecognizer module can be replaced in PyElly by a stub with an empty table and a `match()` method that always returns `False`. In that case, you will have to supply all your own punctuation rules, but most of the time, you can just take the PyElly defaults. This was the approach in all twelve of the functioning example applications in the current PyElly distribution package.

All predefined PyElly single- and multi-character punctuation will be associated with the syntactic type `PUNC`. If you want to make your own system of punctuation, define

your own syntactic types here and then use them in your grammar and vocabulary rules. You can even reuse `PUNC`, but remember that this will come with prior definitions.

PyElly also will qualify the syntactic type `PUNC` with syntactic features under the ID `[| ]` and semantic features under the ID `[! ]`. These are pre-defined in the Python code of PyElly and can be changed only there. Their names currently are as follows:

syntactic feature	Indication
start	can start a sentence
stop	can stop a sentence
***l**	is a left bracket or quotation mark (special use of predefined feature name)
***r**	is a right bracket or quotation mark (special use of predefined feature name)
quo	is quotation mark
com	is comma
hyph	is hyphen
emb	can be included within a PyElly token
***x**	indicates a square bracket when occurring with *L or *R, a period when occurring with STOP, or an m dash otherwise (special use of predefined feature name)

semantic feature	Indication
brk	can divide a sentence without ending it

You can add your own features under the `[| ]` or `[! ]` IDs, but will have to give them new unique names and assign them only to feature bits that are still free. It is easy to get into trouble when you do not know exactly what you are doing.

Remember that punctuation, like all other input text elements, may be translated by PyElly into something else in its output, but may be kept unchanged. You will have to decide on the proper action and lay out the necessary rules for PyElly.

Including the full punctuationRecognizer module in PyElly is equivalent to putting the following internal dictionary rules into every `A.g.elly` language definition file:

```
d:[  <- PUNC[|*l,*x,start]

__
d:]  <- PUNC[|*r,*x]

__
d:(  <- PUNC[|*l,start]

__
d:)  <- PUNC[|*r]
 >>[!spcs]

__
d:"  <- PUNC[|*l,quo,start]

__
d:”  <- PUNC[|*r,quo]

__
d:"  <- PUNC[|*l,*r,quo,start]

__
d:‘  <- PUNC[|*l,quo,start]

__
d:’  <- PUNC[|*r,quo]

__
d:`  <- PUNC[|*l,quo,start]

__
d:'  <- PUNC[|*l,*r,quo,start]

__
d:,  <- PUNC[|com]
 >>[!brk]

__
d:.  <- PUNC[|stop,emb,*x]

__
d:!  <- PUNC[|stop,emb]

__
d:?  <- PUNC[|stop,emb]

__
d::  <- PUNC[|stop,emb]

__
d:;  <- PUNC[|stop]

__
d:…  <- PUNC           # horizontal ellipsis

__
d:™  <- PUNC           # TM

__
d:–  <- PUNC           # en dash

__
d:—  <- PUNC[|*x]      # em dash

__
d:-  <- PUNC[|hyph]    # hyphen or minus

__
```

All PyElly predefined punctuation will translate into itself with neutral cognitive semantic plausibility. You can override this action by defining a vocabulary rule with a different rewriting for a specific punctuation, but if this has the same syntactic features as a default rule, make sure that the new rule has a positive semantic plausibility increment so that PyElly will always choose it instead of the default.

You can also use the ellyCharInputStream module to change the form of punctuation in text before it is looked up. This is how PyElly now handles ellipsis written as three dots.

11.2 Extending Stop Punctuation Recognition

The division of text into sentences by ellySentenceReader can currently be modified in two ways: by the stopException module that recognizes special cases when certain punctuation should not terminate a sentence and by the exoticPunctuation module that checks for cases where sentence punctuation can be more than a single character.

11.2.1 Stop Punctuation Exceptions (`A.sx.elly`)

When PyElly starts up an application `A`, its stopException module will try to read in a file called `A.sx.elly`, or failing that, `default.sx.elly`. This file specifies various patterns for when a regular stop punctuation character (`.` `!` `?` `:` `;`) should not terminate a sentence. These patterns will only be checked when the punctuation is followed by a space character in input text.

A pattern in a `*.sx.elly` file must each be expressed in the following form:

```
l...lp|r
```

where `p` is the punctuation character for the exception, `l...l` is a sequence of literal characters or wildcards for the immediate left context of `p`, and `r` is a single literal character or wildcard for the immediate right context of `p` plus the space after it. The vertical bar (`|`) marks the start of a right context and must be present.

The `l` and `r` parts of a pattern may have only certain Elly wildcards:

- `@` matches a single letter
- `#` matches a single digit
- `~` matches a single nonalphanumeric character
- `!` matches an uppercase letter (exclamation point)
- `¡` matches a lowercase letter (inverted exclamation point)

You may also have `*` wildcard at the right end of a left context, as long there is something else to match. The right context of an exception pattern may be omitted.

Here are some examples of actual exception patterns from `default.sx.elly`:

```
@.|
!*:|!
dr.|
mr.|
mrs.|
u.s.s.|
```

The first pattern above picks up initials, which consist of a single letter followed by a period and a space character. The second lets a sentence continue past a colon (`:`) and a space when this is preceded by a capitalized word, indicated by `!*`, and followed by a capital letter, indicated by the `!` wildcard alone. The other patterns match formal titles for names and work as you expect them to. The file `default.sx.elly` has an extensive list of stop exceptions that might be helpful for handling typical text. You may enlarge this or make up your own list for an application.

As with PyElly patterns elsewhere, lowercase letters in a pattern will match the same letter in text irrespective of case. So the third pattern above will match `DR.`, `Dr.`, `dR.`, and `dr.`. If the exception pattern were `Dr.|`, then it would match only `DR.` and `Dr.`. Uppercase in a pattern must match uppercase in text.

Note also that ordering makes a difference in the listing of patterns here. PyElly will always take the first match from a listing, which should do the right thing. The problem is when a longer and a shorter pattern can both match input text. You probably would want to list the longer pattern first, or otherwise it may never be matched.

PyElly will apply some additional conditions before accepting a match of the left context pattern of stop exception rule. It will limit the range of a left context to the longest sequence of characters preceding a candidate stop punctuation that can appear in a simple token. Rule patterns cannot refer to anything outside that range. In particular, a space wildcard can never be matched.

If a left context pattern ends with a `*` wildcard, then will be compared without the `*` wildcard against the starting characters of the actual left context in input text. If this matches, then any extra characters in the left context will have to be alphanumeric in order to make a full pattern match.

If a left context pattern has no `*` wildcard, then it will be compared against the ending characters in the actual input left context. If this matches, then if the first element of matching input text is alphanumeric, then any preceding input character may not be alphanumeric or ampersand (`&`). For example, with the rule `dr.|`, the input text `XDR.` will match the left context pattern by itself, but PyElly will reject this because of the `X`.

The handling of right context patterns will examine only one text character at most, but this will be the first character after the candidate stop punctuation and its following space. The `*` wildcard is forbidden in a right context pattern; and there must be an actual input text character here if a right context pattern is specified.

As of PyElly release v1.3.23, the stopException module includes hardwired logic to bypass the stop exception table in a few special cases. This currently is only to handle the abbreviations A.M. and P.M. in time expressions, which requires going beyond the simple pattern matching in the stop exception table. The hardwired logic cannot be overridden except by changing the Python code in the class method nomatch().

11.2.2 Bracketing Punctuation

Some punctuation can show up in complementary pairs to bracket segments of text. The most common of these are parentheses and brackets like `(` and `)` or `[` and `]` and quotation marks like ‟ and ″. The PyElly ellySentenceReader module recognizes such paired punctuation and will adjust its sentence boundaries accordingly. For example, the text segment

```
(He walks along.)
```

should be a sentence even though the period (`.`) here is not followed by a space. The closing right parenthesis will also be included in identified sentence.

Paired quotation marks are handled in the same way, but there is an added complication here because some text may use the same character for left and right quotation marks. For example,

```
"He walks along."
```

To handle this situation, PyElly automatically interprets " at the beginning of a sentence or preceded by a space as ‟ and at the end of a sentence or followed by a space as ″. It does not replace the original character, however. You can do the replacement yourself with macro substitutions if you really want to.

Within paired bracketing of any type, colons (`:`) and semicolons (`;`) will not be stopping punctuation and are treated more like a comma (`,`). The other usual stopping punctuation (`.?!`) will also be ignored it if there are fewer than 3 space characters seen so far within the brackets, not counting the spaces after a previously ignored (`.?!`) punctuation marks. This is to avoid highly fragmented sentence analysis.

PyElly currently puts a 80-character limit on any bracketing with respect to sentence boundaries. If a matching character pair like (“”) is farther apart than that, then no bracketing is recognized. You can change this limit by editing the parameter `NLM` in `ellyCharInputStream.py`.

The bracketing logic in PyElly is heuristic and may have to be tuned for a particular application. No information should be lost here, however. The result will just be to divide text input into a different set of sentences, which may or may not matter. For example, a long quotation in input text may be broken up instead being taking as a single segment of text for analysis.

11.2.3 Exotic Punctuation

This is for dealing with punctuation like `!!!` or `!?`. The capability is coded into the Pyelly exoticPunctuation module, and its behavior cannot be modified except by changing the Python logic of the module. This change will be easy, though.

The basic procedure here is to look for contiguous sequences of certain punctuation characters in an input stream. These are then automatically collapsed into a single character to be passed on to the ellySentenceReader module. The main ellyBase part of PyElly should therefore always see only standard punctuation.

11.3 How Punctuation Affects Parsing

Typical input sentences processed by PyElly may currently include all kinds of punctuation, including those recognized by stopException as not breaking a sentence. When PyElly breaks a sentence into parts for analysis, a single punctuation character by default will be taken as a token. PyElly will assign common English punctuation to the predefined syntactic type `PUNC` unless you provide vocabulary table rules or `D:` grammar rules or FSA pattern rules specifying otherwise.

For example, you might put `DR.` into your vocabulary table, perhaps as the syntactic type `TITLE`. Since this will take three characters from an input stream, including the period, PyElly will no longer see the punctuation here. PyElly tokenization will always take longest possible match when multiple PyElly rules can apply; a token including punctuation like quotation marks will probably be longer than anything else.

Identifying punctuation in an input sentence is just the start of PyElly analysis, however. The grammar rules for a PyElly application will then have to describe how to fit the punctuation into the overall analysis of a sentence and how eventually to translate it. This will be entirely your responsibility; and it can get complicated.

In simple text processing applications taking only a sentence at a time and only a small amount of text, you might choose just to ignore all punctuation by making them disappear in macro substitutions, but more usually, punctuation occurrences in sentence will provide important clues about the boundaries of phrases in text input that can greatly help in keeping your syntactic analyses manageable.

When you choose to work with sentence punctuation, you will need at least one grammar rule like

```
g:SENT->SENT PUNC[|STOP]
__
```

for handling stop punctuation terminating a sentence, although the syntactic feature reference is often unnecessary. The setting of syntactic features by the PyElly punctuationRecognizer module will have no on sentence analysis when these are unreferenced by grammar rules.

PyElly parsing will fail if any part of a sentence cannot be put into a single coherent syntactic and semantic analysis; and punctuation handling will be a highly probable point of failure here. Watch out for sentences broken in two by incorrectly interpreted punctuation; this cannot be corrected with macro substitutions since these rules must always operate within the sentence boundaries already found.

12. PyElly Tokenizing and Parsing

Parsing and tokenization are usually invisible in PyElly operation, which should help to simplify the development of natural language applications. Still, we do sometimes need to look under the hood, either when something goes wrong or when efficiency becomes an issue. So this section will take a deep dive into how PyElly analyzes a sentence, a procedure that has evolved over many decades to its present elaboration.

PyElly follows the approach of compiler-compilers like YACC. Compilers are the indispensable programs that translate code written in a high-level programming language like Java or C++ into the low-level machine instructions that a computer can execute directly. In the early days of computing, all compilers were written from scratch; and the crafting of individual compilers was complicated and slow. The results were often unreliable.

To streamline and rationalize compiler development for a proliferation of new languages and new target machines, compiler-compilers were invented. These provided prefabricated and pretested components that could be quickly customized and bolted together to make new compilers. Such standard components typically included a lexical analyzer based on a finite-state automaton and a parser of languages describable by a context-free grammar.

Using a compiler-compiler of course limits the options of programming language designers. They have to be willing to work with the constraint of context-free languages; and the individual tokens in that language (variables, constants, and so forth) have to be recognizable by a simple finite-state automaton. Such restrictions are significant, but being able to have a reliable compiler running in weeks instead of months is so advantageous that almost everyone can accept the tradeoffs.

The LINGOL system of Vaughan Pratt adapted compiler-compiler technology to help build natural language processors. Natural languages are not context-free, but life is simpler if we can parse them as if they were and then take care of context sensitivities through other means like local variables in semantic procedures attached to syntax rules. PyElly follows the LINGOL plan and takes it even further.

12.1 A Bottom-Up Framework

A parser analyzes an input sentence and builds a description of its structure. As noted earlier, this structure can be represented as a kind of tree, where the root of the tree is a phrase node of the syntactic type `SENT` and the branching of the tree shows how complex structures break down into simpler structures. A tool like PyElly must be able to build such trees incrementally for a sentence, starting either at the bottom with the basic tokens from the sentence or at the top by putting together different possible structures with `SENT` as root and then matching them up with the parts of the sentence.

One can debate whether bottom up or top down is better, but both should produce the same parse tree in the end. We can have it both ways by adopting a basic bottom-up

framework with additional checks to prevent a parse tree phrase node from being generated if it would not show up in a top-down analysis. PyElly does this through a true/false matrix m(X,Y) telling whether a syntactic type Y could eventually satisfy a goal of X at parse position; it is automatically compiled when PyElly loads its grammar rules.

LINGOL and subsequently PyElly both take this restricted bottom-up approach. Doing so is quite efficient, and the various resulting subtrees can provide helpful information when parsing fails or when a translation goes wrong. Bottom-up is also more convenient for computing plausibility scores with PyElly cognitive semantics.

The PyElly bottom-up algorithm operates on a queue that lists the newly created phrase nodes of a parse tree. These still need to be processed to create the phrase nodes at the next higher levels of our tree. Initially, the queue is empty, but we then read the next token in an input sentence and look it up to get some new bottom-level parse tree nodes to prime our queue. A token can be a single word or phrase, a number, punctuation, an arbitrary alphanumeric identifier, or a complex entity like a calendar date.

PyElly parsing then runs in a loop, taking the node at the front of its queue and applying grammar rules to create new nodes to be appended to the back of the queue for further action. This process keeps going until the queue finally empties out. At that point, PyElly will then try to read the next token from a sentence to refill the queue and proceed as before. Parsing will stop after every token in sentence has been seen or when current grammar rules cannot generate any more phrase nodes past a given token position.

There is one special circumstance when a new node will not be added to the end of a queue. If there is already a phrase node of the same syntactic type with the same syntactic features built up from the same sentence tokens and if the new node does not have the *UNIQUE syntactic feature, PyElly will note an ambiguity here and will attach the new node as an alternative to the already processed node instead of queueing it separately for further tree building.

This consolidation of new ambiguous nodes serves to reduce the total number of nodes generated for the parsing of a single sentence. Otherwise, PyElly would have to build parallel tree structures for both the old node and the new node without necessarily any benefit. The *UNIQUE syntactic feature will allow you to override the handling of ambiguities here if you really want to do so.

In any event, PyElly immediately computes the cognitive semantic plausibility score of each new phrase as it is generated in bottom-up parse tree building. Whenever an ambiguity is found, PyElly will find the alternative with the highest plausibility and use it in all later processing of a sentence. All the other alternatives will, however, be retained for reporting, for possible backup on a semantic failure, or for automatically adjusting biases to insure that the same rule will not always be taken when there are multiple rules with the same semantic plausibility.

12.2 Token Extraction and Lookup

PyElly token lookup is complicated because it can happen in many different ways: external vocabulary tables, FSA pattern rules, entity extraction, the internal dictionary rules for a grammar, and builtin rules like those for punctuation recognition. These possibilities must also interact with macro substitution, inflectional stemming, and morphological analysis; and so it can be hard to follow what is going on here.

PyElly breaks up a sentence into a sequence of tokens, each a single- or multi-word term, a common expression, a word fragment, a name or other complex entity, a string of defined format like a number, or punctuation. PyElly parsing goes from left to right in a sentence, applying its language rules to get the longest possible token at the next sentence position. For an application `A`, the full lookup procedure is currently as follows:

1. If number rewriting is enabled, rewrite any spelled-out number like SIXTEEN HUNDRED in the current sentence position as digits plus any ordinal suffix like -ST, -RD, or -TH. A spelled-out fraction like THREE-EIGHTHS becomes 3/8THS.

2. Apply macro substitution rules at the current position.

3. Try also to match up the next input up to the next space or other separator with the FSA pattern table for `A`; queue up matches as leaf phrase nodes at the current position if they are consistent with top-down parsing expectations (derivability).

4. Try entity extraction at the current position; put matches as phrase nodes into the PyElly parsing queue if consistent with top-down expectations at the current position. Entities must be within a single sentence; otherwise, PyElly puts no restrictions like above on what extraction code (written in Python) can do for matching. An entity string can include punctuation and spaces.

5. Look up the next input text in the external vocabulary table for `A`; put matches into the PyElly parsing queue as parse tree leaf phrase nodes if consistent with top-down parsing expectations at the current position according to the current grammar rules for `A`. PyElly has special punctuation rules on how far ahead to scan for a match.

6. If steps 3, 4, or 5 have queued up phrase nodes, keep only those with the longest extent. These should be over the same extent of input text.

7. If any queued phrases are longer that the next simple input token, we are done with the generation of leaf phrase nodes and ready to start the main parsing loop.

8. Otherwise, extract the next token of alphanumeric plus embeddable punctuation characters from PyElly input with inflectional stemming and macro substitution.

9. If inflectional stemming is enabled, then apply it to the next token. Put any inflectional endings back into PyElly input.

10. Apply macro substitution rules again. These may override inflectional stemming.

11. Look up the input token as a single word in both external vocabulary table and the internal dictionary for `A`. Queue up a phrase node for any matches here if consistent with top-down expectations and the match is as long as what has been seen so far.

12. If we have queued phrases from any of the preceding steps, we are done with tokenization and ready to go into the PyElly main parsing loop.

13. Otherwise, morphologically analyze our current single-word token with the rules in effect for `A`. Put any suffixes found here back into PyElly input. If analysis resulted in a new token, look it up in the external vocabulary table and in the internal dictionary for `A`. If found and consistent with top-down expectations and long enough, queue up phrase nodes for each match.

14. If there are any queued up phrase nodes, we are done.

15. Otherwise, if nothing has been queued up yet, check if the next token is standard punctuation. If so and the punctuation is consistent with top-down expectations, enqueue a phrase node of syntactic type `PUNC` and quit tokenization.

16. If nothing has yet been queued up, then create a phrase node of `UNKN` type for the next single token up to a break character. This will be without any top-down consistency check.

The lookup process should produce a queue of at least one phrase node for the next token. We then will start in on parse tree building with queued up nodes and continue until the queue is exhausted. When that happens, it is time to look for another token from which to refill our queue until all of an input sentence has been processed.

12.3 Building a Parse Tree

Given bottom-level phrase nodes in the PyElly parsing queue, we can start to build up a parse tree from them. The basic algorithm here is from LINGOL, but it is similar to other bottom-up parsing procedures in systems driven by context-free grammars. The next subsection will cover the details of the basic algorithm's main loop, and the two following subsections will describe PyElly extensions to that algorithm.

12.3.1 Context-Free Analysis Main Loop

At each step in parsing, we first enqueue the lowest-level phrase nodes for the next piece of an input sentence, with any ambiguities already identified and resolved. Then for each queued phrase node, we find all the ways that the node will fit into a parse tree currently being built. This is called "ramification" in PyElly source code commentary.

