PyElly User's Manual

23 December 2013

PyElly is an open-source software tool for creating computer scripts to analyze text in English and other natural languages. These will of course fall far short of realizing the talking robot fantasies of Hollywood, but with only modest effort, you can still build many nontrivial linguistic applications short of full understanding. PyElly will also handle many of the low-level kinds of necessary language processing that can get in the way of solving much more challenging problems.

PyElly should especially be helpful if you just want to master the nitty-gritty of language processing. You can quickly build scripts to do simple tasks like conjugating French verbs, rephrasing information requests into a formal query language, compressing messages for texting, extracting names and other entities from a text stream, and even re-creating the well-known Doctor simulation of Rogerian psychoanalysis.

Such computer natural language processing is hardly new; we did it even when machines were less than a millionth as powerful as they are today. The overall problem remains quite hard even with today's technology, however, requiring development of complex algorithms and codification of extensive linguistic knowledge. PyElly brings together many of these resources in a single convenient free package.

Why do we need yet another natural language processing toolkit? To begin with, a complete natural language solution is still far off, and so we can still benefit from a diversity of tested approaches to the problem. Also, though PyElly is all new code, it is really a legacy system, with its core components having a history going back almost 40 years. This sounds quite ancient, but language changes slowly, and mature software can have benefits.

The impetus for PyElly came from observing that different natural language problems tend to involve the same mundane subproblems. For example, information retrieval and machine learning with text data can both be aided greatly with words reduced to their roots. Instead of variants like RELATION, RELATIONAL, RELATIVELY, and RELATING, we deal with just RELATE. This is of course the familiar stemming subproblem, but available resources here have often been disappointing.

A stemmer of course is easy to build, but it takes time and commitment to do a good job here, and one hardly wants to have to start from scratch each time. This is true of other basic language processing capabilities as well. So, we really want at least some kind of reusable library at hand, but everything will be much more usable if we integrate them more closely.

The current implementation of PyElly was 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 into class projects that have to be completed fairly quickly. PyElly can be of broader interest, though, because of its broad range of natural language support: stemming, tokenizing, entity recognition, sentence extraction, idiomatic transformation, syntactic parsing, and ambiguity handling.

Actual use of PyElly will of course still require some linguistic expertise. You have to define the details of language processing that you want, but much of this has been prebuilt in PyElly for English input. The standard PyElly distribution includes the definition scripts for five different examples of applications that you can modify as a headstart in making your own applications.

The current PyElly code consists of a series of Python modules in over fifty-five separate files. These modules should run on any computer platform with a Python 2.7 interpreter, including Windows 7 and 8, Linux, Mac OS X and other flavors of Unix, IOS for iPhone and iPad, and Android. The PyElly source is downloadable from the Web under a standard BSD license; you may freely modify and extend it as needed. Though intended mainly for educational purposes, there are no restrictions on commercial usage.

For recognizing just a few dozen sentences, a tool like PyElly is probably overkill; you could handle them directly by writing custom code in any standard programming language. More often, however, there will be too many possible input sentences to list out fully, and you will have to characterize them more generally through various rules describing how the words you expect to see are formed, how they combine in text, and how they are to be interpreted.

PyElly is set up as a kind of simple translator; that is, it reads in, analyzes, and writes out text transformed according to supplied rules. So an English sentence like “She goes slowly” might be rewritten as the French sentence “Elle va lentement” or the traditional Chinese 她去慢. Or you might reduce the original sentence to just “slow” by stripping out suffixes and words of low content. Or you may want to to rephrase the sentence as a question like “Does she go slowly?” All of this can be accomplished entirely within PyElly.

PyElly rules can be of various 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 a desired form for output. Creating such rules requires some trial and error to get right, but usually will be no more difficult than setting up macros in a word processor. PyElly takes care of many details automatically and provides debugging tools when there is a problem.

Many natural language system building tools, especially those in academic research, tend to focus on mechanisms for handling thorny problems in language interpretation. This can lead to impressive specifications and possibly interesting theoretical papers, but can introduce many complications without necessarily leading to practical applications. PyElly tries instead to be simple and pragmatic instead. In response to classically tough sentences like “Time flies like an arrow”, you may get by just by replying "Huh?”