For newly enqueued phrase node, PyElly ramification will go through three steps when the syntactic type of the node is `X`:

1. Look for grammar rules of the form `Z->Y X` that have earlier found a `Y` and has set a goal of an `X` in the current position. For each such goal found, create a new phrase node of type `Z`, which will be at the same starting position as phrase `Y`.

2. Look for rules of the form `Z->X`. For each such rule, create a new node of type `Z` at the same starting position and with the same extent in a sentence as `X`.

3. Look for rules of the form `Z->X Y`. For each such rule, set a goal at the next position to look for a `Y` to make a `Z` at the same starting position as `X`.

A new phrase node will be vetoed in steps 1 and 2 if inconsistent with a top-down algorithm. The same derivability matrix was employed in token lookup. Each newly created node will be queued up for further processing with the three steps above. When all the phrase nodes ending at the current sentence position have been ramified, PyElly parsing advances to the next position.

The main difference between PyElly basic parsing here and similar bottom-up context-free parsing elsewhere is in the handling of ambiguities. Artificial languages generally forbid any ambiguities in their grammar, but natural languages are full of them and so we have to be ready to handle them. In PyElly, the solution is to resolve ambiguities outside of its ramification steps.

PyElly sees an ambiguity only when two phrase nodes of the same syntactic types and features cover the same tokens in a sentence. For example, the single word THOUGHT could be either a noun or the past tense of a verb. This will have to be resolved at some point, but if they are marked with different syntactic categories or have different syntactic features, PyElly will put off making any resolution.

It is possible that a parsing ambiguity may found for a phrase node after it has already been ramified. This is no problem if that previous node has a higher plausibility than the new phrase node producing the ambiguity; but if the new phrase is more semantically plausible, then it must replace the old phrase, and the plausibility scores of all of the old phrase's ramifications must also be adjusted upward to reflect the replacement. Such changes of the plausibility of other phrase nodes may in turn require adjustment of their previous ramifications as well. PyElly handles all of this automatically.

12.3.2 Special PyElly Modifications

Except for ambiguity handling, basic PyElly parsing is fairly generic. We can be more efficient here by anticipating how grammar rules for natural language might benefit from special handling as compared to those for context-free artificial languages. The first extension of the core algorithm is the introduction of syntactic features as an extra condition on whether or not a rule is applicable for some aspect of ramification.

On the right side of a rule like `Z->X` or `Z->X Y`, you can specify what syntactic features must and must not be turned on for a queued phrase node of syntactic type `X` to be

matched in steps 2 and 3 above and for a queued phrase node of type `Y` to satisfy a goal based on a rule `Z->X Y` in step 1. This checking involves extra code, but it is straightforward to implement with bit-checking operations.

There is also a special constraint applying to words split into a root and an inflectional ending or suffix (for example, HIT -ING). The parser will set flags in the first of the resulting phrase nodes so that only step 3 of ramification will be taken for the root part and only step 1 will be taken for each inflection part. A parse tree will therefore grow more slowly than otherwise expected, making for faster parsing and less overflow risk.

12.3.3 Type 0 Grammar Extensions

The introduction of the PyElly `...` syntactic type complicates parsing, but handling the type 0 grammar rules currently allowed by PyElly turns out to require only two localized changes to its core context-free algorithm.

1. Just before processing a new token at the next position of an input sentence, generate a new phrase node for the grammar rule `...[.1]->`*nothing*. This is not a normally legal rule! Enqueue the node and get its ramifications immediately.

2. Just after processing the last token of an input sentence, generate a new phrase node for the grammar rule `...[.2]->`*nothing*. Enqueue it and get its ramifications immediately.

Those reading this manual closely will note that the two rules here have syntactic features associated with `...`, which Section 8 said was not allowed. That restriction is still true, and that is because PyElly reserves the syntactic features of `...` to make the type 0 logic handling work properly as done above.

The difficulty here is that the `...` syntactic type is prone to producing ambiguities. This will be especially bad if the PyElly parser cannot distinguish between a `...` phrase that is empty and one that includes actual pieces of a sentence. So PyElly itself keeps track by using syntactic features here, but keeps that information invisible to users.

The solution will propagate up the syntactic feature `[.1]` to indicate an empty phrase due to case 1 and the feature `[.2]` to indicate an empty phrase due to case 2. Though invisible, a grammar will still need to guide this explicitly through setting `*LEFT` or `*RIGHT` in rules for syntactic feature inheritance when a rule involves `....`

12.4 Success and Failure in Parsing

For any application, PyElly automatically defines the special grammar rule:

```
g:SENT->SENT END
```

—

This rule will never be realized in an actual phrase node, but the basic PyElly parsing algorithm uses this rule to set up a goal for the syntactic type `END` in phase 3 of ramification. After a sentence has been fully parsed, PyElly will look for an `END` goal at the position after the last token extracted from the sentence. If no such goal is found, then we know that parsing has failed; otherwise, we can then run the generative semantics for the `SENT` phrase node that generated the `END` goal just found.

There may be more than one `END` goal in the final position, indicating that their respective generating `SENT` phrase nodes were not collapsed as an ambiguity because of syntactic feature mismatch. As a special case, PyElly will compare their cognitive semantic plausibility scores select the most plausible and run its generative semantic procedure to get the interpretation for a sentence. This is equivalent to making actual phrase nodes based on the implicit PyElly `SENT->SENT END` rule, which would trigger PyElly ambiguity handling as just described.

Failure in parsing gives us no generative semantic procedure to run, and our only recourse then is to dump out intermediate results and hope that someone can spot some helpful clue in the fragments of analysis. If the failure is due to something happening in semantic interpretation, though, PyElly can automatically try to recover by backing up in a parse tree to look for an ambiguity and selecting a different alternative at that point.

12.5 Parse Data Dumps and Tree Diagrams

PyElly can produce dumps of parsing data, including all the complete or partial parse trees built up for a sentence. In a successful analysis, this helps in verifying that PyElly is running as expected. In a failed analysis, the partial parse trees will provide clues about what went wrong. For example, you can see where the building of a parse tree had to stop and whether this was due to a missing rule or unexpected input text.

All this information is written to the standard error stream. Such output originally was an informal debugging aid, but has proven so useful that it is now built into PyElly operation. The most important part of parse data dumps are the trees. These will be presented horizontally, with their highest nodes on the left and with branching laid out vertically. For example, here is a simple 3-level subtree with 4 phrase nodes:

```
sent:0000——ss:8001┬noun:8000 @0 [nnnn]
   6 =  3    4 =  2|   1 =  1
                   └verb:0000 @1 [vvvv]
                       2 = -1
```

Each phrase node in a tree display will have the form

```
type:hhhh
   n =  p
```

Where `type` is the name of a syntactic type truncated to 4 characters, `hhhh` is hexadecimal for the associated feature bits (16 are assumed), `n` is a phrase sequence

number indicating the order in which it was generated in a parse, and `p` is the numerical semantic plausibility score computed for the node. The nodes are connected by Unicode drawing characters to show the kind of branching in grammar structures.

To interpret the feature bits `hhhh` here, you should look at the encoding of feature names produced by grammarTable.py. The feature encoded as 0 will be the leftmost bit and will show up as the hexadecimal `8000`; the feature encoded as 1 will be `4000`. In the above example, the top-level node here for type `sent` is

```
sent:0000
   6 =  3
```

This node above has no syntactic features turned on; its node sequence ID number is 6, and its plausibility score is +3. Similarly, the node for type `ss` at the next level is

```
 ss:8001
  4 =  2
```

The actual sentence tokens for a PyElly will be in brackets on the far right, preceded by its sentence position, which starts from 0. In the example above, the tokens are the "words" `nnnn` and `vvvv` in sentence positions 0 and 1, respectively. Every parse tree branch will end on the far right with a position and token, plus a semantic concept if your grammar includes them.

With the analysis of a word into components becoming separate tokens, representing prefixes or suffixes, we can get trees like

```
sent:0000──ss:0000┬──ss:0000─unit:0000─unkn:0000 @0 [it]
  11 =  0   10 =  0│    2 =  0    1 =  0    0 =  0
                   └unit:0000┬unkn:0000 @1 [live]
                       9 =  0│    4 =  0
                             └sufx:0000 @2 [-s]
                                 8 =  0
```

When a grammar includes `. . .` rules, the display will be slightly more complicated, but still follows the same basic format.

```
sent:0000┬sent:C000──ss:C000┬──x:A400┬...:4000 @0 []
  11 =  4│    8 =  4    7 =  4│    3 =  4│    0 = -2
         │                    │          └key:2400 @0 [hello]
         │                    │             2 =  2
         │                    └...:4000 @1 []
         │                        6 = -2
         └punc:2000 @1 [.]
             9 =  0
```

The empty phrases number 0 and number 6 have sentence positions 0 and 1, but these positions are shared by two actual sentence elements HELLO and period (.), as you would expect. Note that the hidden syntactic flags of the `...` type will show up in a displayed tree; just ignore them.

All the examples here show a single analysis of a complete sentence, where PyElly start from for semantic interpretation. This minimal dump will not show any rejected alternative analyses, which might have been generated for ambiguities or for dead ends that led to no full analysis of a sentence recognized as a syntactic type SENT.

When a PyElly analysis fails or when you are unsure whether an analysis is correct, however, you may want to see all the intermediate results of parsing, including ambiguities and dead ends. PyElly can do a full tree dump here showing all phrase nodes in an analysis in the order of their generation.

In a full dump, PyElly first looks for the node with the highest sequence ID number not yet shown in any parse tree for the current dump. PyElly then shows the subtree under that node as done above for the subtree under `sent` and loops back in this way until all phrase nodes are accounted for. For a long sentence, we may have tens of thousands of phrase nodes, but each node will show up in only one subtree.

In addition to all the trees and subtrees, a PyElly full dump will also show the goals at the final position in a sentence analysis, all grammar and internal dictionary rules applied plus the phrases nodes generated, and all ambiguities found in the process. This information should allow you to reconstruct how PyElly parsed a sentence.

Full dumps are default when you are running ellyBase.py from a command line, regardless of whether PyElly succeeded or failed on an input sentence. With ellyMain.py, you will see only a minimal dump if PyElly succeeded, but a full dump if PyElly has failed.

For example, suppose that we have the following trivial grammar, which allows for sentences consisting of either a `NOUN` plus a `VERB` or a `VERB` plus a `NOUN`:

```
# trivial grammar
p:do

_
  left
  space
  right

__
g:sent->noun verb

_
  (do)

__
g:sent->verb noun
 >>-

_
  (do)

__
```

```
g:noun->noun sufx

_
  (do)

__
g:verb->verb sufx

_
  (do)

__
d:dog  <- noun

__
d:dog  <- verb

__
d:-s   <- sufx

__
d:bark <- verb

__
d:bark <- noun

__
```

Here is an example of an ellyBase full dump for an analysis of a short sentence:

```
> dogs bark.

parse FAILed!
dump all

dumping from phrase 8 @0: typ=0 syn[00 00] sem[00 00] : bia=-1 use=0
 sent:0000┬verb:0000┬verb:0000 @0 [dog]
    8 = -1│   4 =  0│   1 =  0
          │         └sufx:0000 @1 [-s]
          │             2 =  0
          └noun:0000 @2 [bark]
              6 =  0

dumping from phrase 7 @0: typ=0 syn[00 00] sem[00 00] : bia=0 use=0
 sent:0000┬noun:0000┬noun:0000 @0 [dog]
    7 =  0│   3 =  0│   0 =  0
          │         └sufx:0000 @1 [-s]
          │             2 =  0
          └verb:0000 @2 [bark]
              5 =  0

rules invoked and associated phrases
rule    2: [7]
rule    3: [8]
rule    4: [3]
rule    5: [4]
rule    6: [0]
rule    7: [1]
rule    8: [2]
rule    9: [5]
rule   10: [6]

3 final goals at position= 3 / 3
 goal 8: sufx typ=6 for [phrase 5 @2: typ=5 syn[00 00] sem[00 00] : bia=0 use=0] rul=5
 goal 9: sufx typ=6 for [phrase 6 @2: typ=4 syn[00 00] sem[00 00] : bia=0 use=0] rul=4
 goal 10: end typ=1 for [phrase 7 @0: typ=0 syn[00 00] sem[00 00] : bia=0 use=0] rul=1

9 phrases altogether
```

```
ambiguities
sent  0000:  7 (+0/0)  8 (-1/0)

4 raw tokens= [[dog]] [[-s]] [[bark]] [[.]]
9 phrases, 11 goals
```

Parsing fails for the input "dogs bark." because our simple grammar expects no punctuation at the end of a sentence. In the full dump, we first see the two subtrees for the first three tokens of the input sentence. The final position with goals are defined is 3, and the goals there are listed; the position of the actual last token seen is also 3, but this may not always be so. In the list of ambiguities we have phrases 7 and 8, both of type `SENT` with no syntactic features set; phrase 7 starts the listing because it has a higher semantic plausibility score as result of the cognitive semantics for rule 3.

If a semantic concept is defined for a phrase at a leaf node in a parse tree, a PyElly tree dump will show the concept immediately after the bracketed token at the end of an output line. For example, the augmented printout for tree the first tree from the example above might become

```
sent:0000┬noun:0000┬noun:0000 @0 [dog] 02086723N
   4 =  0│    2 =  0│    0 =  0
         │          └sufx:0000 @1 [-s]
         │              1 =  0
         └verb:0000 @2 [bark] 01049617V
             3 =  0
```

where `02086723N` and `011049617V` are WordNet-derived concept names as described in Subsection 10.4.1 above. If no concept is defined for a leaf node, then the tree output will remain the same as before. This is the case for the suffix -S here.

12.6 Parsing Resource Limits

PyElly is written in Python, a scripting language that can be interpreted on the fly. In this respect, it is closer to the original LINGOL system written in LISP than to its predecessors written in Java, C, or FORTRAN. PyElly takes advantage of Python object-oriented programming and list processing with automatic garbage collection.

Unless you are running on a platform with tight main memory, PyElly should readily handle sentences of up to a hundred tokens with little difficulty. When writing grammar tules to describe long and complex sentences, however, you will have to control the combinations of the possible interpretations from those rules. Too many combinations here will result in phrase node overflow, producing no translation.

With high degrees of ambiguity in a PyElly grammar, the total number of phrase nodes allocated in generating a complete parse for a sentence can grow exponentially with the number of distinct tokens in the input. This will slow processing noticeably and can even crash your computer. Be careful especially when using the predefined `*unique` syntactic feature or when many words have to be treated syntactically as `UNKN`.

In general, PyElly will generate every possible parse trees for a sentence. Analyses will be reduced, though, by immediate resolution of ambiguities of the same syntactic type with the same syntactic features over the same text, but sentences with many competing interpretations having to be resolved higher up in a parse tree can still clog up PyElly processing. Inflectional stemming or morphological analysis will of course produce extra tokens beyond the basic count of words and punctuation for a sentence.

The PyElly `ellyConfiguration` module as of v1.2.2 imposes a 50,000 node limit on the total number of phrases available for a sentence parse tree. This is generous, since parsing can seem glacially slow when over 10,000 phrase nodes are generated. If a sentence analysis hits the node cutoff, PyElly will throw an overflow exception that will be caught at the top level of ellyBase; only a token list dump will be then be shown.

You can raise the phrase node limit by editing the `phraseLimit` parameter definition in the file `ellyConfiguration.py`. This probably will not help much, however. Node overflow means that your grammar and vocabulary really need to be rethought and tightened up. A remedy here might be to use syntactic features strategically to restrict the applicability of alternate syntax rules and to define more terms explicitly, especially ones with multiple words. Just recognizing a phrase like `in the course of` as a single token can often halve the total phrase node count for a long sentence.

Certain words in a language are unusually prone to producing ambiguities. In English, a few to watch out for are the function words IN, THAT, MAY, and TO. If a sentence contains many of these words, it may produce an overflow even though it might look quite short and simple. The content words NEAR and BASE can also be problematic. Beware when a word has more than three associated syntactic categories in a PyElly language definition.

As of v1.3.5, you can also insert a control character into your input to force PyElly to split the analysis of a long sentence into two parts when you provide an appropriate syntax rule here. This insertion can be done through a macro substitution rule with a pattern that identifies where you want to put break into input text. The control character here could be anything not normally found in input text, but in particular, PyElly predefines ASCII RS (record separator or Unicode u'\u001E) for this purpose.

To make its use easier, the PyElly separator character will automatically be put into the internal dictionary of every PyElly grammar as a 1-character token of syntactic type `SEPR`. The separator character can be specified on the right side of a macro substitution as `\\s`. You then just have to define grammar rules of the `X->Y SEPR` or `X->SEPR Y` to tell PyElly where to expect a `SEPR` token in the input for an application. PyElly parsing will take care of everything else.

There may be more than one separator in a given sentence. Each has the effect of dividing a sentence analysis so that no phrase completely to the left of the separator will generate a goal to the right of the separator. Fewer goal nodes generated will mean fewer phrase nodes. To get maximum benefit here, you should have only one 2-branch syntax rule with `SEPR`. and define it at the level of the `SENT` syntactic type.

With PyElly defaults, the analysis of a sentence with a separator token in input will still produce a single parse tree. Your generative semantics will work exactly the same as before to stitch together a proper output translation for the original input sentence. Unless you have some special requirement, a `SEPR` phrase node should always translate to a null string, since it was not in the original input text.

Outside of phrase and goal nodes, the main defined resource restrictions in PyElly are on the total number of syntactic types (112) and the total number of different syntactic features or of different semantic features for a phrase node (both 16). These are fixed to allow for preallocation of various arrays used by the PyElly to simplify lookup of grammatical symbols when setting up a language definition. The limits also make the formatting of parse trees easier.

You can change these limits on syntactic types and syntactic and semantic feature counts by editing the definitions in the PyElly `symbolTable.py` module. The current numbers should be quite enough for ordinary applications, however. If you increase the feature count past 16, you also must change the `parseTreeWithDisplay.py` module so that its formatting of parse trees will allow more than four hexadecimal digits to represent syntactic feature bits. This will be non-trivial.

You will be better off leaving the limit on the number of syntactic features alone. Instead, try increasing the number of syntactic feature sets or the number of syntactic types; although this may make your grammar rules harder to understand. We are taught in grammar school that there are only eight parts of speech in English, but it is often advantageous to define many more syntactic types in order to make individual syntactic rules more specific. Experiment here to see what works best for you.

In simple PyElly applications, the limits here will be unimportant. In the `marking` example application with tens of thousands of grammar and vocabulary rules, however, the language definition has reached over 100 syntactic types, 9 syntactic feature sets, and 3 semantic feature sets. Almost all of the syntactic and semantic feature sets are full. In a PyElly application that has grown over time, you might want to reorganize its language definition periodically to be more efficient in its use of symbols.

13. Developing Language Rules and Troubleshooting

PyElly rewrites input sentences according to the rules that you provide. A natural language application can involve up to ten different `*.*.elly` text files for rule definition, with some containing hundreds or even thousands of rules. There are plenty of ways to go wrong here; and so we all have to be systematic in developing PyElly applications, taking advantage of all the tools available when problems arise.

Even when trying to do something simple, you must always be alert and cultivate good habits. In general, the best way to use PyElly is to approach a solution bit by bit, trying to take care of just one sentence at a time. Let PyElly to check everything out for you at each step and go no further until everything is clean and satisfactory. Remember that a change in your rules can break previous analyses. Always expect the worst.

Application building will never be a slam dunk, but remember that you are already a natural language expert! Despite enormous advances in hardware and software, an intelligent young child nowadays still knows more about natural language than Siri or Watson. If you can harness some basic analytical skills and add some programming chops to your innate language expertise, then you should do well with PyElly. Just keep your goals clear and proceed slowly and carefully.

Start with the simplest sentences requiring only a few rules. Once these can be handled successfully, move on to more complex sentences, adding more rules as needed to describe them. With a modular PyElly rule framework, you should be able to build on your previous rules without having to re-create them. This is one big advantage of processing sentences recursively around the syntactic structures of sentences.

When testing out a new sentence, you should not only verify that PyElly is producing the right output, but also inspect its parse tree dump to see that it is doing what you expect from your current grammar rules. If everything is all right, then add the sentence and its translation to a list to be run in regression testing with `ellyMain.py` later. Such testing will take only seconds or minutes, but must be done.

13.1 Pre-Checks on Language Rule Files

As your language definition files get longer, PyElly can help to verify that each of them is set up correctly and makes sense by itself before you try to bring everything together. This can be done by running the unit tests of the modules to read in definition files and checking the acceptability of tables of rules. Running `ellyBase.py` or `ellyMain.py` will also do some of this, but you can sort out issues more easily when looking at only one definition file at a time. For example, with application `X`, you can run any or all of the following unit tests from your command line:

```
python grammarTable.py X

python vocabularyTable.py X

python patternTable.py X

python macroTable.py X

python nameTable.py X

python compoundTable.py X

python conceptualHierarchy.py X
```

Each command will read in its corresponding `X.*.elly` file, check for errors, and warn about possible problems. Such unit test checking will be more thorough than regular rule compilation. If a table or a hierarchy can be generated, PyElly will dump it all out for inspection, except for external vocabularies, which can be too big for a full listing.

The vocabulary, pattern, macro, and name table unit tests will also prompt you for additional examples to run against their rules for further verification of correct lookup or matching. This can be helpful when debugging a PyElly language definition with a specific problematic bit of text.

PyElly error messages from language definition modules will always be written to sys.stderr and will have a first line starting with "`**` ". They may be followed by a description line starting with "`*` " showing the input text triggering the problem. For example, here is an error message for a bad FSA rule for assigning a part of speech:

```
** pattern error: bad link
*  at [ 0 *bbbb* ZED start ]
```

PyElly will continue to process an input rule definition file after finding an error so as to catch as many definition problems as possible in one pass. No line numbers are given in error messages because PyElly normalizes all its input to simplify processing for definition modules. This will strip out comments and will eliminate any blank lines, thus changing the original numbering of lines.

Once each separate table has been validated in isolation, you can run `ellyBase.py` to load everything together. Its unit test will run a cross-table check on the consistency of your syntactic categories and of your syntactic and semantic features. For application `X`, do this with the shell command

```
python ellyBase.py X
```

This is also how you would normally test the rewriting of individual test sentences, but the information provided by PyElly from the loading of language rules is a good way to look for omissions or typos in your language rules, which can be hard to track down otherwise. Running `ellyMain.py` will skip this detailed kind of checking.

13.2 A General Application Development Approach

For those wanting more specifics on setting up PyElly language definition files, here is a reasonable way to build a completely new application `X` step by step.

1. Set up initially empty `X.g.elly`, `X.m.elly`, `X.stl.elly`, `X.n.elly`, and `X.v.elly` files. For the other PyElly language definition files, taking the defaults should be all right as least to start with.

2. Select some representative target sentences for your PyElly application to rewrite. Five or six should be enough to start with. You can add more as you go.

3. Write `G:` grammar rules in `X.g.elly` to handle to handle one of your target sentences; leave out the cognitive and generative semantics for now. Just check for correctness of the rules by running the PyElly module `grammarTable.py` with `X` as an argument.

4. Add the words of a target sentence as `D:` internal dictionary rules in `X.g.elly` or as vocabulary entries in `X.v.elly`. Run `grammarTable.py` or `vocabularyTable.py` with `X` as an argument to verify correctness of language definition files as your working vocabulary changes.

5. Run PyElly module `ellyBase.py` with `X` as an argument to verify that your language definition files can be loaded together. Enter single target sentences as input and inspect the parse data dumps to check that analyses are correct. Ignore any generated output for now.

6. Write the generative semantic procedures for your grammar rules and check for correctness by running `grammarTable.py`. If you have problems with a particular semantic procedure, copy its code to a text file and run the PyElly module `generativeProcedure.py` with the name of that text file as an argument.

7. When everything checks out, run `ellyBase.py` with `X` as an argument and verify that PyElly translates a target sentence as you want.

8. When everything is working for a target sentence, repeat the above from step 2 with other sentences. Always test your new system against all previous target sentences to ensure that everything is still all right after any change of language rules. You can create `X.main.txt` and `X.main.key` files to automate such testing with `doTest`.

13.3 Miscellaneous Tips

This subsection is a grab bag of advice about developing nontrivial PyElly grammars and vocabularies. It is distills many lessons learned about rule-based language definition for all kinds of applications since PARLEZ, the original PyElly ancestor. In writing language rules for a new application, expect to make many mistakes; but try to avoid repeating the same old ones.

13.3.1 General Advice

1. PyElly has a bit over eleven thousand lines of Python code, exclusive of commentary. It will translate natural language strings into other kinds of strings and nothing more. In other words, you cannot replicate Watson or Siri with PyElly. Go ahead and push the limits, but be aware that PyElly is not magic and that developing large sets of language rules can be hard work.

2. The PyElly `ellyMain.py` module is the better choice to rewrite batches of raw text data because of its sentence recognition. If you are working with only one input sentence at a time, run `ellyBase.py`. This is friendlier for interactive processing and also provides more diagnostic information about its translations.

3. PyElly analysis revolves around sentences, but you can define them however you want and need not follow what you learned in 8th grade. It is all right to break sentences at colons and semicolons or even at certain subordinate conjunctions. Shorter sentences will simplify your grammar rules and parsing.

4. In developing a grammar, less is better. You are more likely to hit trouble when you increase the number of syntactic categories and rules. In many applications, for example, you can ignore language details like gender, number, tense, and subject-verb agreement. Avoid making rules for what does not matter to you.

5. Get the syntax of a target input language right before worrying about the semantics. PyElly automatically supplies you with stubs for both cognitive and generative semantics in grammar rules. You can replace those stubs later with full-fledged logic and procedures after you can analyze sentences syntactically.

6. Natural language always has regular and irregular forms. Tackle the regular first in your grammar rules and make sure you have a good handle on them before taking on the irregular. The latter can often be handled by macro substitutions: for example, just change DIDN'T to DID NOT or if tense is unimportant, DO NOT.

13.3.2 Generative Semantics

1. Writing semantic procedures for a grammar is a kind of programming. So, follow good software engineering practices. Divide the rewriting of text into smaller subtasks that can be finished quickly and individually tested. Test thoroughly as you go; never put yrdyomh off until all your language rules have all been written out.

2. Make your generative semantic procedures short, fewer than twenty lines if possible; this will make it easier to verify the correctness of your code visually. If a long procedure is unavoidable, run it first in a separate file using the unit test for the PyElly module `generativeProcedure.py`. Throw in some `TRACE`, `SHOW`, and `VIEW` commands temporarily to monitor what that your code is doing.

3. Be liberal with named generative semantic subprocedures to shorten large blocks of code. A subprocedure may have as few as only two or three commands. Calling such a subprocedure could actually take more memory than equivalent inline commands, but clarity and ease of maintenance will trump efficiency here. Common code used in multiple procedures should always be broken out as a named subprocedure.

4. Group the rules for syntactic types into different levels where the semantic procedures at each level will do similar things. A possible succession of levels here might be (1) sentence types, (2) subject and predicate types, (3) noun and verb phrase types, and (4) noun and verb types with inflections. This is also a good way to organize the definition of local variables for communication between different generative semantic procedures.

13.3.3 Macro Substitutions

1. Macro substitutions will usually be easier to use than syntax rules plus semantic procedures, but they will be fairly specific about the words that they apply to. Syntax rules are better for patterns that apply to broad categories of words.

2. Macro substitution can be dangerous. Watch out for infinite loops of substitutions, which can easily arise with `*` wildcards. Macros can also interact unexpectedly; in particular, make sure that no macro is reversing what another is doing, which can also lead to infinite loops.

3. Macros can match multiple tokens in input when the pattern for a substitution includes one or more '`_`' wildcards. This will match any space character, including ASCII HT, NL, CR, US, and SP or Unicode NBSP and thin space. Literal spaces are prohibited in macro patterns and will halt rule loading if seen by PyElly.

4. You can put the text matched by pattern wildcards into a substitution through the elements `\\1`, `\\2`, `\\3`, and so forth; remember that a '`_`' wildcard is always bound separately, and you must count it correctly in a substitution; for example, the two alphabetic segments of a string matching the pattern '`&@_&@`' will be `\\1` and `\\3`.

5. Try to avoid macros in which the result of substitution is longer that the original substring being replaced. These are sometimes necessary, but can get you into trouble; PyElly will warn you if it comes across such a macro, but will allow it.

6. The ordering of rules in a macro substitution table is important. Rules at the top of a definition file can change the input text seen by a macro further down in the file.

7. PyElly macro substitution comes after transformations of spelled out numbers, but before external vocabulary lookup, and entity extraction. It comes again just before the next token is taken from its input buffer after any inflectional stemming and again just before the next token is taken with morphological prefixes and suffixes split off. This will allow macros to undo word analyses in special cases. Be careful here; macros actions can undo, but cannot themselves be easily undone.

8. Macro substitutions and transformations can change the spelling of words in sentences being processed. Make sure that your internal dictionary and the external vocabulary rules for an application take such changes into account. Sometimes you can change the spelling of a word in a specific context to tell PyElly how to interpret it; for example, you can indicate that an instance of WHAT is a conjunction by rewriting it as cnjWHAT or WHĀT or WHT, which can then be defined unambiguously in a vocabulary table.

9. Avoid macros for matching literal phrases like "International Monetary Fund." Unless you need the wild card matching in macros or want to keep only part of a match, use vocabulary tables instead. This will be faster and easier to debug.

10. Macros can be slow because all rules will be checked again after any successful substitution except when an entire match is deleted by a rule. Each substitution will involve much copying and recopying of text in a buffer.

13.3.4 Patterns for Identifying Syntactic Types

1. The PyElly finite-state automaton for syntax typing can have wildcards in the patterns to be matched for going to a next state. More than one such pattern can be matched by the same text input, allowing for more than one next state. PyElly will automatically keep track of all possibilities here so that everything comes out right in end. Technically speaking, a PyElly FSA is non-deterministic, although it is always equivalent to a deterministic FSA with many more states.

2. Currently, an FSA pattern cannot have a space wildcard _. It can still recognize a token with components joined by hyphens or other punctuation, but not words separated by spaces. This restriction is mainly to keep a PyElly FSA from becoming unmanageably complex. To define more general multi-word tokens, use PyElly external vocabulary table lookup or entity extraction instead.

3. An FSA can have rules just with a `0` as a starting state and `-1` as a next state, which will then look like a deterministic FSA. If more than one pattern matches at a state, however, the FSA will become non-deterministic.

4. The `-2` final state will let you override PyElly default tokenization. This will let you split a token at a character normally in the middle of a single token or join what would be separate tokens at a non-space character normally seen as a separator. This option is for specialized usage and can usually be avoided.

13.3.5 Vocabulary Building

1. Vocabulary building with internal dictionary rules or external vocabulary rules should be the last thing you do. You should define at least a few terms to support early testing, but hold off on the bulk of your vocabulary. Adding vocabulary is easy; adding grammar rules can be tricky because of their many unexpected side effects.

2. In English and other languages including Chinese, certain combinations of words are so frequent that they really should be recognized as a single vocabulary table term; for example, FAKE NEWS. This increases the number of different rules, but can make parsing much easier. Do this liberally when working with longer sentences and with multi-word terms containing otherwise highly ambiguous words like IN.

3. You can also absorb punctuation into vocabulary table entries for lookup. This is most commonly done with hyphens, but quotation marks, commas, apostrophes, and others are also fair game here. It can help greatly when working with awkward kinds of syntax involving embedded punctuation.

4. For large ambiguous grammars, predefine as much vocabulary as possible in order to keep parsing reasonably fast and parse trees smaller. Try to avoid the `UNKN` syntactic type in rules. Use `ellySurvey.py` to find out what tokens appear in the text data you want to process and what status they have with current language definition rules. The file `default.v.elly` (WordNet 3.0) is a good source of possible vocabulary rules specifying parts of speech for words and phrases. It may be missing important terms for a particular content area, however. Add in your own terms as needed, especially multiword terms.

13.3.6 Syntactic and Semantic Features

1. Syntactic and semantic features can be quite helpful. The former lets you be more selective about which rules to apply to an analysis; the latter lets you better choose between different interpretations in case of ambiguity. These can reduce the total number of syntax rules, but will add a different kind of complexity to your grammar.

2. PyElly will allow a syntactic category to be associated with only one set of syntactic feature names and only one set of semantic feature names. Different syntactic categories may share the same set of feature names, which makes it possible for features to be inherited across categories.

3. Features are encoded in a grammar table rule and saved in a parse tree node only as *n* anonymous bits. The same feature name may be in different feature sets, but will not necessarily refer to the same feature bit. This can lead to bad surprises when inheriting syntactic and semantic features through the `*L` or `*R` mechanisms; make sure that inheritance is only with phrases having the same set of feature names. PyElly will check here, but only when feature names are referenced in rules.

4. You must always say whether feature inheritance in a parse tree is from a left or a right descendant. There is no default for either syntactic or semantic features. All the features from a designated descendant will be inherited, but you can always explicitly turn off particular automatically inherited features afterward.

5. Avoid defining your own feature names beginning with '*****'. PyElly will allow this, but it can lead to confusion because such names will normally identify syntactic and

semantic features predefined by PyElly. Also, do not forget the '*****' when you do use one of these PyElly-defined feature names.

6. The `*l`, `*r`, and `*x` syntactic features can be used in ways that disregard their side effects in building parse trees for a sentence analysis. This option should be tried only when you have run short of syntactic features in a grammar; it is most useful at the leaf level of parse tree nodes, where no inheritance is possible anyway.

13.3.7 Parsing Problems

1. To dump out the entire saved grammar rule file for application `A`, run `grammarTable.py`. with `A` as an argument. This will also show generative and cognitive semantics, which `ellyBase.py` omits in its diagnostic output.

2. Keep PyElly parse data dumps enabled in PyElly language analysis and learn to read them. This will be the easiest and most helpful way to obtain diagnostic information when an application is not working as expected (usually the case). Full dumps will show all subtrees generated for any ambiguous analyses, even those not incorporated into actual PyElly output; if you want details, this is where to get it.

3. PyElly will cut a tree display off at 25 levels of nodes by default. You can adjust this limit in the ellyMain and the ellyBase command lines, but deeper trees will be hard to read when output must be broken up to fit into the width of your display.

4. If you run into a parsing problem with a long input sentence, try to reproduce the same problem in with a shortened version of that sentence. This will help to isolate the issue and also make your parse tree dumps easier to read.

5. When a parse fails, the last token in the listing shown in a parse tree dump will show approximately where the failure occurred. It may be a few tokens before, however. Check for a token for which no phrase node was generated. Look also at the last set of goals generated in a bottom-up parsing and see what token position they occur in.

6. If you are working with English input and have not defined syntax rules for handling the inflectional endings -S, -ED, and -ING, a parse will fail on them. The file `default.p.elly` can define certain endings as the syntactic category `SUFX`, but you still need something in your grammar rules to do something with them.

13.3.8 Diagnostic Options in Rewriting

1. To verify the execution of a generative semantic procedure, put in a `TRACE` command. This will write to the standard error stream whenever it is run. In a procedure attached to a phrase node, `TRACE` will show the syntactic type and starting position of that phrase node and the grammar rule governing the node. In a named subprocedure, PyElly will also show the first attached generative procedure calling a chain of subprocedures.

2. To see the value of local variables during execution for debugging, use the `SHOW` generative semantic command, which writes to the standard error stream. Remember that both local and global variables will always have string values, with an empty string possible. Local variables may be from further up in a parse tree.

3. To see the contents of your current and your next output buffer at some point in a generative semantic procedure, use the `VIEW` command. This is also a good way to learn what the various PyElly semantic commands do.

4. Minimize the number of `TRACE`, `SHOW`, or `VIEW` commands at one time. Being overwhelmed with too much instrumentation can be as problematic as having too little information. Clean up such commands when no longer needed.

13.3.9 Punctuation

1. Punctuation is tricky to handle. Remember that a hyphen will normally be treated as a token break; for example, GOOD-BYE currently becomes GOOD, - , and BYE unless specified as a single term by pattern matching, entity extraction, or external vocabulary lookup. An underscore or an apostrophe is not normally a word break.

2. Quotation marks can be quite troublesome, since this may involve special Unicode forms as well as ambiguous uses of the ASCII apostrophe character. PyElly predefines quotation marks, including Unicode variations for formatted text, but these all must still be properly handled by your grammar rules.

3. Default vocabulary rules for punctuation can be overridden by `D:` rules defined by a PyElly application, but only when their cognitive semantic scoring make these particular interpretations preferred by PyElly.

4. Macro substitution will not apply across sentence boundaries. To override punctuation otherwise seen as a sentence stop, define PyElly stop exception rules (described in Subsection 11.1.1). Note that macros can already deal with embedded periods and commas, since these will be seen as non-stopping punctuation.

13.3.10 Ambiguity

1. Ambiguity is often seen as a problem in language processing, but intentional ambiguity in small doses can sometimes simplify a grammar. For example, the English word IN can be either a preposition or a verb particle. Define rules for both usages with appropriate cognitive semantic plausibility scores and let PyElly figure out which one to apply. To avoid parse node overflow, though, avoid bringing in unnecessary ambiguity with rare senses of words like IN; for example, THE INS AND OUTS or AN IN. Have PyElly recognize the whole phrase.

2. Ambiguous grammar rules and alternative external vocabulary definitions should be assigned different plausibility scores according to their probability in actual text.

Otherwise, PyElly may switch between them arbitrarily when analyzing different sentences in the same session, which is probably not what you want.

3. When assigning plausibility scores to rules, try to keep adjustments either mostly positive or mostly negative. Otherwise, they can cancel each other out in unexpected and often unfortunate ways. Plausibility for a phrase is computed by adding up all the plausibility scores for a phrase and all its immediate subconstituents. Note that scores may change in the middle of an analysis as ambiguities are resolved.

4. To see what is going on in cognitive semantic logic for a particular phrase node, put `?>>?` as the first clause. This will turn on tracing for subsequent clauses in a rule and write to the standard error stream to verify that the logic is being executed. It will identify which clause is actually used to compute plausibility. This can track the generation of new phrase nodes and show what grammar rules are used.

5. When a problem arises in a particular input sentence, run the sentence by itself with `ellyBase.py` to get more diagnostic information. This will show what grammar rule is tied to each phrase node and what ambiguities it is associated with.

6. A common failing is when the wrong interpretation gets the highest plausibility score. To remedy this, identify the phrase nodes contributing most to that score and check their ambiguous counterparts for a better interpretation. This should tell you where scoring in grammar or vocabulary rules needs adjustment. You may have to go up or down in a parse tree to find which ambiguities to focus on.

7. The predefined `*unique` syntactic feature allows you to control where PyElly ambiguity resolution happens. A phrase marked with that feature will remain unresolved in PyElly parsing even when there is another phrase of the same syntactic type and with the same syntactic features covering the same set of input sentence tokens. The `*unique` feature cannot be inherited.

8. Ambiguity will produce many possible parsings of a sentence, often leading to exponential growth in total processing. A sentence with four points of ambiguity, each with two possible interpretations, will have sixteen possible analyses. The problem worsens in longer sentences with more room for ambiguity. To minimize this problem, avoid input tokens taken to be `UNKN`, and be sparing with the syntactic feature `*unique` in grammar rules. When reasonable, recognize multi-word phrases as single tokens. Use RS control characters to partition analyses if possible.

13.3.11 Name Recognition

1. The PyElly name recognition capability is based on the idea that we can list out the most common names in a particular domain of text input and that other names will be lexically or phonetically similar and can be contextually inferred. This will reduce the number of `UNKN` tokens seen by PyElly and will help in recognizing isolated surnames after the appearance of a full name.

2. The `name.n.elly` definition file is quite diverse, being based on U.S. Census data. For Arabic, Chinese, or Russian names, however, you probably want to extend your tables with sample names from other sources. Do not expect to get away with only a little work here. Current PyElly phonetic signatures reflect American pronunciation and may have to be adjusted to make inferences about foreign names more reliable.

13.3.12 Other Recommended Practices

1. Experiment. PyElly offers an abundance of language processing capabilities, and there is often more than one way to do something. Find out what works for you and share your ideas with others. You may be the first person to employ a particular combination of PyElly language rules.