PyElly was specifically built to be compact enough to run on mobile devices, if necessary. 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 actual number of rules actually defined. A major project may involve hundreds of grammar rules and hundreds of vocabulary elements, but useful text analyses may run with just a few dozen rules and no domain vocabulary.

What is a grammar, and what is a vocabulary? A vocabulary establishes the words you want to recognize; a grammar defines how those words can be arranged into sentences. You may also

specify idiomatic rewriting of particular input word sequences prior to analysis as well as define patterns for making sense of various kinds of unknown words. For example, you can recognize 800 telephone numbers or Russian surnames ending in -OV without having to list them all out.

This manual will explain how to do all of this and also cover the basics of language and language processing that every PyElly user should know. As a prerequisite, you should already be able to create and edit text files from the command line of whatever system you choose to work on and set up file directories where PyElly can find them. In an ideal world, some interactive development environment (IDE) could make everything easier here, but that is yet to be.

Currently, PyElly is biased toward English input, although it is set up to handle the Latin-1 subset of Unicode at the character level. This subset includes the familiar ASCII characters as well as all the letters with diacritical marks used in Western European languages. For example, PyElly knows that é is a vowel, that ß is a letter, and that Œ is the uppercase form of œ. This can be helpful even for English data, since one often encounters names with foreign spellings.

The technically savvy may also want to know that PyElly has real builtin analytic expertise: stemmers for English inflectional word endings as well as many prefixes and suffixes, a macro preprocessor for idioms and special cases, part-of-speech recognition through multiple algorithms, syntax-driven parsing of sentences when given a grammar and vocabulary, a special evaluator of ambiguous meanings, and a rewriter to translate parsed sentence input.

As any beginning student of a foreign language soon learns, its rules are often messy. Irregularities always arise to trip up someone mechanically trying to speak or write from a simple grammar. PyElly users still face the same kind of problem here, but by dealing with exceptions as they crop up, we can evolve our rules so that we can eventually reach some useful level of parlance. There is no royal road to natural language understanding, but persistence can achieve wonders over time, and PyElly can help you keep going in this endeavor.

You need not be an experienced linguist or computer programmer to develop PyElly applications; and I have tried to write this manual to be understandable by non-experts. The only requirements for users are basic computer literacy as one might expect of current high school graduates, linguistic knowledge as might be picked up from a first course in a foreign language, and willingness to learn. PyElly users should start out with simple kinds of processing and gradually progress to more complex analysis as experience is gained.

In addition to this introduction, the PyElly User's Manual consists of twelve major sections and plus an appendix as follows:

1 The Syntax of a Language

2 The Semantics of a Language

3 Defining Tables of PyElly Rules

4 Operations for PyElly Generative Semantics

The first two sections cover what you need to know about language in general. The next two go into actual PyElly syntactic and semantic definition facilities. The fifth gives an example of some simple PyElly language rules. The sixth explains how to work with PyElly code and data files. Then come two sections on advanced PyElly facilities for a language grammar and associated vocabulary. The ninth section is about rules for handling sentences and punctuation. The tenth is on the algorithms of PyElly parsing, and the eleventh is on how to compile language rules and make them work correctly. A final section explores possible uses of PyElly and describes five different example applications.

PyElly ("Python Elly") was inspired by the Eliza system created by Joseph Weizenbaum over 50 years ago for studying natural language conversation, but PyElly has a completely different design. Its Python implementation is the latest in a series of related natural language processors 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 PyElly parsing algorithm is based onVaughn Pratt's LINGOL (LISP, 1973). Frederick Thompson's REL system (1972) also influenced the PyElly design.

1. The Syntax of a Language

A language is as a way of putting together words or other symbols to form sentences that other people can make sense of in a particular context. In general, not all combinations of symbols will make a meaningful sentence; for example, "Cat the when" is nonsense in English. The definition of a language for PyElly processing consists of first identifying those combinations that do make sense and then assigning suitable interpretations to them.