2. Fix any language definition problems due to typos first. It is easy to make mistakes in keying in names of syntactic categories or names of semantic or syntactic features. Always check the complete listing of grammar symbols in ellyBase diagnostic output to verify that there are no unintended ones due to typos.

3. PyElly uses '`#`' to mark comments in rule files and also as a wildcard matching numeric characters '`0`' through '`9`'. This is a hazard. To be safe, use a backslash (`\`) to escape a single '`#`' wildcard in a rule file to make it unambiguous. A space must appear before and after a '`#`' marking a comment in the middle of an input line.

4. All semantic subprocedures must be defined in the `*.g.elly` file for an application, but not necessarily prior to any call to a subprocedure in the generative semantics. Such calls are also allowed in external vocabulary definitions. There is no easy way of checking that the subprocedure in a call is actually defined at compilation time. So you must be vigilant about run-time error messages, which can be easy to miss in the diagnostic output produced by PyElly. If you see a parse tree node with a cognitive semantic weight of -99, this is likely to be from an undefined subprocedure.

5. Whenever a grammar or a vocabulary is complex, something will almost always be wrong with it. Check underlying analyses to make sure the PyElly is really doing what you expect; it is not enough for just the final output to look correct. Watch out especially for ambiguous phrases with the same plausibility score.

6. You need to test thoroughly any new language definition. Always collect sentences plus their expected translations to cover each grammar rule and each dictionary entry in a language definition. Repeat some sentences in a test set to check for possible variations in the resolution of ambiguities depending on context.

7. In natural language processing, you have to be systematic and committed for the long haul; always rerun your test sets after any significant change in language definition rules. Slight changes in your language definition rules can result in PyElly being unable to translate test sentences it previously handled well. Always check as soon as possible after adjustment of rules; waiting here can only multiply your

problems, which then can be quite difficult to sort out. Be especially careful if you change a `default.*.elly` file because it will affect more than one application and may cause obvious problems in only some of them.

8. Being able to rewrite *n* sentences properly will not guarantee that you can handle the next new sentence. To increase your chance of success here, we have to keep challenging an application with a larger and more diverse sets of target sentences. The `marking` example application can give you an idea of how hard you have to work here. It currently has been successful with almost 14,000 real-world sentences, but it is still easy to find new sentences where its rules will fail. Natural language is complex and hard to model regardless of the approach taken to process it.

9. You may have to scrap some earlier language definitions to make a clean start on handling especially problematic language features. Learn from failures and always be ready to change your processing strategy if you hit a roadblock. PyElly rules can surprise you in many ways.

13.4 Making PyElly Work for You

As a rule-based system based on simple kinds of rules, PyElly requires that you approach natural language in a detailed way. This will often be hard. Some people might want to take only a high-level view of language and let the details be handled by some automaton, but this is not how humans acquire language in the real world. A child learns sentence by sentence in particular contexts and then is taught only much later that various general rules might be at work there.

PyElly is set up to let you work closely with such rules, but will not infer them by itself. Compiling the needed rules for a particular application can seem formidable when starting from scratch, but any educated person is already an expert on his or her native language. PyElly provides a computational framework for codifying this knowledge as rules, which can then be saved and shared. You may be unable or unwilling to do the codification, but other people who enjoy digging into details can take care of it.

An adult learning a second language soon discovers how hard it is to achieve even a minimum level of fluency. The same is true for building a non-trivial natural language application, even in the age of Siri, Watson, and Google Translate. The problem is always in the details. If you are willing to tackle them, then tools like PyElly can take you a long way to credible functionality. This is not the only path to NLP, but it is consistent with how we humans relate to natural language and is well worth emulating.

14. PyElly Applications

PyElly is a collection of basic natural language processing capabilities developed over many decades. These address various low-level details of language in general and can be useful as tools to support the building of higher-level applications. Understanding natural language is difficult, but we can simplify matters by dividing the problem into smaller parts to be tackled separately.

A PyElly application will typically combine selected low-level tools to replicate various kinds of specific natural language expertise. Many of these historically were milestones in computational linguistic research, and so it is a hopeful sign of progress when even high school students can replicate them using PyElly or some other package.

PyElly can implement all the applications previously developed with its predecessors and also can be used for a wide variety of new applications, both simple and complex. None of these really require PyElly, but it is significant that we can readily handle them all within the paradigm of rewriting one language into another according to user-provided rules. PyElly is versatile; it is no one-trick pony.

14.1 Current Example Applications

The example applications included in the PyElly distribution package fall into two classes: those for debugging and validation only and those for potentially useful NLP functionality. PyElly integration testing consists of running all debugging and validation applications plus all functional applications on preset test input files and checking that the results are as expected (See Appendix D).

Below, we describe each current PyElly example application and list their language definition files, indicated briefly by their PyElly file-type designators like `.g` or `.v`, or `.stl`. These definition files are in the subdirectory `applcn` of the unpacked PyElly distribution package. You can look at the rules there to see how to develop your own for new PyElly applications.

Seven example applications are just for support, debugging, or validation:

> **`default`** (`.g`,`.m`,`.p`,`.t`,`.ptl`,`.stl`,`.sx`,`.t`,`.v`) - not really an application, but a set of language definition files that will be substituted by PyElly if your particular application does not specify something explicitly. These include general rules for morphological stemming (`.stl`), basic stop punctuation exceptions (`.sx`), and vocabulary rules (`.v`) covering most of the terms in WordNet 3.0. You can override defaults by providing your own rules, which can be an empty file.

```
input:  - -
output: - -
```

echo (.g,.m,.p,.stl,.v) - a minimal application that echoes its input after being analyzed by PyElly into separate tokens. By default, it will show inflectional stemming of English words and do basic number transformations. You can disable that stemming in your ellyMain.py command line (see Section 7) and editing the .stl file to remove rules for the English comparative endings -er and -est.

```
input:  Her faster reaction startled him two times.
output: her fast -er reaction startle -ed him 2 time -s.
```

test (.g,.m,.p,.ptl,.stl,.t,.v) - for testing with a vocabulary of mostly short fake words for faster keyboard entry; its grammar defines only simple phrase structures with a minuscule vocabulary. This essentially replicates most of the testing done to validate PyElly through its early alpha versions and also some after introduction of new modules to extend PyElly capabilities.

```
input:  nn ve on september 11, 2001.
output: nn ve+on 09/11/2001.
```

stem (.g,.m,.p,.v) - to check specifically that PyElly English inflectional stemming is properly integrated with both internal and external vocabulary lookup. Such stemming happens across multiple PyElly modules, which requires integration testing to detect possible interaction problems.

```
input:  Dog's xx xx xx.
output: dog-'s xx xx xx.
```

fail (.g,.m,.p,.v) - to check that PyElly generative semantic FAIL command is properly implemented. A failure will trigger some complicated context switching, which has to be coordinated across multiple modules.

```
input:  xyz.
output: generative semantic FAIL
        generative semantic FAIL
         /Test 1/== aaxyz
```

The sample output here shows two warning messages from PyElly that the generative semantic FAIL command has been executed, causing a translation to back up to a prior point of ambiguity in a sentence analysis. This eventually should lead to the output as shown above.

bad (.g,.h,.m,.n,.p,.sx,.v) - deliberately malformed language rules to test PyElly error detection, reporting, and recovery. No grammar or vocabulary table will be generated here because of the malformed rules, and so PyElly can do no translation. The files bad.main.txt and bad.main.key are supplied for use with doTest, but are both empty.

```
input:  - -
output: - -
```

The integration test here is just to verify that PyElly can read in ill-formed language definitions without crashing.

zhong (`.g`,`.m`,`.p`,`.stl`,`.v`) - check basic PyElly handling of simplified Chinese input text, which typically will omit spaces between characters and after punctuation. This will require different tokenization than when processing alphabetic languages.

```
input:  这辆汽车不秆驶。
output: (这)(辆)[[汽车]](不)[[秆驶]]。
```

Only a trivial translation is done. Known words (詞) have their characters grouped in double brackets, while unknown characters are individually marked by parentheses. All punctuation remains unmarked.

A second, more substantial, class of example applications are mostly derived from various demonstrations developed for PyElly or its predecessors. These have nontrivial language definitions, show a broad range of PyElly capabilities, and provide a basis for extensive integration testing of PyElly. They range from skeletal prototypes to nearly functional capabilities. You can flesh them out for your own purposes by adding in your own vocabulary and grammar rules.

indexing (`.g`,`.p`,`.ptl`,`.v`) - removes purely grammatical words (stopwords) and does stemming, morphological analysis, and dictionary lookup. By obtaining roots of content words from arbitrary input text, it can predigest input English text for text searching, statistical data mining, or machine learning systems. This application was written for PyElly. It uses `default.stl.elly`.

```
input:  We never had the satisfaction.
output: - - - - satisfy -
```

Note that PyElly will replace each non-content word and each broken-out word prefix or suffix from its input with a hyphen (-) in its output to give an idea of the extent of original text rewritten by PyElly.

texting (`.g`,`.m`,`.p`,`.ptl`,`.stl`,`.v`) - with a big grammar and a special-case vocabulary plus nontrivial generative semantic procedures. This implements a more or less readable text compression similar to what we would see in mobile messaging. This was a demonstration written originally for the Jelly predecessor of PyElly and shows how full-fledged texting compression might be organized.

```
input:  Government is the problem.
output: govt d'prblm.
```

doctor (`.g`,`.m`,`.ptl`,`.stl`,`.v`) - This has a big grammar taking advantage of PyElly ambiguity handling to choose one of several scripted responses for user input containing specific keywords. It uses the PyElly `...` syntactic type to set up grammar rules to emulate Joseph Weizenbaum's Doctor program for Rogerian psychoanalysis, adapting the full keyword-based script published by him in 1966.

The language definition rules were first written for the nlf predecessor of PyElly and then adapted for Jelly and now PyElly.

```
input:  My mother is always after me.
output: CAN YOU THINK OF A SPECIFIC EXAMPLE.
```

chinese (.g,.m,.p,.ptl,.v) - a test of PyElly Unicode handling, translating simple English into either traditional [tra] or simplified [sim] Chinese characters. Both the grammar and the vocabulary of this application are still in progress.

```
input:  they sold those three big cars.
output: [sim]他们卖了那三辆大汽车.
output: [tra]他們賣了那三輛大汽車.
```

In actual operation, only one form of Chinese output will be shown at a time. You get traditional character output when ellyMain.py is run with the flag -g tra . The default is simplified output with the option -g sim. The current integration test is run with traditional characters. Work on this application started in Jelly, but this was greatly expanded in PyElly where most of the rule development has been done.

querying (.g,.m,.ptl,.stl,.v) - heuristically rewrites English queries into SQL commands directed at a structured database of Soviet Cold War aircraft organized into multiple tables. This is a reworking of language definition files for the very first nontrivial application written for the PARLEZ and AQF predecessors of PyElly; it was updated in PyElly to produce SQL output.

```
input:  how high can the foxbat fly?
output: from Ai a,AiPe b
        select ALTD
        where NTNM=foxbat,a.NTNM=b.NTNM
        ;
```

Table and field names in the demonstration are abbreviations: Ai is "aircraft," AiPe is "aircraft performance, ALTD is "altitude," NTNM is "NATO name," and so forth. The original AQF system aimed to make such cryptic names transparent to database users as well as to hide the mechanics of proper query formation.

marking (.g,.m,.p,.v) - rewrite raw text with shallow XML tagging, a canonic PyElly application. This shows how a data scientist might preprocess unrestricted raw text for deeper mining. It has evolved to have the most complex grammar and vocabulary rules of all PyElly example applications by orders of magnitude and is especially notable in its extensive cognitive semantics.

```
input:  The rocket booster will carry two satellites into orbit.
output: <sent>
        <nclu><det>the</det><noun>rocket booster</noun></nclu>
        <vclu><aux>will</aux><verb>carry</verb></vclu>
        <nclu><num>2</num><noun>satellite -s</noun></nclu>
        <nclu><prep>into</prep><noun>orbit</noun></nclu>
        <punc>.</punc></sent>
```

This application is still evolving in its language definition rules. It is the biggest part of the PyElly integration test suite with about a thousand sentences from the Worldwide Web. Such test data has been quite challenging and should give you a better appreciation of the complexity of processing any natural language. See Appendix F for more discussion.

`name` (`.g`, `.m`,`.n`,`.p`,`.ptl`,`.stl`,`.v`) - identify personal names in part or in whole within raw text both by lookup and by inference. This capability could eventually be merged with other example applications like **`marking`**, but it has been kept separate as an integration test validating the special name recognition modules. It became available in PyElly entity extraction as of release v1.1.

```
input:  John Adams married Abigail Smith of Weymouth in 1764.
output: "John Adams"
        "Abigail Smith"
```

The application uses a rule table based on 2010 U.S. Census data. It recognizes the 1,000 most common surnames, the 900 most common male names, and the 1,000 most common female names; and it can infer other name components from context and from their similarity to known names. This heuristic logic is still experimental; it is a reworking of a name searching demonstration built in the 1990's using C.

`disambig` (`.g`,`.h`,`.stl`,`.v`) - disambiguation with a PyElly conceptual hierarchy by checking the semantic context of an ambiguous term. This is only a first try and still crude compared to the other applications listed here. It was written mainly as an integration test focusing on cognitive semantics. Its output is a numerical scoring of semantic relatedness between pairs of possibly ambiguous terms in its input and showing the generality of their intersection in a conceptual hierarchy. Output will show with the actual WordNet 3.1 concepts assigned to input words by PyElly.

```
input:  bass fish.
output: 11 00015568N=animal0n: bass0n/[bass] fish0n/[fish]
```

This uses the PyElly output option to show the plausibility of a translation along with the translation itself, which is right of the second colon (`:`) in the output line. The plausibility will be left of that colon. The output score seen above is 11, which is high, and the output also includes the concepts associated with a sentence analysis. The example above shows that the intersection of `bass` and `fish` is under the WordNet concept `00015568n`, which bears the label `animal0n`.

chemic (.g,.m,.p,.ptl,.stl,.v) - identify chemical nomenclature within raw text both by syntax and by morphology. The rules were developed from lists of chemical names compiled by the American Chemical Society (ACS) as examples of major issues involved in the proper indexing of technical publications.

```
input:  how α³-chloro-2,3-furandimethanol,α²-acetate is.
output: how ⟦α³-chloro-2,3-furandimethanol,α²-acetate⟧ is.
```

The output puts Unicode double brackets ⟦ ⟧ around any chemical name found. This is still an evolving application and works so far only for just over a hundred input examples. Nomenclature is a much harder problem than one might first think.

minimarking (.g,.m,.p,.v) - a simplification of the marking example application. It drops XML tags from output and flattens the levels of structures identified. This was defined with a subset of the grammar rules of marking, but with semantic procedures replaced. It can work with the other marking language definition files, an example of reusing language rule files to build new applications.

```
input:  The rocket booster will carry two satellites into orbit.
output: np ⟦the rocket booster⟧ vp ⟦will carry⟧
        np ⟦2 satellite -s⟧ np ⟦into orbit⟧ .
```

The input shown here is the same as for the marking example application above. The output identifies basic text elements with double brackets ⟦ ⟧ , but labels only simple noun and verb groups. Problematic usages of English words like THAT, TO, and FOR will be tagged to reduce ambiguities in further stages of language processing beyond PyElly.

ngram (.g,.m,.p,.ptl,v) - analyze words according to a finite set of word fragments in them.

```
input:  orthogonal axes.
output: orth tho hog oga gan ax xi ix
```

Fragmentary indexing can effectively capture similarity and dissimilarity between pairs of documents even we may be unable to reconstruct all their original words from their word fragments.

These eleven nontrivial example applications show just some of the simple translations of natural language that can be done. All the processing is still rudimentary—no deep learning or detailed world models, but it calls for many kinds of linguistic competence. You can accomplish much with text data without turning to deep artificial intelligence.

14.2 Building Your Own Applications

The PyElly example applications described above are just a few possibilities for NLP short of full understanding. They cover many levels of linguistic competence, ranging from simple recognition and breakdown of words to detailed markup of complex sentences. These basic capabilities are only like a baby learning how to crawl, but one has to start somewhere.

We can continue to make progress just by adding more rules to existing PyElly applications and collecting broader samples of input for them to process. A bigger challenge, however, is in developing entirely new applications that will stretch our linguistic competence. We have yet to reach any hard limit with PyElly on what basic NLP can accomplish.A new application for PyElly should meet the following conditions:

1. Your input data can be UTF-8 Unicode text divisible into sentences consisting of ASCII, Latin 1 Supplement, Latin Extender A and B, and special punctuation characters. As of PyElly v1.6 , it can also be Chinese text.

2. Your intended output will be translations of fairly short input sentences into arbitrary Unicode text in UTF-8 encoding, not necessarily into sentences.

3. No world knowledge is required in the translation of input to output except for what might be expected in a standard dictionary or glossary.

4. You understand broadly what your translation process involves and mainly need some support in implementing its details.

5. Your defined vocabulary is limited enough for you to specify manually with the help of a text editor like vi or emacs, and you can tolerate some terms being treated as the `UNKN` syntactic type.

6. Your computing platform has Python 3.8 installed. This will be needed both to develop your language rules and to run your intended application.

7. You are comfortable with trial and error development of language definition files and are willing to put up with idiosyncratic non-commercial software.

You will have to formulate the various rules to define the PyElly translations you want. In the extreme case, for a PyElly application `A`, you can have up to ten language definition files: `A.g.elly` (grammar), `A.v.elly` (vocabulary), `A.m.elly` (macro substitutions), `A.p.elly` (patterns for syntactic types of words), `A.n.elly` (name components), `A.t.elly` (multiword templates), `A.ptl.elly` (prefix removal rules), `A.stl.elly` (suffix removal rules), `A.h.elly` (semantic concept hierarchy), and `A.sx.elly` (stop punctuation exceptions). To start out with, you can get by with just `A.g.elly` and leave the other files empty or just take the defaults for them.

So, where can we go from here? There are many possibilities, but a good strategy is to focus on building new applications with challenging kinds of input text data. This will

stress PyElly and expose its weak points. For historical reasons, PyElly was developed mostly on elementary school English input but a true NLP tool needs to aim wider. Here are some ideas on a logical progression of linguistic competence.

14.2.1 Other Simple Applications

As a teaching aid, PyElly could never have too many applications that students could finish in only a week or so. The PyElly distribution package includes some simple ones that could be reimplemented, but it is easy to come up with new ones. For example,

- A basic bowdlerizer to replace vulgar terms in text with sanitized ones or creative punctuation.
- A translation from English to pig Latin or some other "secret" kid language.
- Redaction of sensitive data from text: names, telephone numbers, addresses, and so forth, perhaps writing out bars of Unicode black large square characters ■ (=U+2B1B) instead.
- Transliteration of English words or names into a non-Latin alphabetic or a syllabic or an ideographic representation. This is the simplest possible kind of language translation, but may still require a bit of foreign language expertise.

All of this is legitimate NLP, which you can do without a convolutional neural net. Let your imagination run free here.

14.2.2 More Diverse English

There are many variations of English found in special domains like sports, politics, finance, literature, the arts, fashion, crime, terrorism, celebrity gossip, and so forth. These are still recognizable as English, but have their own vocabularies and grammars, often with the apparent purpose of making it hard for outsiders to understand what insiders are saying. PyElly can help to translate and clarify such jargon.

A PyElly application reading an English sublanguage will probably require the formulation of many new grammar and vocabulary rules, though few if any changes to PyElly code. These would be projects taking up to a month to complete. For example,

- A decoder for business jargon or the current dialect of teenspeak.
- Rewriting verbose English into concise English, correcting common misspellings and removing clichés. There are now commercial apps that do this for you, but PyElly is free and allows for easy customization.
- Simplify complex text by aggregating phrases into black box units for a linguistic analysis. For example, in a medical article, we could identify disease names, drug names, names of researchers, names of institutions, genetic information, and other entities. We would replace them in text with generic markers (e.g. <DRUG NAME>) to

make it easier to figure the structure of sentences. The markers would have a pointer back to the original text to allow us to recover its information later as needed. This procedure could be expanded into a much broader scope.

- Analyzing text at the phrase level to score the similarity of two documents, perhaps to assess the possibility of plagiarism. This can be done just by counting simple function words like THE, AND, and OF; but we can reduce the noise in our statistics by looking at more complex text features requiring morphological and syntactic analysis of text.

You might also try building on existing example applications, like **`marking`** and **`name`**, perhaps combining them. With any new input text, however, you can expect to have to add many new rules and maybe modify old ones. It might even be necessary to change ta `default.*.elly` definition file. If so, make sure that other applications relying on it will still work. As a test, delete every saved `*.elly.bin` file to force PyElly to rebuild their language definitions. Then rerun all your integration tests.

14.2.3 Western European Languages

PyElly currently can recognize characters from the first four Unicode blocks plus a few extras as input text. These include ASCII, common letters with diacritical marks like é and ü and legacy characters like æ, œ, ß, Þ, and ð. In theory, this would allow you to process French, Spanish, German , Czech, Hungarian, and other Western European languages as well as Pinyin for Chinese.

Much of the power of PyElly, however, is in its support of English, which becomes irrelevant with a different language, even one written with a similar alphabet. In particular, the English inflectional stemmer, morphological analysis, recognition of dates and times, expected form of personal and organizational names, and even punctuation rules all may need to be replaced.

The loss of these capabilities is not fatal. This just means more work has to be done in defining the rules for a particular language of interest. Given enough time, this should be fairly straightforward. It all depends on how comprehensive you want to make your final application. A basic demonstration of a processing capability in your language should be reasonable for a term project.

14.2.4 Languages Written in Non-Latin Alphabets

PyElly cannot directly handle input languages like Russian, Greek, Arabic, Hindi, Thai, or Korean. Its character definitions currently recognize only a particular subset of Unicode as being alphabetic and also identify them as consonants or vowels. It is easy enough to add, say, Cyrillic characters to its current mix, but unless you have a definite continuing interest in Russian, you probably would be better off just transliterating your input into a Latin alphabet outside of PyElly with your own text filter.

For example, the Buckwalter transliteration maps Arabic text to plain ASCII in a way that is reversible. In this way, PyElly can process Arabic without worrying about which direction to read it, how to handle indications of short vowels, or the variation in form of Arabic letters depending on their position in a word. You only have to obtain a coding of the Buckwalter transliteration and pipe its output to the standard input of PyElly.

This will put you into a situation somewhat like what we had above for processing non-English Western European languages. It will not be as easy because languages written in another alphabet probably differ in many ways from English other than the forms of letters. This is the perennial challenge for humans in learning a foreign language, and it cannot be disregarded by automated NLP systems.

14.2.5 Ideographic Languages

Chinese text can be transliterated in multiple ways into a Latin alphabetic form. The most common nowadays in pinyin, but this is hard to map back to Chinese ideographic characters. In automatic processing of Chinese, almost everyone prefers input data to be ideographic, which requires that an NLP tool targeting Western European languages change its methods for recognizing individual tokens in text and delimiting sentences.

The solution in PyElly was to implement a different mode for processing Chinese text, to be specified as a command line flag for the the top-level ellyMain.py module that starts up PyElly. This involved changes in ten Python source files, but most of them turned out to be minor. As of v1.6, PyElly will accept simplified and traditional Chinese input, although this has yet to go through any extended testing.

We now want example applications to exercise PyElly's Chinese capabilities. First up probably will be a parser able to handle all the answer sentences in the final examination section of the book "Chinese Demystified" by Claudia Ross (McGraw Hill, New York, 2011). Try out your own ideas here.

15. Going Forward

PyElly offers many different tools and resources for practical computational linguistics. Most of them are still evolving in both code and documentation; but the overall Python system has been tested extensively since 2013 and should be fairly reliable for non-critical applications. PyElly has some strict limitations, but within them, you can achieve useful results and learn firsthand about the structure of language.

To approximate how a human understands natural language, you must deal with its full complexity. Using simplified models of text can expedite some kinds of processing, but we often throw away too much information here. Text as yet cannot be reduced automatically into a clean and logical encoding of information. To dig into text data and mine it more effectively, we do really need language expertise like PyElly at some point.

15.1 What PyElly Tries To Do

PyElly is old-fashioned in its use of manually crafted rules to describe text data for rewriting. This contrasts with current NLP practices favoring unsupervised automatic machine learning of language structure from large unlabeled samples of text data. Machine learning does reduce human labor; but a rule-based approach can still be advantageous if someone else has already laid out most of the rules in a usable form.

The challenge here is that natural language operates at many different levels: orthography (spelling), morphology, lexicography (parts of speech), syntax, semantics, and pragmatics (intended meaning). Competence in processing text at higher levels is the Holy Grail, but it depends on achieving competence first at lower levels, where a system like PyElly is strong. If we want to build higher-level NLP systems quickly, we have to avoid reinventing all the low-level wheels in every project.

Creating a PyElly language definition is a non-trivial problem, given the sheer amount of knowledge required to deal effectively with a language like English. Earlier efforts in building rule-based systems for artificial intelligence foundered because the number of rules for real-world applications soon became unmanageable. PyElly tries to avoid this fate by focusing on low-level linguistic knowledge, which is still difficult, but has been well studied by linguists and can be compiled incrementally.

In particular, PyElly provides various integrated prebuilt capabilities like English inflectional stemming and morphological analysis, American name recognition, and sentence delineation. None of these are "game-changing" technologies, but they can provide critical basic support. Even when your primary analysis will rely mainly on machine learning, some linguistic awareness about your data can greatly reduce the dimensionality of your problem space and speed the finding of solutions.

For example, take issues like punctuation, capitalization, and diacritical marks in raw text data. A data scientist might be tempted to ignore such details, but some author had a reason to do the work to put them into text. They can help to break up text into sentences and to determine the roles of words in text. Humans can then tell if a word is

being used denotatively in a name like "Tendersweet Clams™" or connotatively in a description like "sweet potato." Such low-level competence can add up to make it easier to frame a big picture of how language works.

In a sense, we can be seduced by how much we can accomplish with simplistic processing strategies. Treating text as a bag of words is often good enough for a basement level of understanding; but if we want to be smarter about language than an eight-year-old, then we have to buckle down. A neural net will hide many confusing linguistic details from us, but if English, Japanese, Arabic, Mayan, and Turkish all then look more or less the same to a neural net, our approach may be too myopic.

In processing English, knowledge about its -S, -ED, and -ING inflectional endings can help in parsing it more effectively. Add in all the other kinds of knowledge built into PyElly or captured by PyElly rules supplied by users, and we can quickly get ahead of the game in NLP. The extent of such rules will be much wider than most people expect. Otherwise, why do most non-native speakers still make so many mistakes in English even after many years of immersion in it?

PyElly will let users control what aspects of their text data to focus on. For example, to process text from a large corpus of recipes, we might want to identify beforehand basic elements such as equipment, measurements, cooking techniques, common ingredients, and descriptions of texture and taste. This can greatly reduce noise in deep learning so that we can avoid getting terms like "one tsp" as a major feature for distinguishing between recipes. In NLP, we should never be or act dumber than we have to be.

Any PyElly application will of course rewrite its text input imperfectly, but even this can be a significant improvement over completely ignoring the structure of the text data. If a PyElly translation needs to improve, we can always add rules to address specific deficiencies. You can also change PyElly code itself. In line with modern software engineering, no real code or system will ever be completely finished.

Remember that PyElly applications are never meant to pass any kind of Turing test. PyElly works with minimal world knowledge and lacks full reasoning and common sense, limiting its degree of language understanding. This sounds like a handicap, but it means that PyElly can focus on what it is good at. Think of it as being like a high-end food processor in the kitchen, which can make a skilled chef more productive, but will by itself be no guarantee of Michelin three-star cuisine at every meal.

PyElly has already gone far beyond its original PARLEZ roots. If its development can continue, we would hope to improve various existing capabilities like entity extraction, handling of semantic concepts, and name recognition. Underlying PyElly algorithms could also be better tuned for efficiency and resilience, especially in ambiguity resolution. Eventually this should all lead to a successor to PyElly, for we are still learning about natural languages and how to process them better.

A major breakthrough would be welcome of course, but PyElly and its predecessors have been more about continually accumulating and integrating various kinds of basic linguistic competence over time. Experience gained in building more applications will

guide PyElly development in new directions. Progress may seem slow, but we are engaged with NLP for the long term; and whatever rules we learn stay learned.

15.2 Practical Experience in PyElly Usage

The PyElly `marking` example application can give you an idea of what to expect in a big NLP project. It started out as a way to test the limits of PyElly on a simple problem: a shallow XML markup of actual English text from the Worldwide Web. At the outset, it was unclear how best to attack this problem and how well PyElly could handle the complexity of "wild" input. Shallow markup seemed likely to require at least an order of magnitude more rules than any previous PyElly example application.

Appendix F describes how the `marking` application evolved. Our general interest here will be on just the results of that effort. As with most PyElly applications, the basic strategy was to start with a simple grammar and vocabulary and then extend the rules to handle an ever larger sample of sentences. Any set of PyElly rules will often fail on unfamiliar input, but by adjusting and expanding our rules, we can usually get an acceptable markup even for a big set of hard sentences. The catch, however, is that any rule changes have to preserve any previous successful markups.

The `marking` language definition started out with only 132 grammar rules, 161 vocabulary rules, 28 pattern rules, and 0 macro substitution rules. These had been checked out beforehand with some simple input before moving on to real Web text. The first data sample from the Web for processing consisted of segments of text taken from 22 different sources totaling 129 sentences altogether. Every segment was processed as continuous input in which PyElly had to figure out the sentence boundaries.

The hope was that PyElly would be flexible enough to let us to produce acceptable markups eventually for every test sample through enough rule adjustments, possibly highly specific to problem sentences. This exercise was also useful in uncovering many previously undetected problems in PyElly's Python code; the hope was that both PyElly and the `marking` application in particular would keep improving.

Eventually, PyElly did process all 129 sentences of the first Web sample acceptably. This showed that PyElly was capable of heavy-duty work, although the viability of a general markup application remained in doubt. More testing was needed, and so another five samples with 759 sentences were eventually collected from eighty new Web sources. These took about another two years to mark up successfully. Over that time, the `marking` language definition grew to 612 grammar rules, 6,157 vocabulary rules, 97 pattern rules, and 50 macro substitution rules.

The `marking` application remains unfinished. The failures when processing the last segment of the sixth sample of expanded data for the first time were similar to those for the first segment of the first sample. Nevertheless, the PyElly shallow markup rules were getting better and so it seemed reasonable to collect even more data from the Web to see how far we could go. This led eventually to the gathering of almost 13,000 real-world sentences from Google news text.

Natural language now seems to be too complicated for complete success in shallow markup anytime soon, but PyElly has now marked up all the news text data— no minor feat. And the set of `marking` language definition rules has kept growing, and PyElly itself has kept evolving. The current `marking` rule count is 726 grammar, 39,538 vocabulary, 148 pattern, and 73 macro. As can be seen, the biggest growth over time has been in vocabulary rules.

The `marking` example application has shown that PyElly handle a large number of grammar and vocabulary rules; and it does need all of them. The loading of those rules from definition files has so far been manageable for PyElly, and processing with them has yet to push PyElly anywhere near a breaking point. Sentences in the latest news data now take an average of about 2 seconds for markup on a laptop, although most sentences can be processed in less than a second. This performance seems reasonable for Python code running interpretively without concerted optimization.

On the whole, the results of the markup project have been positive. It has helped to uncover problems of understanding English text and also to test out PyElly code more thoroughly. We now know that a purely combinatoric approach to ambiguity runs into difficulty with long sentences having many unknown tokens; that will remain a big challenge in the future. We have shown that PyElly can operate beyond a sandbox and could reasonably claim to have achieved shallow markup with a failure rate of fewer than one out of ten sentences in real-world news text never seen before.

There is plenty of work left to do. One direction going forward is to feed still more Web text to the `marking` application until we hit some kind of hard wall. The actual details of markup also need to be checked out fully; so far, the priority has been just to get a markup for every sentence in our news data sample by adjusting our language rules to get past any PyElly parsing failure. With more data, the PyElly `marking` application should keep improving and PyElly should gain credibility as an integrated NLP toolkit.

15.3 Where We Now Stand

PyElly remains a work in progress. Its original ancestor over forty years ago was a fairly simple syntax-driven rule-based system for language analysis, but practical usage in many different applications has pushed it in new directions. Just within PyElly, old rule types became more complex, and new rule types emerged. We have yet to exhaust the possibilities for language competence short of full understanding, however; and computational linguistics still has room for grammars and such to evolve.

The PyElly open-source package so far remains compact, however, with only about 11 thousand lines of code. Even with all the language definition files of the various example applications, it is easy to download and run on a regular home computer. Unless you want to go to extremes with big data, no high-performance computing is required beyond what is sufficient to stream HD video. The PyElly software is also free and unencumbered, which should fit in well with student projects in schools. This is natural language for the people—no Ph.D. or M.B.A. needed.

PyElly does operate at a fine level of detail, which may put off many people. Many researchers still prefer to hide the linguistic complexity of their text data, though this severely limits their options in processing it. Familiarity with rules for the structure and mechanics of natural language can help students and experimenters alike to see better the big potholes on the road to NLP and maneuver around them.

PyElly requires the coordination of many different kinds of language rules to produce a coherent result. This is what makes any natural language "natural." No one sat down to design a language like English, Chinese, Amharic, or Quechua from clean first principles. These instead all grew undirected over thousands of years driven by local needs and circumstances. We should not expect that any set of rules for describing the product of such a prolonged process will be simple and short.

A recent book, "Babel" by Gaston Dorren, Atlantic Monthly Press (New York), 2018, surveys the 20 most spoken languages in the world and lays out the main difficulties that humans would face in learning them as an adult. These would be major obstacles for machine learning as well, and the problems differ from language to language, so that mastering one language can help only a little in taking on another. Natural language is complicated, and a rule-based approach has to approach such complexity head on.

Having a mountain of rules in an application is workable if most of them can be kept simple. Such rules will be easier to formulate as well as to debug. Some PyElly rules sometimes must get quite complicated, but in PyElly applications so far, we have been able to make most of them general and simple and then augment them with a few specific rules to deal with complex cases.

The rule-based framework of PyElly seems to hold up at least to a level of hundreds of grammar rules and tens of thousands of vocabulary rules. This remains true even after adding other rule types for token recognition, macro substitution, morphology and other kinds of linguistic analysis. Such knowledge can be a chore to compile for the first time, but when properly integrated, linguistic expertise should be transferable for a given language and give us a head start in the building of ambitious applications for it.

We could easily spend another forty years just tinkering with various computational linguistic tools in PyElly. In the 1980s, artificial intelligence researchers thought that they had everything needed to understand natural language within a decade or so, but no one should be so confident today, even with deep machine learning. A more achievable agenda is for us just to continue making progress through sustained small efforts along with the occasional sudden inspiration. Everyone can help to blaze new trails here; and the more people at work on it, the better.

The main takeaway here is that, for any nontrivial natural language processing, success will call for paying some serious attention to details. Our own human skills for language understanding are extraordinary and need to be matched somehow in our NLP algorithms. The `marking` samples of real-world text data show clearly how even shallow text analysis will require nontrivial knowledge of both a target language and its world of

discourse to resolve ambiguities. Linguistic expertise is far from being obsolete yet for serious data scientists trying to wring the most value out of voluminous text data.

Yes, an unsupervised neural net in theory can learn everything automatically with enough training data, but one would have to re-create the equivalent of the twelve years of exposure to language that a human child gets in the home, the streets, schools, and the media around us. Except in cases of child abuse, much of that learning for humans also involves extensive adult supervision; a neural net "child" deserves no less.

We may never find a magic bullet for NLP. Lack of knowledge about the inner and outer workings of language is not the problem, however. As an educational tool, PyElly is about developing and packaging our linguistic competence and then turning researchers loose to experiment and explore where such expertise can come into play. In most cases, PyElly code is mostly unremarkable, but it is nontrivial and has been around the Horn a few times already.

Bear in mind that the architecture of PyElly grew in large part out of happenstance and is in no way optimal. Nothing about it is sacrosanct; its components reflect only what was necessary to make PyElly work passably with specific example applications. One could easily imagine completely different configuration of components for capturing useful linguistic knowledge. If that is your bent, then pursue your own bliss here.

The tools of PyElly do exist, however, Try them on your favorite text corpora and see whether a bit of competence can advance your data analysis and data mining. Share with others any new PyElly rules you develop or old rules you have refined. Any criticisms or suggestions about PyElly itself are welcome, and of course, you may freely improve on the PyElly open-source Python code and move it in your own directions. PyElly v2.0 rewrote PyElly v1.6 in Python 3.8, which is a significant milestone.

Natural language processing still has room for some enterprising computational linguists and their ideas. Robots have yet to take over everything, although trying to compete head to head with them is probably futile in the longer run. PyElly shows one plausible way that accumulated human knowledge about language can complement what robots with deep neural nets can accomplish. You might also decide to build your own comprehensive natural language processing tools here. The world lies wide open before us. Let us boldly go out into it!

Appendix A. PyElly Implementation in Python

This appendix is for Python programmers. You can run PyElly without knowing its underlying implementation, but at some point, you may want to modify PyElly or embed it within some larger information system. The Python source code for PyElly is released under a BSD license, which allows you to change it freely as needed. You can download it from

`https://github.com/prohippo/pyelly.git`

PyElly was written in Python 3.8 under Mac OS X 10.9 and 10.10; it will not run under earlier versions of Python because of changes in the programming language. To implement its external vocabulary tables, PyElly v1.2+ no longer requires the Berkeley Database (BDb) database manager or the bsddb3 third-party BDb Python API wrapper. The PyElly code with BDb in release v1.1 or earlier should be discarded because a GPL copyleft license would be required for it, greatly limiting freedom of use. Vocabulary table lookup code has also evolved greatly in PyElly v1.2 and v1.3.

Currently, the PyElly v2.2.1 source code consists of 69 Python modules, each a text file named with the suffix `.py`. All modules were written to be self-documenting through the standard Python `pydoc` utility. When executed in the directory of PyElly modules, the command

```
pydoc -w x
```

will create an `x.HTML` file describing the Python module `x.py`.

The code was written neither for speed of execution nor for space efficiency. This is normal in Python development practice, however, because it is an interpreted language, and it is consistent with PyElly's emphasis on quickly putting together a broad range of functionality for doing useful language processing right now. Although the code has become fairly stable through extensive testing, it remains experimental and still keeps many debugging print statements that can be reactivated by uncommenting them.

The algorithms underlying PyElly have become somewhat intricate after decades of tinkering. Section 12 of this manual does describe the bottom-up parsing approach of PyElly, but other important aspects of the system have to be gleaned from source code. The most notable of these are wildcard string matching, macro substitution, the non-deterministic finite automaton for token pattern matching, compiling PyElly code for generative and cognitive semantics, and multi-element vocabulary table lookup.

Here is a listing of all 67 current PyElly Python source files grouped by functionality. Some non-code definition and data files are also included below when integral to the operation or builtin unit testing of the modules there. They are shown in the shaded rows of the tables.