If a language is small enough, such as the language of obscene gestures, we can simply list all its possible "sentences" and write down what each of them mean. Most nontrivial languages, though, have so many possible sentences that this approach is impractical. A way around this problem is to note that languages tend to have regular structures; and by identifying such structures, we can formally characterize the language much more concisely than by listing all possible sentences one after another.

The structural description of a language is called a *grammar*. It states what the building blocks of a language are and how they form simple structures, which in turn combine into successively more complex structures. Almost everyone has studied grammar in school, but formal grammars go into much more detail. They are organized as sets of *syntactic rules*, each describing one particular kind of language structure. Such syntactic rules will provide a basis for both generating and recognizing sentences of a language with a computer.

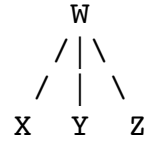
In linguistics, a formal rule of grammar is expressed in terms of how one or more structures come together to produce a new composite structure. These syntactic rules are commonly written with an arrow notation as follows:

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

This 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 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 kinds of sentence diagramming once taught in junior high school and coming back into vogue. In fact, we would draw the following equivalent diagram on a blackboard for the syntactic rule above.



where the `W` can in turn be part of a higher-level structure. Either way of representing a syntactic structure is fine, but the arrow notation will be more compact and easier to type out on a keyboard, especially as syntactic structures grow more complex.

Syntactic rules generally can be much more complicated than `W->X Y Z`, but for PyElly, it turns out that we can get by with rules restricted to just three types:

```
X->w
X->Y
X->Y Z
```

where `X`, `Y`, and `Z` are structure types and `w` is a word or other kind of vocabulary element. To express a single rule like

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

we can instead use the pair of restricted rules

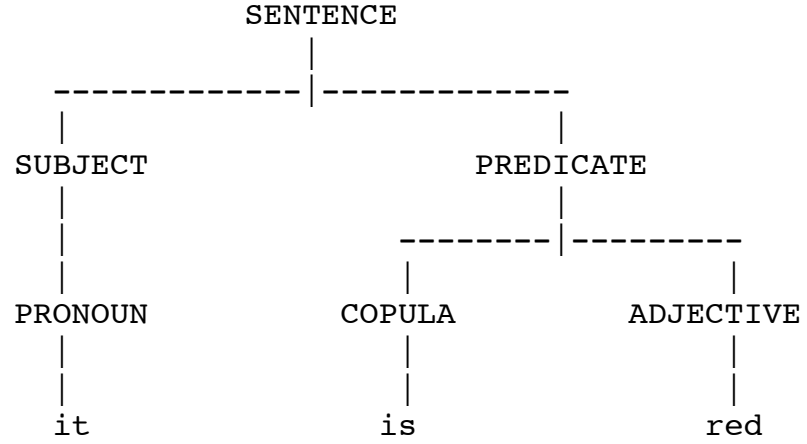
```
Z->W T
T->X Y
```

The following simple grammar might be employed to describe the sentence “It is red.”

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

Explicit words and other vocabulary elements are shown here in lowercase letters, while structure names are uppercase. The order of the rules here is not important.

The structure of a sentence as implied by such rules can be expressed graphically as a labeled tree diagram, where the root type must be `SENTENCE` and where branching corresponds to splitting into constituent substructures. For example, the sentence “It is red” as described by the syntactic rules above 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 a parse tree; for example, the phrase node `PREDICATE` in the tree above encompasses the actual sentence phrase `is red`. A competent natural language system can do all this analysis automatically, given the grammar rules; and the resulting tree diagram then provides a starting point for interpreting the sentence.

Note that the various parts of speech and structure types used in rules do not have to be same ones taught in grammar school. Except for the starting type `SENTENCE`, you can call them anything you want, as long as you are consistent. It does help, however, to use familiar descriptive names to make your rules more readable to yourself and others.

We can now extend the range of sentences described by our example grammar with more rules for new kinds of structures and vocabulary. For example, adding the new rules

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

will make "An apple is red" a sentence in our language. The addition of

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

will also put "It falls" and "An apple falls" into the language we can recognize here. This process may be continued to encompass more types of structure and more vocabulary. You can add new rules in any order you want.