Inflectional Stemmer (English)	
ellyStemmer.py	base class for inflection stemming
inflectionStemmerEN.py	English inflection stemming
stemLogic.py	class for stemming logic
Stbl.sl	remove -S ending
EDtbl.sl	remove -ED ending
Ttbl.sl	remove -T ending, equivalent to -ED
Ntbl.sl	remove -N ending, a marker of a past participle
INGtbl.sl	remove -ING ending
rest-tbl.sl	restore root as word
spec-tbl.sl	restore special cases
undb-tbl.sl	undouble final consonant of stemming result
deinflectedMatching.py	for simple English inflectional stemming in lookups only

Tokenization	
ellyToken.py	class for linguistic tokens in PyElly analysis
ellyBuffer.py	for manipulating text input
ellyBufferEN.py	manipulating text input with English inflectional stemming
ellyBufferZH.py	manipulating input text of Chinese characters
substitutionBuffer.py	manipulating text input with macro substitutions
macroTable.py	for storing macro substitution rules
patternTable.py	extraction and syntactic typing by FSA with pattern matching

Parsing	
symbolTable.py	for names of syntactic types, syntactic features, generative semantic subprocedures, global variables
syntaxSpecification.py	syntax specification for PyElly grammar rules
featureSpecification.py	syntactic and semantic features for PyElly grammar rules
grammarTable.py	for grammar rules and internal dictionary entries
grammarRule.py	for representing syntax rules
derivabilityMatrix.py	for establishing derivability of one syntax type from another so that one can make bottom-up parsing do nothing that top-down parsing would not
ellyBits.py	bit-handling for parsing and semantics
parseTreeBase.py	low-level parsing structures and methods
parseTreeBottomUp.py	bottom-up parsing structures and methods
parseTree.py	the core PyElly parsing algorithm
parseTreeWithDisplay.py	parse tree with methods to dump data for diagnostics

Semantics	
generativeDefiner.py	define generative semantic procedure
generativeProcedure.py	generative semantic procedure
cognitiveDefiner.py	define cognitive semantic logic
cognitiveProcedure.py	cognitive semantic logic
semanticCommand.py	cognitive and generative semantic operations
conceptualHierarchy.py	concepts for cognitive semantics

Sentences and Punctuation	
ellyCharInputStream.py	single char input stream reading with unread() and reformatting
ellySentenceReader.py	divide text input into sentences
stopExceptions.py	recognize stop punctuation exceptions in text
exoticPunctuation.py	recognize nonstandard punctuation
punctuationRecognizer.py	define single-character punctuation defaults for English

Morphological Stemmer	
treeLogic.py	binary decision logic base class for affix matching
suffixTreeLogic.py	for handling suffixes
prefixTreeLogic.py	for handling prefixes
morphologyAnalyzer.py	do morphological analysis of tokens

Entity Extraction	
entityExtractor.py	runs Python entity extraction procedures
extractionProcedure.py	some predefined Python entity extraction procedures
simpleTransform.py	basic support for text transformations and handling of spelled out numbers
dateTransform.py	extraction procedure to recognize and normalize dates
timeTransform.py	extraction procedure to recognize and normalize times of day
nameRecognition.py	identify personal names
digraphEN.py	letter digraphs to establish plausibility of possible new name component
phondexEN.py	get phonetic encoding of possible name component
nameTable.py	defines specific name components
compoundTable.py	miscellaneous multiword entities defined by template rules

Top Level	
ellyConfiguration.py	define PyElly parameters for input translation
ellySession.py	save parameters of interactive session
ellyDefinition.py	language rules and vocabulary saving and loading
ellyPickle.py	basic loading and saving of Elly language definition objects
interpretiveContext.py	handles integration of sentence parsing and interpretation
ellyBase.py	principal module for processing single sentences
ellyMain.py	top-level main module with sentence recognition
ellySurvey.py	top-level vocabulary analysis and development tool
dumpEllyGrammar.py	methods to dump out an entire grammar table

External Database	
vocabularyTable.py	interface to external vocabulary database
vocabularyElement.py	internal binary form of external vocabulary record

Extended Generative Semantics	
extendedContext.py	interface to hard-coded generative semantic sub-procedures
wordFragment.py	basic 2-, 3-, 4-, and 5-gram analysis of words
Test Support	
parseTest.py	support unit testing of parse tree modules
stemTest.py	test stemming with examples from standard input
procedureTestFrame.py	support unit test of semantic procedures
generativeDefinerTest.txt	to support unit testing of generative semantics definition
cognitiveDefinerTest.txt	to support unit testing of cognitive semantics definition
suffixTest.txt	to support comprehensive unit test with list of cases to handle
morphologyTest.txt	to support unit test of prefix and suffix tree logic plus inflectional stemming
sentenceTestData.txt	to support unit test of sentence extraction
zhongwen.txt	to support unit test of Chinese character input
testProcedure.*.txt	to run with the generativeProcedure.py unit test to verify correct implementation of generative semantic operations
*.main.txt	Input text for integration testing
*.main.key	expected output text for integration testing with provided input
GN.txt	daily Google News text samples for markup testing

All `*.py` and `*.sl` files listed above are distributed together in a single directory along with `*.main.*` integration test files. The `*.txt` files for unit testing will be in a subdirectory `forTesting`. The `*GN*.txt` files are in the subdirectory `extra`.

The first v0.1beta version of the Python code in PyElly was completed in 2013 with some preparatory work done in November and December of 2012. This was an extensive reworking and expansion of the Java code for its Jelly predecessor, making it no longer compatible with Jelly language definition files. PyElly v1.0 moved beyond beta status as of December 14, 2014, but active development continues. The latest release is v2.2.1.

The emphasis in PyElly is now moving away from adding on new Python modules and towards improved reliability and usability. Existing modules will also continue to evolve, but mainly to provide better support for building real-world applications to process unrestricted natural language text. This will involve the construction of big grammars and big vocabularies, the ultimate test of any natural language toolkit.

Version 2.2 allows knowledgeable users of PyElly to extend its output rewriting capabilities with code in Python and possibly other programming languages as well. This is implemented through the extendedContext.py module, which currently supports 2-, 3-, 4-, and 5-gram analysis of words with code in the external wordFragment.py module. Other capabilities like simple arithmetic can be supported similarly.

This makes PyElly easily extensible through links to arbitrary string manipulation code. The extendedContext.py module allows PyElly generative semantics to call this code by way of special named subprocedures defined outside of grammar rule definition files. A user can edit extendedContext.py to exploit other external capabilities . These can be shared with other PyEllt users, but will require editing of PyElly code in Python.

Appendix B. Historical Background

PyElly natural language tools have evolved greatly in the course of being completely written or rewritten five times in five different languages over the past forty years. Nevertheless, it retains much of the flavor of the original PDP-11 assembly language implementation of PARLEZ. Writing such low-level code forced PyElly software architecture to be simple, a help in porting the system to later computing platforms.

The PARLEZ system, for example, had its own stripped-down custom language for generative semantics because nothing else was available at the time on a DEC PDP-11. That solution is platform-independent, however, and so has been carried along with only a few changes or additions in systems up to and including PyElly for generating text output. And, sorry, arithmetic is still unsupported.

PyElly departs in major ways even from its immediate predecessor Jelly.

- It is written in object-oriented Python 3.8 instead of Java.
- The English inflectional stemmer has simplified its basic operations and has eliminated internal recursive calling. Stemming logic is now in editable text files loaded at run-time. Its number of rules has greatly expanded.
- Morphological analysis now includes proper identification of removed prefixes and suffixes as well as returning stems (lemmas). This provides a true analytic stemmer, befitting a general natural language tool. Current rules are still only for English, but they have greatly grown, now covering most WordNet "exceptions."
- The syntactic type recognizer was upgraded to employ an explicit non-deterministic finite-state automaton with transitions made when an initial part of an input string matches a specified pattern with both literal characters and wildcards in a state. A special null pattern was added for more flexibility in defining automata state changes.
- New execution control options were added to generative semantics. Local and global variables were changed to store string values, and list and queue operations were defined for local variables. Deleted buffer text can now be recovered in a local variable. Support for debugging and tracing of semantic procedures was expanded.
- Vocabulary tables have become more easily managed by employing the SQLite package to manage persistent external data. External vocabulary rules have been limited to single input lines to facilitate automatic compilation of definitions.
- Support for recognizing and remembering personal names or name fragments is now available in PyElly entity extraction. This is based on 2010 U.S. Census data.
- Templates for recognizing multiword compounds as entities were added.
- A new interpretive context class was introduced to coordinate execution of generative semantic procedures and consolidate data structures for parsing and rewriting.

- Handling of Unicode was improved. UTF-8 is now allowed in all language definition files and in interactive PyElly input and output. Non-data parts of rules remain ASCII.
- Ambiguity resolution was completely overhauled with expanded cognitive semantics. It can now test capitalization, the token and character counts of phrases, and where a phrase starts in a sentence.
- Semantic concepts were added to cognitive semantics for ambiguity handling. This makes use of a new semantic hierarchy with information derived from WordNet.
- Control character recognition in input allows for closer management of the explosion of possible phrase nodes due to ambiguity of natural language.
- Sentence and punctuation processing is cleaner and more comprehensive. Unicode formatted punctuation like " and ". Standard ASCII and Unicode punctuation is predefined in a special Python module.
- Small Greek letters, superscript numbers, and musical symbols are now recognized in input. Various Unicode space characters are also recognized.
- The PyElly command line interface was reworked to support new initialization and rewriting options. This now now allows Chinese characters as input text to process.
- Error handling and reporting has been broadened in the definition of language rules. Warnings have also been added for common problems in definitions.
- New unit tests have been attached to major modules. Integration tests have been set up with current example applications to exercise a broad range of PyElly features.
- New example applications have also been written to show a broader range of PyElly processing. Older example applications from Jelly and earlier systems were converted to run in PyElly.
- Many bugs from Jelly or earlier were uncovered by extensive testing and fixed.

Jelly is now superseded and has been retired.

PyElly is by no means complete and might be rewritten in yet another programming language. The ultimate goal here, however, is less a utopian system than an integrated set of reliable tools and resources immediately helpful for practical natural language processing. Some PyElly tools and resources will seem dated to some; but these have had time prove their usefulness. There is no need to keep reinventing them.

This PyElly User's Manual reorganizes and greatly extends the earlier one for Jelly, but still retains major parts from the original PARLEZ Non-User's Guide, first printed out on an early dot-matrix printer. Editing for clarity, accuracy, and completeness continues. Check https://github.com/prohippo/pyellytoo.git for the latest PDF version.

Appendix C. Berkeley Database and SQLite

Berkeley Database is an open-source database package available by license from Oracle Corporation, which in 2006 bought SleepyCat, the company holding the BDb copyright. BDb was the original basis for PyElly external vocabulary tables, but has been replaced by SQLite because of changes in Oracle licensing policy for BDb.

The current PyElly v1.2+ vocabulary tables with SQLite should incur no noticeable performance penalty despite having to access all persistent data through an SQL interface instead of function calls. SQLite will allow PyElly to remain under a BSD license, since SQLite is included in the Python 2.7.* and 3.* libraries. It does not have to be downloaded separately.

If running with Berkeley Database is really important, you can download it yourself and return to the former PyElly vocabulary table code in v1.1. The latest versions of BDb do come with a full GNU copyleft license, though, with possible unattractive legal implications. The apparent intent of Oracle here is to compel many users of BDb to buy a commercial license instead of using free open-source code.

The Python source of `vocabularyTable.py` in the previous PyElly v1.1 release has NOT been updated, however, to display a GNU copyleft license as required for use of Bdb. That will not happen unless there is a reason to fork off a specific BDb version of PyElly in the future.

Downloading Berkeley Database and making it available in Python is a complex process depending on your target operating system. You will typically need Unix utilities to unpack, compile, and link source code. For background on Berkeley Database, see

http://en.wikipedia.org/wiki/Berkeley_DB

For software downloads, you must go to the Oracle website

http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/index.html

to get the latest Berkeley Database distribution file. The instructions for doing so on a Unix system can be viewed in a browser by opening the Berkeley Database documentation file:

```
db-*/docs/installation/build_unix.html
```

The installation procedure should be fairly straightforward for anyone familiar with Unix. An actual MacOS X walkthrough of such compilation and installation can be found on the Web at

https://code.google.com/p/tonatiuh/wiki/InstallingBerkeleyDBForMac

To access Berkeley Database from Python, you must next download and install the bsddb3 package from the web. This is available from

https://pypi.python.org/pypi/bsddb3

The entire installation procedure turns out to be quite complicated, however, and difficult to carry out directly from a command line. The problem is with dependencies where a module A cannot be installed unless module B is first installed. Unfortunately, such dependencies can cascade unpredictably in different environments, so that one fixed set of instructions cannot always guarantee success.

To avoid missteps and all the ensuing frustrations, the best approach is use a software package manager that will trace out all module dependencies and formulate a workable installation path automatically. On MacOS X, several package managers are available, but the current favorite is `homebrew`. See this link for general details:

http://en.wikipedia.org/wiki/Homebrew_(package_management_software)

As it turns out, `homebrew` will also handle the installation of Berkeley Database and the upgrading of Python on MacOS X to version 2.7.5 (recommended for bsddb3). If you have a MacOS system with Xcode already installed, you can follow these steps to download the `homebrew` package and use its `brew` and `pip` commands to get Berkeley Database:

```
# get homebrew
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

# get latest Python
brew install python —framework

# get BdB
brew install berkeley-db
sudo BERKELEYDB_DIR=/usr/local/Cellar/berkeley-db/5.3.21/ pip install bsddb3
```

(This procedure is subject to change. See the latest pertinent webpages for the most current information.)

This web page explains what is going on here:

http://stackoverflow.com/questions/16003224/installing-bsddb-package-python

The `homebrew` package manager is helpful because it maintains a shared community library of tested installation "formulas" to work with. These resources are specific to MacOS X, however, making `homebrew` inapplicable to Windows or even Linux or other Unix operating systems. If you are running on a non-MacOS X platform, you have to turn to other software package managers; see

http://en.wikipedia.org/wiki/List_of_software_package_management_systems

Some of these managers implement parallels to `homebrew` commands, but you will have to check what parts are actually equivalent.

Appendix D. PyElly System Testing

After any major change to PyElly source code, you should thoroughly validate the resulting system. Check that every module passes muster with the PyElly interpreter and run the current suite of unit and integration tests in the PyElly package.

The following PyElly Python modules have builtin unit tests, executed with the command `python3 M.py`, where `M` is the module name:

```
cognitiveDefiner          cognitiveProcedure        compoundTable
conceptualHierarchy       conceptualWeighting       dateTransform
deinflectedMatching       derivabilityMatrix        dumpEllyGrammar
ellyBits                  ellyBuffer                ellyBufferEN
ellyBufferZH              ellyChar                  ellyCharInputStream
ellyDefinition            ellyDefinitionReader      ellySentenceReader
ellySurvey                ellyWildcard              entityExtractor
extendedContext           extractionProcedure       featureSpecification
generativeDefiner         generativeProcedure       grammarRule
grammarTable              inflectionStemmerEN       macroTable
morphologyAnalyzer        nameRecognition           nameTable parseTree
parseTreeBase             parseTreeBottomUp         parseTreeWithDisplay
patternTable              phondexEN                 prefixTreeLogic
punctuationRecognizer     simpleTransform           stemLogic
stopExceptions            substitutionBuffer        suffixTreeLogic
syntaxSpecification       timeTransform             treeLogic
vocabularyTable
```

Most of these unit tests are self-contained with predefined input test data, but some also will read sys.stdin to get additional input for testing:

```
ellyBase                  ellyBufferEN              ellyBufferZH
ellyCharInputStream       entityExtractor           inflectionStemmerEN
macroTable                morphologyAnalyzer        nameRecognizer
nameTable                 patternTable              phondexEN
stemLogic                 substitutionBuffer        suffixTreeLogic
vocabularyTable
```

Running your own examples with the unit test of one of these modules can help you track down a specific problem in a language description. Enter as many inputs as you want; just type a `<RETURN>` by itself to terminate the input loop here. Manually entered test examples will be optional for pre-release PyElly unit validation, however.

The following PyElly modules have specific test input files included in the standard distribution from GitHub. Their associations are as follows:

```
cognitiveDefiner:        cognitiveDefinerTest.txt
ellySentenceReader:      sentenceTestData.txt
generativeDefiner:       generativeDefinerTest.txt
generativeProcedure:     testProcedure.*.txt
morphologicalAnalyzer:   suffixTest.txt
```

In unit testing, these files are either specified in a command line argument or read from redirected standard input. See the Python code for each tested PyElly module to find out how to do this. Unit test output can usually be quickly evaluated by inspection.

For unit testing of ellyBase, ellySentenceReader, and vocabularyTable.py with Chinese input text, you must explicitly specify the command line flag -zh in the invocation of the module after its name. Otherwise, Chinese characters will be ignored.

For integration testing, the `doTest` shell script with argument `A` will run `ellyMain.py` with preselected parameters for application `A` while reading input from `A.main.txt`. In full integration testing, run `doTest` with each of the PyElly applications having language definition files in the `applcn` subdirectory of the PyElly download package:

```
./doTest echo
./doTest test
./doTest stem
./doTest fail
./doTest zhong
./doTest indexing
./doTest texting
./doTest doctor
./doTest chinese
./doTest querying
./doTest marking
./doTest name
./doTest disambig
./doTest chemic
./doTest minimarking
./doTest ngram
```

The `minimarking` example application is included in integration testing, but it and its test examples are still evolving.

The `doTest` script will automatically compare PyElly output for `A.main.txt` input with the corresponding `A.main.key` file with the builtin `diff` shell command. For example, `doTest querying` might produce

```
test application= querying, input= querying.main.txt

real     0m0.404s
user     0m0.345s
sys      0m0.040s

  <  ACTUAL
  >  EXPECTED
                ...
```

This reports the running time of the test along with any significant output differences found by `diff` listed out below. A successful test should produce no differences.

The comparison here is always with the PyElly translation as the first argument to `diff` (**`ACTUAL`**) and with the `.key` file (**`EXPECTED`**) as the second. The `doTest` script will try to ignore any PyElly output that is not part of a translation. If your application produces any extra output, however, such as from PyElly execution monitoring commands in your rule semantics, then it may show up as a difference. Please make the appropriate adjustments in your interpretation of the test results.

The `bad` application will always fail to generate a language definition because its rules are deliberately malformed. Nevertheless, you should run it with `doTest` as part of integration testing to verify that PyElly can recover from and report on errors in rules. There should be no uncaught exceptions arising from rule input. Error messages from table building also should be make sense.

Integration testing will be the main way to verify that a new version of PyElly or application rules can still handle what it used to do correctly. This is important to do frequently, even with only small changes in PyElly code or in application rules. This will catch problems early, so that translation issues will not accumulate and compound, making debugging much more difficult.

If you do change PyElly code or either the language definitions for a test application or its integration test input, you should also update the `*.main.key` files to reflect any expected changes in output that will show up in PyElly translations.

With `doTest`, you can also process input from a particular file `x.main.txt` while running with the language definition files for an application `A`. Do this with the command line for `doTest`:

```
./doTest A x
```

The output will be checked against the file `x.main.key`, which must be present.

The `marking` integration test in the PyElly package now includes five extra test input files `marking.more*.main.txt`, to run in the above way for extended testing with more sentences. There are corresponding `marking.more*.main.key` files to compare with the translations done by PyElly with the `marking` language definition rules. This splitting of test data into multiple files is mainly historic; but it allows for faster checking when trying to fix an isolated problem found in integration testing.

The main part of the `marking` integration test rewrites 878 sentences from a variety of Web sources; processing them and comparing results with key files take only about 13 minutes altogether. This test data has been augmented with a set of 116 problematic sentences from almost thirteen thousand Google News top stories collected to test the `marking` application on a large text sample. These 116 sentences are in the input file `marking.news.main.txt`; expected results are in `marking.news.main.key`.

The new data presented new difficulties. Web news text typically lack the fairly careful editing of print journalism or even of the earlier 878-sentence test samples. The fast

publication cycles for news tend to produce sprawling sentences and often tortured syntax. This forces a reader to work hard, and a PyElly language definition has to capture major amounts of world knowledge in rules to avoid parsing overflows. The set of 116 especially challenging news text sentences for integration testing will take about 25 minutes to mark up.

The broad suite of integration test applications along with `marking` serves to validate the range of actual processing demonstrated historically in PyElly and its predecessors. The test instances are still limited, however, so that we can never guarantee that any given language definition will be able to mark up all future uncontrolled input. We can only achieve a reasonable level of competence at some point with some set of language definition rules and then continually try to do better.

Although all the other tests are necessary, the `marking` integration test currently is the most important. It has by far the most grammar and vocabulary rules and includes more test sentences than all the other integration tests combined. Its input is actually from the Worldwide Web and are not sentences cooked up with nice linguistic characteristics. Such "wild" data has always been a challenge for any natural language system to process and so provides a serious shakedown of both PyElly code and language definition rules for an application.