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

Technically speaking, PyElly grammar rules as described here define a "context-free language." With such a grammar, you cannot correlate the possibilities for a given structure with the possibilities for a parallel structure. For example, consider the 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 categories of person and number: "We fall" versus "he falls". When grammatically acceptable `SUBJECT` and `PREDICATE` structures can be formed in more than one way, our simple rule here will not allow us to restrict a `SENTENCE` to have only certain combinations of subjects and predicates for agreement. We could of course write different correlated rules like

```
SENTENCE->SUBJECT1 PREDICATE1

SENTENCE->SUBJECT2 PREDICATE2

SENTENCE->SUBJECT3 PREDICATE3

...
```

where `SUBJECT`*i* always agrees 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 everything easier, not harder.

Although English and other natural languages are not context-free in the theoretical sense, we can still treat them as context-free in a practical sense. There is a tremendous advantage in doing so, because we can then apply many sophisticated techniques already developed for parsing artificial computer languages, which do tend to be context-free. This is in fact the approach implemented in PyElly.

For convenience, PyElly also incorporates semantic checking of the results of parsing and allows for various shortcuts to make grammars more concise (see Section 7). These extensions can be put on top of a context-free parser to give it some context-sensitive capabilities, although some kinds of sentences still cannot be handled by PyElly. (The classic context-sensitive examples are parallel subjects and predicates, such as in the sentence “He and she got cologne and perfume, respectively.”)

The syntax of natural language can get quite complex in general; but we usually can break this down into 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.

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 sticks to the basics that you will need to know.

2. The Semantics of a Language

The notion of meaning is difficult to talk about. This can be complicated even for single sentences in a language, because their meaning involves not only their grammatical structure, but also where the sentence 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 news 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 grasp references to subjects like sex, golf, or the current weather.

Here we shall limit the scope of semantics even further: PyElly will deal with the meaning of sentences only to the extent of being able to translate them into sentences of another language and to evaluate alternate options when we have more than one possible translation. This has the advantage of making semantics less mysterious while allowing us still to achieve 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 "I feel hot" into a computer 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 we happen to have an effective processor for the second language, but not the first. This modest approach to semantics allows PyElly to deal with some quite practical problems of language understanding; and it certainly beats talking endlessly about the philosophical meaning of meaning without ever accomplishing anything.

As noted before, the large number of possible sentences in a natural language prevents us from compiling a table to map each input into its corresponding output. Accordingly, we must work instead with the semantics of the various constituent structures defined by a grammar and combine their individual interpretations to derive the overall meaning of a given sentence. Here, it is convenient to define the semantics of a language structure as *procedures* associated with the grammatical rule for the structure.

PyElly will actually define two different kinds of semantic procedures here: those that do translation will be called "generative," while those that do evaluation will be called "cognitive." At this point, however, we shall focus on generative semantics, and leave cognitive semantics to Section 7. Cognitive and generative procedures will be built up from different basic actions.

In PyElly, the semantic procedure for a constituent structure will be called as a subroutine by the next higher-level structure containing it. After an input sentence has been parsed to get its levels of constituent structures, we can then derive its whole meaning by first calling the semantic procedure associated with the highest-level structure, which by convention will always be that of type `SENTENCE`. This call should then set off cascades of procedure calls through the various lower constituent structures of the sentence to give the final output.

Since the creation of output sentences will mainly be manipulation of characters, PyElly semantic procedures are defined in terms of basic text editing operations commonly found in word processing programs: inserting and deleting, buffer management, searching, and substitution. Consistent with the idea of PyElly semantics being procedures, there will also be local and global variables, pointers, structured programming control structures, subroutine calls, special modeling functions, and consistency checks.

Communication between different semantic procedures will be through local and global variables. Local variables will have a scope as 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 from there. When there are multiple active declarations of a variable with a given name, the most recent one applies. Upon exit from a procedure, all of its local variables become undefined. Global variables, on the other hand, are accessible to all procedures and will remain defined even across the processing of successive sentences.

Let us define some semantic procedures as an illustration. Suppose that we have the syntactic rules

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

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

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

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

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 all the way from `SENTENCE` down to individual words like `we` and `know`. These connections give us a framework to extend our translation capabilities just by adding more syntactic rules plus their semantic procedures such as

```
PRONOUN->they
```

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

You may have noticed, however, our example above is not quite right. More so than English, French verbs must agree in person and number with their subject, and so the translation of `know` with `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 `SUBJECT PRONOUN` becomes `they` (third person plural).

Obviously we need more elaborate semantic procedures here for correct agreement. This is where various other PyElly commands mentioned have to come in; and in particular, we shall use some local variables to pass information between the semantic procedures for our syntactic structures so that they can agree (see Section 5). Nevertheless, our framework of semantic procedures attached to each syntactic rule and called recursively remains the same.

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

Writing syntactic rules and their semantic procedures is in fact a kind of programming and will require programming skills. It will be harder than you first might think when you try to deal with languages like English or French. PyElly, however, is designed to help you to do this programming in a highly structured way, and it certainly beats trying to write the same kind of translation code explicitly in a language like PASCAL or BASIC or even LISP.

3. Defining Tables of PyElly Rules

You should understand by now the concept of syntactic rules and semantic procedures. This section will introduce the mechanics of how you actually define them for PyElly by setting up tables in 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, just provide the appropriate files for PyElly to load.

PyElly tables fall into five main types: (1) grammar, (2) vocabulary, (3) macro substitutions, (4) patterns for determining syntactic types, and (5) morphology. The grammar of a language for an application tends to reflect the capabilities supported by a target system, while a vocabulary tends to be geared toward a particular data base; macros support particular users of a system, and special patterns tend to be specific to certain applications. Separate tables make it easier to tailor PyElly processing in different environments while allowing language definitions to be reused.

This section will focus on the grammar, special pattern, and macro tables, which will be required by most PyElly applications. The creation and use of tables for vocabulary and morphology will be described in Section 8, "Advanced Programming: Vocabulary." Some of the more technical details of semantic procedures for vocabulary definitions will also be postponed to Section 7, "Advanced Programming: Grammar."

The current PyElly package defines each type of table as a different Python class with an initialization procedure that reads in rules from an external source, typically a text file. PyElly follows a convention for the names of the text files associated with a particular application `A`:

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

`A.m.elly` for macro substitutions.

`A.p.elly` for special patterns.

You may replace the `A` here with whatever name you choose, subject to the file-naming rules of the file system 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 substitutions or patterns. Section 6 will explain how PyElly will look for definition files and read them.

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 need to rename your files afterward because of their insistence on writing out files only with extensions like `.txt`.

In its current Python implementation, PyElly can handle up to 48 different structure types (i.e. parts of speech) in its input files. Five of these structure types will be predefined by PyElly with special meanings.

`SENT`	Short for SENTence. 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 UNKNown. This structure type is automatically assigned to strings not known to PyElly through its various lookup options. (See Section 8 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 7 for more details.)
`PUNC`	For punctuation. See Section 9.

You will of course have to make up your own names for other structure types that you need for a PyElly application. Names may be arbitrarily long in their number of characters, but may include only letters, digits, and periods (`.`); upper and lower case will be the same. You do not have to use traditional grammatical names like NOUN, but there is no point in being obscure here.

You may want to keep structure names unique in their first four characters. This is because PyElly may truncate names to that many characters in order to produce properly aligned formatted diagnostic output like parse trees (see Section 10, "The PyElly Parsing Algorithm"). This might be confusing if you have syntactic types like `NOUN` and `NOUNPHRASE`.

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

```
# PyElly Definition File Example
# example.g.elly
g:sent->ss

__
g:ss->unkn

__
g:ss->ss unkn

__
```

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

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

These will be explained in separate subsections below.

3.1 Grammar (`A.g.elly`)

An `A.g.elly` text file may have three different types of definitions: (1) syntactic rules with their associated semantic procedures, (2) a small number of words with their associated semantic procedures, and (3) general semantic subprocedures callable by other semantic procedures. These definitions will be respectively identified in the `A.g.elly` file by special markers at the beginning of a line: `G:`, `D:`, and `P:`. The definitions may be appear in any order.