Further PyElly work will push testing with even more "wild" data. Fortunately or unfortunately, finding sentence examples to break PyElly processing is still quite easy with any application. Validation therefore always has to be tentative for any kind of language competence; but then that is how human beings themselves naturally learn language. We can hope only to develop our rules through enough examples so as to cover most of what the world can throw at us.

In the `marking`, `minimarking`, `chinese`, and `name` example applications, a major goal is to see how far we can go with PyElly before language definitions become too complicated or too bloated for users to manage. So far, experience with `marking` has been encouraging; despite many unexpected issues and many bugs uncovered, the PyElly approach to natural language has held up. Problems of phrase node overflow with long sentences have been the most serious issue so far.

A new release of PyElly will be uploaded to the Web only after it passes all current unit and integration tests and the `pylint` tool has checked any modified PyElly Python code for common problems in source code. The ellyMain.py module should also be run with the `bad` language rules to verify that this will not crash PyElly. PyElly will receive a new version number only for significant code changes, which will usually mean that previously saved language rule tables have to be regenerated. Any change in how PyElly works will also have to be described in an update of this PyElly User's Manual; that should be done even for releases with no change in version number.

Appendix E. PyElly as a Framework for Learning NLP

PyElly is about doing computational linguistics with rules. It calls for some serious digging into the structure of text, which many data scientists nowadays would rather avoid. With many kinds of text processing, minimizing prior language expertise can be a good strategy for system building; but real-world language in general will be hard, and will usually defeat simple solutions. We need to be both smart and knowledgeable to do NLP more effectively. PyElly brings many resources together to let students experiment freely and to give practitioners proven capabilities right out of the box.

Lately, machine learning and artificial intelligence have been hot areas in computing applications. Notable successes have come out of image recognition and playing of games like chess, go, and poker; and deep neural nets have become the go-to technology for automatic machine translation of natural language. Neural nets in particular are able to learn without supervision when given sufficient training data, and so one must ask whether the rule-based techniques of 20th Century computational linguistics still matter. The hope here is simply to turn AI loose on big data, sit back, and clip coupons.

If you look at some actual machine translation on the Web like Google Translate, however, it is easy to find atrocious results even with deep learning. This is especially bad when the languages come from quite different family groups like Chinese and English. In theory, one might think that a neural net able to learn how to translate well from Spanish to English would be immediately applicable to learning to go from Chinese to English. In reality, translation can be problematic even with closely related languages.

Much has already been written about the complexities and idiosyncrasies of human language. Its structure is messy, usually with many different dialects well on their way to evolving into different languages. Any data scientist should at least be aware of such problems, but to be really serious about processing text data effectively, one needs deeper understanding possible only with extensive practical experience. PyElly will let students get it by tackling and digging into real-world text.

The PyElly toolkit currently includes a non-deterministic finite-state automaton, macro substitutions with wildcard matching, bottom-up parsing driven by a grammar, ambiguity handling through semantics, support for large external dictionaries with multiword terms, basic text entity extraction for dates and times, template matching to recognize common phrases, punctuation handling and sentence demarcation, and special logic for recognizing the names of people. You can combine these resources in various ways to build nontrivial text processing systems like the example applications in the PyElly distribution package.

The PyElly tools should all be familiar to computer science students and remain useful despite the new technologies like automatic programming or artificial intelligence with deep learning. Not every data scientist has to be an expert on low-level text processing, but someone on a major project should be conversant about its ins and outs, if only to temper any magical thinking about breezing through it if given enough training data. Students and practitioners both need to learn some humility here.

With PyElly, a student will see how computational linguistics actually works, whether this is in analyzing the structure of sentences or just extracting stemmed keywords. All the rules for PyElly language processing will be fully visible, never buried in the entrails of some deep neural network. Appropriate rules for a new application will take time to compile and develop, and analyses with them may be hard to follow, but such difficulty is inherent to all natural languages and is the challenge in processing them.

PyElly encourages students to play with language and makes it easy to build toy applications that nevertheless carry out real NLP. Such simple exercises may be of little interest for academic research or for commercial exploitation, but they are excellent for learning and opens the way to more ambitious projects down the road. It could conceivably even be fun, even for students uninterested in eventually becoming a professional computational linguist.

A good way to employ PyElly in secondary schools is to let students take on individual or group projects focused on a single narrow NLP task like those described in Section 14, not necessarily covering all the supported features of PyElly. You probably want to avoid something as complex as the PyElly `marking` example application, but something like `chinese` or `texting` is quite doable. Avoid projects involving extensive extra-linguistic background research, since that alone could easily eat up all available time; let students focus instead on natural language and NLP.

A reasonable PyElly project should take from two to six weeks and may involve defining about a hundred rules altogether. Experimentation is important; no one should be satisfied with initial success in processing a given set of text data input. Instead, students should try to expand the coverage of their applications as much as possible and to see what happens when small changes are made to language definition rules. It is all too easy to get rules tangled up, and future practitioners need to know the hazards.

Unsuccessful projects can be as valuable for learning as successful ones. Natural language processing of unrestricted text is hard and can fail. The varieties of grammatical structure and the exponential possibilities for alternate interpretations of a sentence with ambiguities can be overwhelming. This problem will not go away with approaches like unsupervised deep learning. Students will have to confront their text data and get a clear sense of its nature just like a tailor needs to be an expert on cloth.

As always, it is best to start with something simple and gradually build up rules step by step to into something more elaborate. This can take quite a while. Students should expect to make many mistakes and to spend plenty of time in debugging, reworking, and clarifying their language descriptions. They should nevertheless try to process as much real-world text as they can. That experience will undoubtedly be where most important NLP learning will happen with PyElly or any other system.

Appendix F. A Shallow XML Markup Application

The `marking` example application started out as a limited demonstration of rewriting English text with with shallow XML markup. The plan was to define just enough PyElly language definition rules to analyze a small set of randomly chosen raw text from the Worldwide Web. This initial data consisted of 22 continuous text segments in English, containing 2,766 words in 129 sentences. It was expected to be a major challenge for PyElly, but still suitable for a short-term project.

When putting together demonstrations, one typically selects the input data carefully and makes sure that everything is clean. Any NLP application targeting the Web, however, will meet unvarnished stuff that people actually write and post. Text that breaks every rule taught in grammar school is everywhere. Venturing out on the Web is like driving a vehicle off a test track onto actual streets, highways, and dirt roads, where you can break an axle, crack a windshield, or blow up a radiator.

PyElly is basically a framework to encapsulate linguistic competence in a software package compact enough to run on a home computer. The idea is to be good at the linguistics and to rely less on world knowledge, which can quickly overwhelm an NLP project. The result will fall short of full AI understanding, but such limited processing can still be useful. The question is just how far PyElly could go here in handling actual Web text. How hard could that be?

Shallow markup of Web text would mainly identify simple noun phrases like THE SIX RED APPLES and simple verb phrases like HAS NOT YET BEEN FOUND plus assign traditional parts of speech like noun, verb, adjective, and so forth to the various elements in a sentence. This would also label low-level text entities like numbers, dates, and times, but would ignore more complex recursive grammatical structures like relative clauses, participial phrase modifiers, or direct and indirect quotations.

Here is a shallow XML markup for a simple made-up sentence:

Several dedicated men promised they would reunite on September 11th, 2021.

```
<sent>
<nclu><quan>several</quan><adj>dedicate -ed</adj><noun>men</noun></nclu>
<vclu><verb>promise -ed</verb></vclu>
<pro>they</pro>
<vclu><aux>would</aux><verb>reunite</verb></vclu>
<nclu><date><prep>on</prep>09/11/2021</date></nclu>
<punc>.</punc></sent>
```

Each basic noun phrase is tagged as `nclu` and each basic verb phrase is tagged as `vclu`. The word SEVERAL is a `quan` (quantifier) in a noun phrase and THEY is a `pro` (pronoun). The preposition ON is tagged as a `prep` attached to a `date` element, which is recognized by PyElly as an entity and interpreted syntactically as a noun. The word WOULD is an `aux` (auxiliary) in a basic verb phrase. The final period (.) is marked `punc`, but the comma (,) in the date was dropped in its rewriting. Finally all the elements are grouped between `<sent>` and `</sent>` tags marking a full sentence.

This is far from understanding what the sentence means. Reliably tagging each word with its part of speech and then grouping those words into simple phrases is a good start, however. In particular, it could provide nontrivial information to support and guide deeper sentence analysis; and as a kind of text preprocessing, it can greatly reduce the ambiguity of words that show up in a sentence.

There is of course no single way to mark up a given sentence, but our particular scheme here should be adequate for a proof of principle. It is consistent with how most of us understand the grammar of English and how it fits in with what PyElly language rewriting can readily produce. As it turns out, the main difficulty of markup will not be in the form of output, but in figuring out highly unpredictable input. The magnitude of the task soon becomes evident when we try to rewrite actual "wild" text.

Text from the Web often lacks the editorial control of published books and periodicals. Sentences often run on and on, capitalization and punctuation are irregular, typos are common, and obscure jargon is rampant. The writing is often a word salad tossed together with little regard for readability either by human or by machine. And even in edited text, common words like ADVANCE, BEST, DIRECT, INITIAL, LEAN, OKAY, READY, SMART, WARM, and ZERO can each assume multiple parts of speech, exploding the possible combinations of meanings for a reader or a parser to sort out.

Low-quality data makes shallow markup surprisingly hard, even with only a finite number of target sentences to process. A few general language rules actually will handle many real-world sentences, but wider coverage will often require many rules down to the level of specific words or groups of words. For example, DOUBLE DOWN ON is a verb. This is not cheating; it is just part of common knowledge about how language is used. Such background is required for mastery of any new language; and it should be expected that any automaton will have to figure this out.

The `marking` example application started with only a few grammatical rules just for the syntactic type NCLU (noun cluster) and a few others for VCLU (verb cluster). An NCLU was assumed to have the general form QUAN + DET + NUM + ADJ + NOUN; for example, ALL THE SIX RED BALLS. A VCLU would have the general form AUX + NEG + HAVE + VERB; for example, WILL NOT HAVE KNOWN. Some of these components may of course be missing in an actual text NCLU and VCLU.

Defining grammatical rules is fairly easy if we postpone thinking about cognitive or generative semantics. In PyElly, they will be in the form NCLU->xN NCLU or the form VCLU->xV VCLU; for example, NCLU->DET NCLU. We can use syntactic features with the NCLU and VCLU syntactic types to control the order in which the rules will be applied. Complications can arise, but can be handled by adjusting rules. A showstopper could also rear up at some point, but we still want to see how far we can get.

Overall sentence tagging for shallow markup will also have simple rules. They will take the form SS->ELEM SS, where SENT->SS and where ELEM->NCLU, ELEM->VCLU, and ELEM->x, where x might be a conjunction, interjection, or a miscellaneous syntactic type as well as punctuation. It should be easy to see how this will all work with

the Dick-and-Jane text for teaching young children to read; the challenge is to scale up our language rules to handle Web text from the X-rated wild.

PyElly will also have to know about explicit vocabulary in many cases. In a minimal language definition, we might try to define only functional words like THE, OF, AND, and NOT along with inflectional word endings like -S, -ED, and -ING and common morphological word endings like -LY, -MENT, and -ION. The idea is to allow most words in a sentence to be unknown (UNKN). We would then define some grammar rules like VERB->UNKN -ED, ADJ->UNKN -ICAL, and ADV->ADJ -LY. For example, an unknown word XXXXICALLY could be recognized as an adverb in this way.

With our initial target sample of 129 sentences, however, a minimum explicit vocabulary approach quickly fails. Some cases of markup are easy, but unless PyElly is more specifically restrained, it can do some strange tagging. Our rules really have to be much more detailed; and we have to work carefully to avoid cringe-worthy results. A bit of luck also helps; but some "wild" text data of course could defy any markup. In extreme cases, no PyElly rule may be adequate or a new rule to address a markup problem might derail markups for previous sentences.

Fortunately, PyElly passed its initial test, even though our "wild" data kept uncovering processing problems in almost every new segment of text. These included bugs in PyElly Python code, various errors in the definition of rules, unexpected Unicode characters, new kinds of text entities not covered by existing rules, idiomatic expressions requiring special-case handling, and tricky cognitive semantic scoring. With persistent reworking of both PyElly modules and `marking` rules, however, PyElly did eventually manage to mark up all our initial 129 target sentences with a single set of rules.

That result was a victory of sorts, but required so much work that it quashed hope for a general markup capability. Yet, the number of rules really must be finite for a natural language, or else no one would ever be able to learn it. So, we then might add more rules and hope for some kind of eventual convergence where fewer new ones will be needed to process new sentences. In a practical markup application, we probably also have to be satisfied with reasonable markups of only some large fraction of new sentences.

With no solution yet for markup, we needed more test data to process, and so another 13 text segments were collected from the Web, containing 1,773 words in 91 sentences. In the best of worlds, PyElly would sail through the new data, but the bad news was that former pattern of failures repeated with each new text segment. We again and again had to adjust both PyElly modules and marking rules before we could process all 220 sentences of our expanded test corpus—not a win, but not a total loss either.

PyElly lived to fight another day, however, and its code was getting better through all the testing. So we got four more text samples from various other Web sources to keep on testing. These included 7 segments with 1,289 words in 62 sentences, 10 segments with 1,149 words in 61 sentences, 13 segments with 2,752 words in 133 sentences, and 37 segments with 8,033 words in 403 sentences. We still saw the same pattern of failure

with each new segment in that data, but got better at extending rules to get acceptable markups for all sampled sentences in the end.

We still cannot declare victory yet; but the PyElly rule-based approach has held up so far. It usually takes only a day or two of adjusting to take our marking rules to a new level where they can handle new sentences without failing on old ones. This kind of steady improvement is encouraging. If we can sustain such progress through more and more "wild" Web text, then we can envision achieving a respectable level of competence in markup, but this might take years.

The overall problem in PyElly for shallow markup is now fairly clear, boiling down to combinatorics. When we add more grammar and vocabulary rules, they multiply possible analyses. Unless we can somehow hold down consequent ambiguity in a sentence analysis, the number of possible interpretations of the sentence will grow exponentially. PyElly parsing will then get glacially slow, and the odds against choosing a good markup through cognitive semantic scoring will become quite unfavorable.

The easiest way to reduce ambiguity in PyElly analysis is with a big external vocabulary table with many multi-word terms. Just having fewer unknown terms can help PyElly, but multi-word terms are even better. For example, defining DEPARTMENT OF JUSTICE will combine three separate tokens into one much less ambiguous term. This reduces the degrees of freedom in parsing a sentence; and finding only four or five such long tokens in a text segment can speed up PyElly parsing by an order of magnitude.

To identify possible single- and multi-word vocabulary for a PyElly markup application, we can turn to any large digital dictionary of English. WordNet in particular works out well here because of its comprehensiveness, its organization into ASCII text files, and its liberal open-source licensing. The file `default.v.elly` distributed with PyElly is mostly WordNet 3.0 without some numerical terms already handled elsewhere in PyElly and without some terms incorporating punctuation.

The `marking` example application uses only a subset of `default.v.elly` because the full file takes a long time to load, and it includes many rare senses of common words unnecessarily increasing ambiguity. The current `marking` vocabulary table starts out with recognizable single words and some names in our target text, which may or may not be in WordNet. WordNet is quite comprehensive, but it omits irregular forms of nouns and verbs, most inflected variants, most names, specialized jargon, many common idioms, and of course new terms coined after WordNet 3.0. These non-WordNet additions have been kept separate in the `marking.v.elly` definition file.

For a multi-word vocabulary, we can look for WordNet terms in target text with their first one or two words already in the `marking` vocabulary table. After a little manual pruning of rare or obscure terms that would would be unhelpful, the `marking` vocabulary table grew to almost 40,000 entries, with about half being compounds of two or more words. In comparison, the PyElly `marking` application has just over 700 grammar syntax rules, about 140 pattern rules, and about 70 macro substitution rules.

Here are some actual markups for sentences taken at random from the original target text samples. The first is from Robert Louis Stevenson's "Treasure Island":

He had taken me aside one day and promised me a silver fourpenny on the first of every month if I would only keep my 'weather-eye open for a seafaring man with one leg' and let him know the moment he appeared.

```
<sent>
<pro>he</pro>
<vclu><aux>had</aux><verb>taken</verb></vclu>
<pro>me</pro>
<adv>aside</adv>
<nclu><num>1</num><noun>day</noun></nclu>
<conj>and</conj>
<vclu><verb>promise -ed</verb></vclu>
<pro>me</pro>
<nclu><det>a</det><adj>silver</adj><noun>fourpenny</noun></nclu>
<nclu><prep>on</prep><noun>the 1st</noun></nclu>
<nclu><prep>of</prep><quan>every</quan><noun>month</noun></nclu>
<conj>if</conj>
<pro>i</pro>
<vclu><aux>would</aux><adv>only</adv><verb>keep</verb></vclu>
<nclu><dem>my</dem>
<punc>‘</punc>
<noun>weather-eye</noun><adj>open</adj>
<prep>for</prep><det>a</det><noun>seafaring man</noun>
<prep>with</prep><num>1</num><noun>leg</noun>
<punc>’</punc></nclu>
<conj>and</conj>
<vclu><verb>let</verb></vclu>
<pro>him</pro>
<vclu><verb>know</verb></vclu>
<nclu><det>the</det><noun>moment</noun></nclu>
<pro>he</pro>
<vclu><verb>appear -ed</verb></vclu>
<punc>.</punc></sent>
```

Our second example comes from a website for the Mennonite Church:

Mennonites value the sense of family and community that comes with a shared vision of following Jesus Christ, accountability to one another and the ability to agree and disagree in love.

```
<sent>
<nclu><noun>mennonite -s</noun></nclu>
<vclu><verb>value</verb></vclu>
<nclu><det>the</det><noun>sense</noun></nclu>
<nclu><prep>of</prep><noun>family</noun></nclu>
<conj>and</conj>
<nclu><noun>community</noun></nclu>
<>that</>
<vclu><verb>come -s</verb></vclu>
<nclu><prep>with</prep><det>a</det><adj>share -ed</adj><noun>vision</noun></nclu>
<vclu><prep>of</prep><verb>follow -ing</verb></vclu>
<nclu><noun>jesus christ</noun></nclu>
<punc>,</punc>
<nclu><noun>accountability</noun></nclu>
<pro><prep>to</prep>one another</pro>
<conj>and</conj>
<nclu><det>the</det><noun>ability</noun></nclu>
<vclu>to<verb>agree</verb></vclu>
<conj>and</conj>
<vclu><verb>disagree</verb></vclu>
```

```
<nclu><prep>in</prep><noun>love</noun></nclu>
<punc>.</punc></sent>
```

A third example comes from a conspiracy-theory blog that loves quotation marks:

There are a lot of people hearing alarm bells going off in their heads with the recent announcement of multiple Walmart Supercenters closing, all at once, in multiple states (some of which are states being prepared for the Jade Helm military exercises), all claiming a "plumbing" issue, with recent news of some type of underground tunnel projects involving Walmart and DHS.

```
<sent>
<vclu><adv>there</adv><verb>are</verb></vclu>
<nclu><quan>a lot of</quan><noun>people</noun></nclu>
<vclu><verb>hear -ing</verb></vclu>
<nclu><noun>alarm bell -s</noun></nclu>
<vclu><verb>go -ing off</verb></vclu>
<nclu><prep>in</prep><dem>their</dem><noun>head -s</noun></nclu>
<nclu><prep>with</prep><det>the</det><adj>recent</adj><noun>announcement</noun></nclu>
<nclu><prep>of</prep><quan>multiple</quan><noun>walmart supercenter -s</noun></nclu>
<vclu><verb>close -ing</verb></vclu>
<punc>,</punc>
<adv>all at once</adv>
<punc>,</punc>
<nclu><prep>in</prep><quan>multiple</quan><noun>state -s</noun></nclu>
<punc>(</punc>
<conj>some of which</conj>
<vclu><verb>are</verb></vclu>
<nclu><noun>state -s</noun></nclu>
<vclu><aux>being</aux><verb>prepare -ed</verb></vclu>
<nclu><prep>for</prep><det>the</det><noun>jade helm military exercise -s</noun></nclu>
<punc>)</punc>
<punc>,</punc>
<nclu><quan>all</quan></nclu>
<vclu><verb>claim -ing</verb></vclu>
<nclu><det>a</det>
<punc>"</punc>
<noun>plumbing</noun>
<punc>"</punc></nclu>
<vclu><verb>issue</verb></vclu>
<punc>,</punc>
<nclu><prep>with</prep><adj>recent</adj><noun>news</noun></nclu>
<nclu><prep>of</prep><quan>some</quan><noun>type</noun></nclu>
<nclu><prep>of</prep><adj>underground</adj><noun>tunnel project -s</noun></nclu>
<vclu><verb>involve -ing</verb></vclu>
<nclu><noun>walmart</noun></nclu>
<conj>and</conj>
<nclu><noun>dhs</noun></nclu>
<punc>.</punc></sent>
```

A fourth and final example comes from a Wikipedia article about a highway in Missouri:

The route reaches a roundabout where it intersects Route VV and Prathersville Road, the latter providing access to the southbound direction of US 63.

```
<sent>
<nclu><det>the</det><noun>route</noun></nclu>
<vclu><verb>reach -s</verb></vclu>
<nclu><det>a</det><noun>roundabout</noun></nclu>
<conj>where</conj>
<pro>it</pro>
<vclu><verb>intersect -s</verb></vclu>
<nclu><noun>route vv</noun></nclu>
```

```
<conj>and</conj>
<nclu><noun>prathersville road</noun></nclu>
<punc>,</punc>
<nclu><det>the</det><noun>latter</noun></nclu>
<vclu><verb>providing</verb></vclu>
<nclu><noun>access</noun></nclu>
<nclu><prep>to</prep><det>the</det><adj>southbound</adj><noun>direction</noun></nclu>
<nclu><prep>of</prep><noun>us 63</noun></nclu>
<punc>.</punc></sent>
```

As can be seen, actual text from the Web is much more complex and irregular than the sentence cooked up at the start of this appendix to illustrate shallow markup. Nevertheless, we can successfully mark up such sentences with fairly reasonable rules. Doing 878 real sentences like the above four examples required many rule adjustments, some nontrivial, but nothing outrageous.