3.1.1 Syntactic Rules

These must be entered as text in a strict line-by-line format. Syntactic rule definitions will follow the general outline as shown in monospaced font:

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

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

b. A '`#`' at the beginning of a line or a '` # `' elsewhere indicates that a comment follows on the right; PyElly will ignore all comments within definition text.

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

d. The single and double underscores (_ or __) here serve to mark the semantic procedures for a rule. A generative semantic procedure for a syntactic rule will always appear between such underscores.

e. A cognitive semantic procedure will always appear before the single underscore, but this will be described later in Section 7.

f. The actual basic actions for generative semantics will be described in Section 4 ("Operations for PyElly Generative Semantics").

g. If a semantic procedure is omitted, various defaults apply; see Section 5 ("PyElly Programming Examples").

3.1.2 Grammar-Defined Words

In general, the vocabulary for a PyElly application should be separate from a grammar. For scalability, PyElly will store its vocabulary in an external Berkeley DB database; and Section 8 will describe how this should be set up. Some word definitions, however, may also appear alongside the syntactic rules in a grammar definition file.

For example, some words like THE, AND, and NOTWITHSTANDING are associated with a language in general instead of any particular content. These words are probably best defined in a grammar file anyway. In other cases, there may also be so few words in a defined vocabulary that it is convenient just to include them all internally in a grammar rather than externally.

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

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

a. The `D:` is mandatory in order to distinguish a word rule from a grammatical rule of the form `X->Y`.

b. The underscore separators are the same as for syntactic rules. A word definition will also have cognitive and generative semantics.

c. 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.

d. The `w` must be a single word, possibly hyphenated. Multi-word terms in an application must be defined in PyElly's external vocabulary.

3.1.3 Generative Semantic Subprocedures

Every PyElly generative semantic procedure will be code written in a special programming language for text manipulation. This language allows for named subprocedures, which can be defined in a grammar file without being attached to a specific syntax rule or internal vocabulary rule. Such subprocedures may be called by any generative semantic procedure for a PyElly rule or by another subprocedure. The calls may be recursive, but try this only if you know how to avoid an infinite loop here.

Any subprocedure must be properly defined in a `*.g.elly` grammar file before it can be called. Subprocedure definitions may appear anywhere in the grammar file.

A subprocedure will take no arguments and will return no values. All communication between semantic procedures must be through global and local variables and from passing text in the current PyElly output buffers. This process will all be covered in Section 4 of this manual.

A subprocedure definition will have the following form:

```
P:n                     # a marker + procedure name "n"
_                       # 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 string of alphanumeric characters without any spaces. It can be of any non-zero length. The case of letters is ignored.

c. The underscore separators are the same as for syntactic rules and word definitions.

d. The procedure body is always generative semantics. There will be no cognitive semantics for a generative subprocedure.

3.2 Special Patterns (`A.p.elly`)

Many elements of text are identifiable by their form, but are too numerous to list out completely; for example, Social Security numbers, web addresses, and Russian surnames. PyElly allows you to recognize such elements through the specification of the patterns that such text elements will match. That is now also how you will have to define ordinary decimal numbers.

PyElly pattern matching of text constituents relies on a finite-state automaton (FSA). This is a common way to organize algorithms to recognize strings that every aspiring computational linguist should be familiar with. The basic concept to understand here is a *state*, which sums up what we need to remember at each step of trying to make a match with an input string.

An FSA starts out in the state when there is nothing yet to remember. From any current state, one then has a limited number of possibilities to check, and if there is any match to an input string, then one goes to a different new state and moves forward in the input string. This process continues until one reaches some final state, which determines whether one has a final match.

PyElly identifies states by number, with the initial state always being 0. The possibilities for matching at a state are defined as patterns of literal characters and wildcards to match. A `*.p.elly` definition file will consist of separate lines specifying a possible pattern for a given state, a PyElly syntactic structure type associated with a match, and a next state upon a match.

Here is a simple example file of some FSA pattern definitions:

```
# 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 entities of type `NUM` as plain integers like `1024`, simple decimal values like `3.1416`, and longer digit strings with commas like `1,001,053`. A pattern line in general has four parts as follows:

```
state    Pattern    Syntactic Type    next
```

a. The first part is an integer ≥ 0 representing a PyElly automaton state.