Here are some general take-aways for anyone also planning to tackle English Web text:

- Function words like A, AND, OF, and THE are the bones of English and not the meat of any text. These are omitted from WordNet, which lists only nouns, verbs, adjectives, and adverbs. In PyElly, function words are usually defined in the internal dictionary of a grammar. They together tend to be quite frequent in text; but most are individually infrequent and easily overlooked in a language definition. Unidentified function words will almost always be handled wrong by PyElly because we have no reliable way to tell whether they are a preposition, conjunction, or something else. Listing them all out with parts of speech is a chore, but makes life easier. Word sequences like AS WELL AS or AS A RESULT OF should also be taken as a compound function word in a proper markup.
- Common content words like ACCESS, FACE, NEED, and WAGE can be more than one part of speech and may be hard to tag correctly all the time. PyElly cognitive semantic rules are supposed to take care of this problem, but sometimes the logic can get so complex that it may not always produce a desired result. A proliferation of interpretations also makes parsing more likely to hit a phrase node overflow. The best solution here is to recognize such words in compounds as much as possible, like MINIMUM WAGE, WAGE WAR, or WAGE SLAVE, which will reduce ambiguity in analyses.
- Punctuation can often be difficult, especially enclosing punctuation like parentheses and quotation marks. For example, in the noun phrase A **(RED)** KNIT CAP, **(RED)** is meant as an adjective; but we need to introduce some special rules here so that the PyElly parser can figure this out. Otherwise, KNIT CAP will be a separate noun phrase and A **(RED)** will just be some unknown construction. The extra rules are not hard to define, but without them, markup will fail. Quotation marks are especially troublesome in Web text because many writers use them willy-nilly.
- Our original concepts of NCLU and VCLU phrases as described above turned out to be too limited; For example, the simple NCLU defined above does not cover common kinds of English expressions like ACT 2, Sagittarius A*, US 63, or VERSION 10.1.2B. These and many other similar arbitrary forms require their own grammar and vocabulary rules.
- Certain function words like THAT, FOR, IN, TO, FIRST, CAN, LIKE, MAY, and ONE are extra difficult to tag because they have patterns of usage unlike any other words in English. For example, THAT can be a demonstrative, a conjunction starting a relative clause, or an optional marker of an indirect quotation, TO can also identify the English infinitive form of a verb, FOR can be a conjunction in some contexts, IN can also be a noun, adjective or adverb, and MAY can be part of a name, a verb auxiliary, or part of a date. In the `marking` example application, we find only a handful of such difficult function words in sample text, but they can wreak havoc with parsing and must be handled with care through a coordinated combination of grammar, vocabulary, and macro substitution rules.
- Many idiomatic expressions need to be recognized explicitly in a proper markup. For example: FROM BEGINNING TO END, ROCKET SCIENCE, PURCHASING POWER, FOR EXAMPLE, BETTER OFF, and BEAT AROUND THE BUSH are respectively an adverb, a noun, another noun, an adjective or adverb, an adjective, and a verb. WordNets lists many idiomatic expressions, but not all. PURCHASING POWER had to be added as a noun in the PyElly vocabulary table for `marking`. This was done despite the fact that the alternative interpretation of buying electricity is not implausible.

- Names like JOHN JONES need special treatment because they generally do not follow the same rules as ordinary words; for example, we do not want to see JONES split into JONE -S or BAKER into BAKE -ER. PyElly can extract personal names from text as text, but to keep everything simple, the `marking` rules will just list names and name components as straight vocabulary. When new names occur in text for markup, they often also need to be identified as vocabulary. Profligate name dropping is a major cause of parsing overflow especially when many of the name components may be unknown to PyElly.
- Unicode is often a problem. Hyphens are often replaced inappropriately by the Unicode en dash. Other troublesome Unicode characters are ellipsis, non-breaking space, narrow space, trademark ™, Greek letters, letters with diacritical marks, and numerical exponents as in CM². These all have to be recognized and handled properly within PyElly language definition rules. See Appendix G.
- Much of text on the Web is poorly written or poorly edited. It will often be a challenge for humans to read in the absence of context and other cues, and so we cannot really expect that an automated system could process it any better. Lazy writing full of clichés and buzzwords can be a big problem in Web text for sports, business, technology, and politics. They make sentences longer and more likely to result in parsing overflows when they have to be analyzed word by word.

These various issues with Web text data are not peculiar to PyElly, but working within the PyElly framework is a good way to uncover them. A black box approach to natural language with machine-learning framework would be challenged by the same issues, but the problems there may be harder to diagnose. Leaving the dirty work to a neural net can speed up system building, but more or less assumes that the messiness of actual text data can be safely swept under a rug. Such confidence is always premature.

With PyElly, we are obliged to take time to explore the structure of natural language. If nothing else, experience with the `marking` example application shows us that no one should expect smooth sailing in real-world text processing. There is still much to learn about English even after analyzing many thousands of Web sentences. An incremental strategy of getting more rules for text seems to be workable, however, and offers a reasonable way toward validating (or invalidating) the rule-based approach to NLP.

Testing in the `marking` example application has now continued beyond the original six Web text samples. Our newest data comes from taking initial paragraphs from the top 12 stories listed on Google News for a series of days. This usually amounts to about 90 to 100 sentences per day, or about 12 or 13 thousand characters of text. The writing tends to be in a tossed-off short-deadline style, where sentences tend to be longer and quirkier than those in the six Web samples previously seen by the `marking` application.

With the Google News data, we were less concerned with quality of markup than with expanding our vocabulary and finding major processing problems and remedying them. In every day's sample so far, PyElly has run into many parsing overflows and other failures; but continued editing of rules has been adequate for successful parsing and a reasonable markup. Most of the adjustments have been in vocabulary rules; but sometimes new grammar, pattern, and macro substitution rules were needed.

Overcoming parsing overflows in PyElly sentence analysis has been particularly troublesome. In the previous six mixed Web data samples, this occurred in only three or four sentences, but in the Google News data, four or five sentences in each one-day sample typically would overflow in parsing on a first pass through PyElly. After a few rounds of rule adjustment, however, a problem sentence may take only about ten seconds or less to process.

Here is an actual Google News sentence where overflow had to be remedied by multiple rule additions:

> **Special Counsel Robert Mueller's sprawling investigation has spawned a new guessing game in Washington centered on retired Lt. Gen. Michael Flynn, President Trump's former national security adviser.**

This comes from an article in The Hill, a Washington political publication with good editing. The sentence is only about average length for Google News, but with the `marking` rules in the distribution package for PyElly release v1.4.17, PyElly parsing will overflow on it. At this point, we can then successively add new multi-word vocabulary to reduce the number of degrees of freedom in the analysis. Here is what happens with the total time for parsing the sentence after each new vocabulary entry:

	Vocabulary Term Added	PyElly Parsing Time
1	national security adviser	∞
2	guessing game	∞
3	Robert Mueller	2m57.9s
4	special counsel	30.4s
5	Lt. Gen.	9.3s
6	President Trump	9.1s
7	Michael Flynn	2.9s

PyElly continued to overflow even after the addition of the first two multi-word terms. With the third term, parsing did finish, but took about 3 minutes, which is slow. With the 4th and 5th terms, parsing time drops dramatically, but adding the 6th term makes almost no difference. Adding the 7th term once again speeds up parsing significantly. The parsing times are for a MacBook Pro running a 2.7 GHz Intel Core i5 under MacOS 11.13.* (High Sierra) with Python 2.7.9.

The impact of combinatorics is obvious here, and reducing the number of degrees of freedom in a PyElly analysis can greatly speed up parsing. The multi-word vocabulary here is definitely a kind of outside world knowledge, helping to reduce ambiguity and make parsing easier. Except for GUESSING GAME, none of the added multi-word terms are in WordNet 3.0, but most people would agree that we have good reason to put them into our vocabulary table beyond just enabling the processing of one particular sentence.

For example, ROBERT MUELLER is name that will not be found in a standard printed dictionary of English. Yet recognizing it as a unit, either by knowledge of the contemporary political scene or perhaps just by capitalization, seems critical for overcoming the combinatorics of an analysis. We see this happening over and over with other problematic Google News sentences.

Here is some of the extra-linguistic knowledge added to the vocabulary table so that the `marking` example application could handle particular Google News problem sentences.

```
announced the discovery of : VERB
another front in the culture wars : NOUN
armed with fresh ammunition : ADJ
complex new modeling : NOUN
counts of murder : NOUN[:plur]
disappeared from the fever swamps : VERB[^ed]
effectively acknowledging : VERB[^ing]
making a false statement to the FBI : VERB[^ing]
repeatedly used the flag : VERB[^ed]
```

Although these rules are not unreasonable additions, they go far beyond describing simple syntactic relationships between broad categories of words. Such extensive attention to individual special cases may be surprising, but this is only due recognition of the complexity of natural language. In a rule-based system going up against actual Web text, one cannot succeed with only simple rules.

So, how many special case rules will we eventually need to define? Given that the statistics of natural language seem to follow Zipf's Law, we can reasonably expect that a small fraction of vocabulary special cases will account for most of their occurrences in actual text. This means that we have a good chance of achieving a respectable competence in markup just by processing more and more text to discover what rules we .are missing. Can keep this up for long with a tool like PyElly?

With more web text, we are bound to run into pathological sentences that will break any finite set of PyElly language definition rules. We consequently have to tolerate some failures, but if they keep showing up with about the same frequency in every new batch of data, then we still fall short of any solution. So far, after processing more than 130 daily batches of Google News text with about a hundred sentences each, we have only verified that natural language processing is indeed hard.

The good news is that PyElly could work with all the additional rules required to get through all the Google News sentences without a parsing failure or an overflow. As a shortcut, however, we did not make a detailed evaluation of the quality of the markup for every sentence. That would take too much time. Our focus instead was on the analyses produced by PyElly for about a hundred especially challenging sentences. These became a major part of overall integration testing for PyElly.

The general procedure for Google News text was to adjust `marking` rules to handle the latest batch and to check that all the preceding batches could still be processed. If so, then we would move on to the next batch and repeat. Initial processing of each batch typically will produce markup failures in about ten sentences, either because of a parse tree overflow or because of the inadequacy of rules to support a complete analysis.

The following table shows some statistics for the processing of daily batches in twenty-six groups arbitrarily labeled a through z: the collection date of the last batch in a group,

the total number of sentences in a group, the number of sentences with rule failure in the first batch of a group, the number of sentences with parse tree overflow in the first batch, processing time after adjustment of rules to handle the group, and reprocessing time after final rules adjustment of rules for all twenty-six groups.

Batch Group	Last Batch Date	Sentence Count	Initial Batch Rule Failure	Initial Batch Tree Overflow	Processing Time 1st Try	Processing Time Nth Try
a	11/15/2017	385			23m 10s	20m 31s
b	11/20/2017	510			27m 0s	21m 52s
c	11/20/2017	358			23m 28s	14m 44s
d	12/1/2017	397	5	7	32m 19s	14m 18s
e	12/7/2017	389	5	0	29m 53s	22m 57s
f	12/13/2017	428	4	4	35m 15s	18m 21s
g	2/27/2018	443	3	5	40m 10s	21m 27s
h	3/6/2018	527	5	4	28m 45s	16m 42s
i	3/19/2018	467	11	6	32m 39s	23m 44s
j	3/25/2018	447	4	5	19m 48s	16m 1s
k	3/31/2018	471	8	7	38m 24s	26m 57s
l	4/7/2018	482	6	4	35m 28s	17m 33s
m	4/14/2018	521	4	6	45m 30s	32m 32s
n	5/3/2018	545	5	4	38m 31s	30m 31s
o	5/8/2018	551	5	6	44m 31s	34m 25s
p	5/13/2018	536	2	8	27m 25s	22m 4s
q	5/18/2018	549	7	7	40m 28s	29m 7s
r	5/23/2018	527	6	5	50m 21s	44m 47s
s	5/28/2018	547	5	5	41m 4s	34m 51s
t	6/3/2018	546	3	6	33m 2s	30m 18s
u	6./8/2018	517	3	4	29m 38s	25m 39s
v	6/15/2018	553	5	2	35m 8s	27m 31s
w	6/28/2018	556	6	4	36m 22s	23m 24s
x	7/4/2018	557	4	0	35m 56s	26m 3s
y	7/9/2018	542	5	2	39m 6s	30m 17s
z	7/14/2018	537	7	1	43m 24s	37m 50s

Each daily batch was processed only after all previous batches could be parsed with acceptable markup. Since each new batch itself required more rules for successful processing, the `marking` language definition kept growing larger and more comprehensive. This could have resulted in a steady reduction in the parsing failures and overflows for the first batch of each of our 26 groups, but we instead got little consistent improvement over time.

New batches were being processed faster on an initial run, however. This would be due to the percentage of unknown terms dropping with each batch, since the total vocabulary for a given language has to be finite. Still, even with over 32 thousand WordNet 3.0 definitions in the final markup vocabulary, more new WordNet terms continue to turn up with each new batch. These terms will tend to be rarer in web text, but any one of them can still trip up PyElly processing.

The batches vary in sentence count because each text sample was collected as a whole number of paragraphs. This makes performance numbers in the table harder to compare, but the important thing is that nothing catastrophic happened. Our language rules could have gotten so complicated that changing them to get past the next markup failure might become too hard. Even after 12,888 Google News sentences in our 26 sample groups, however, we have had no big problem yet.

Just another batch of text may of course derail PyElly processing. That is to be expected: no given set of language definition rules will ever be able to handle everything posted on the Web. On the whole, however, an incremental approach to building a language definition seems to be holding up. It is, after all, how human children learn language. If we can keep advancing along this avenue for about a decade, PyElly should only keep getting more competent at shallow markup.

(Moving from Python 2.7 to Python 3.8 made PyElly run about 15% faster on an integration test benchmark with the 116 most problematic sentences of the Google News data set. PyElly 1.6.1 took 20m18s; PyElly 2.0 took 16m45s.)

Appendix G. Unicode Issues

You might think that processing natural language text in Unicode should be easy when you can already handle ASCII text. After all, any decent programming language like Python or Java will have libraries that work with Unicode in exactly the same way as with ASCII. There are some complications, however, that we have to get around.

The first issue is what subset of Unicode we want to recognize. It is impractical to try to work with 136,755 distinct glyphs in 139 different alphabets plus all the special symbols, including emoji. Even if our processing strategy had been to map everything into plain ASCII, we still have to decide what actually to convert and what just to ignore. This can make a big difference in the kind of toolkit we end up with.

PyElly opts to recognize characters from the first four pages of Unicode from 0000 through 01DF plus a few characters more; that includes the ASCII character set. This range is mostly unneeded for English but it has been a useful exercise to gain some experience with more general text processing. It also lets us deal with the spelling of European names and with occasional foreign phrases showing up in mainly English text.

PyElly v1.0 worked with only the first two pages of Unicode, but that coverage has steadily grown after working with more and more "wild" text from the Web. A serious general natural language toolkit really needs to address diacritical marks, extended punctuation like " and ", Greek letters, special symbols like © and º, some exponents like cm², currency symbols like € and £, different kinds of Unicode spaces, Chinese pinyin tone markers like ā and ǎ, and even musical notation like B ♭ .

Most of PyElly handling of Unicode is done by the ellyChar module, which currently runs to 330 lines of Python code. This is hardly trivial, but it will have to be expanded many times further to deal with Russian, Chinese, Arabic, or any of the Indic languages at any reasonable level of competence. It is only one of the reasons why natural language processing is hard even with deep neural nets.

An unexpected issue arose with UTF-8, which is a standard byte-encoding of Unicode. Pure Unicode is written out to files in 32- or 64-bit codes. Current practice on the Web is to use UTF-8 instead, which has the advantage of allowing ASCII-only text to look the same as it did before. This requires, however, libraries for programming languages be able to convert UTF-8 into 32-bit Unicode for internal processing in main memory and reconvert Unicode for output to external devices.

Unicode also allows for some tricks in processing that may surprise someone unfamiliar with all its various corners. In particular, the Unicode Consortium has defined three "private usage areas" (PUA) that are permanently unassigned. That is, we should not expect any conforming Unicode text to use any of these codes. PyElly takes advantage of the PUA starting at E000 to encode the character and string wildcards used in its language definition rules. That allows a pattern of literal characters and wildcards to be saved conveniently as a straight Unicode string in a binary file.

PyElly release v1.4.24 also experiments with using Unicode to pass information between various stages of processing. This involves the ASCII hyphen-minus =002D, which is currently treated as non-embedding in default token formation and which also will fail to match the `$` wildcard for the end of a token. This can sometimes be problematic when trying to get around certain common text irregularities with hyphens.

As of v1.4.24, we can use a macro substitution to change a hyphen-minus =002D in certain contexts into a Unicode hyphen =2010. The Unicode hyphen in particular will match the PyElly `$` wildcard. This may seem like a strange kind of natural language processing, but it can solve some non-trivial problems in PyElly rule definition. We have no such option when working with ASCII-only text data.

As of v2.0, PyElly has been rewritten in Python 3.8, which simplifies Unicode handling. The Python 2.* definition of strings as ASCII caused many problems. In Python 3.*, strings are Unicode and text files are assumed to be in UTF-8. This eliminates much explicit encoding and decoding done in earlier releases of PyElly.

With PyElly v2.2, the generative semantic command `UNICODE` takes a hexadecimal argument and converts this into a single Unicode character that is ten put into a PyElly output buffer. This allows PyElly output rewriting to produce arbitrary Unicode, including non-alphanumeric and non-printing characters.

The bottom line is we have to get our hands dirty with Unicode text as it exists in the wild before we can understand how to process it properly in PyElly. That is only a narrow detail of computational linguistics, but it is still something unwise to gloss over. Natural language is hard, and we need to apply all available knowledge to give us the most traction possible in developing practical solutions.