b. The second part is a pattern specifying arbitrary sequences of letters, numbers, and certain punctuation: hyphen (`-`), comma (`,`), period (`.`), slash (`/`). If these are present, they must be matched exactly within a word being analyzed.

c. A pattern may also have various wildcards as follows:

`#`	will match a single digit `0` - `9`
`@`	will match a single letter `a` - `z` or `A` - `Z`, possibly with diacritics
`?`	will match a single digit or letter
`*`	will match a n arbitrary sequence of non-blank characters, including a null sequence
`&#`	will match one or more digits in a sequence
`&@`	will match one or more letters in a sequence
`_`	will match a single space character
`^`	will match a single vowel
`%`	will match a single consonant
`$`	will match the end of a word, but not add this to the extent of matching

d. Brackets `[` and `]` in a pattern will enclose an optional subsequence to match; only one level of bracketing is allowed and no wildcards are allowed inside.

e. The third part of a pattern line is a part of speech like NOUN.

f. The fourth part is the next state to go to upon matching a specified pattern. This will be an integer ≥ -1. A -1 state means to stop scanning an input string and return the latest match status.

For example, the pattern `###-##-####` matches Social Security numbers, while the pattern `(###)_###-####` matches a telephone number with an area code. See Section 8 for more on possible number patterns. A special character like `&` may be matched in a pattern by escaping it with a backslash character: `\&`.

All patterns not ending with the `*`, `_`, or `$` wildcards will have a wildcard `$` appended automatically.

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

This facility allows the replacement of specified substrings in input by other substrings before the start of parsing. This provides a convenient way of defining synonyms and abbreviations and of carrying out simple syntactic transformations inconvenient to handle within the framework of augmented context-free grammar rules. The general way to define a macro substitution rules for processing as follows:

```
P Q R->A B C D
```

a. Each macro definition is limited to a single line. Since macros are in their own file, we need no identifier at the beginning of each line.

b. The left and right sides of the substitution may have an arbitrary number of components, each being separated from the others by a blank.

c. Upper and lower case is significant only on the right side.

d. Input words matching the pattern of the left side are replaced by the right side; all possible substitutions are made before advancing input.

e. The left side of a substitution rule may have wildcards; these will be the same as recognized in the special patterns described by the preceding subsection (3.2).

f. A `\\1`, `\\2`, `\\3`, and so forth, on the right stands respectively for the first, second, third, and so forth, sequences of text matched by wildcard patterns on the left. Note that each of these applies to a sequence of contiguous wildcards; for example, the pattern `###abc@@@` on a match will associate the first 3 digits of a match with `\\1` and the last three letters with `\\2`.

g. When any macro is matched, its substitution will be done. Then all macros will be checked again against the result at the same text position. When a macro eliminates its match, though, checks will be ended at that position.

h. The order of macro definitions is significant. Those defined first will be applied first and may affect the applicability of those defined afterward. Macros starting with a wildcard will be checked after all others, however.

Macro substitutions are trickier to work with than grammatical rules because it is possible to define them to work at cross-purposes. You can even get into an infinite loop of substitutions if you are not careful. Nevertheless, macros can greatly simplify a language definition when you use them properly and keep their patterns fairly short. They are always applied before any syntactic analysis of a sentence with grammatical rules and so can override grammatical rules in effect. Macros have no associated semantic procedures.

Use macro substitutions to handle idioms or exceptions to other rules.

4. Operations for PyElly Generative Semantics

PyElly semantic procedures are expressed in a structured programming language with the operations of a character-oriented text editor having multiple buffer work areas. Each statement in the language consists of an operation name followed possibly by arguments separated by blanks. The operations fall into several main categories, described in separate subsections.

4.1 Insertion of Strings

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

```
APPEND string           # copies "string" into buffer

BLANK                   # append a blank character

SPACE                   # same as BLANK

LINEFEED                # start a new line

OBTAIN                  # copies the token text at
                        # the current phrase position
                        # into buffer
```

4.2 Subroutine Linkage

For calling procedures and returning from them:

```
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 when a rule is of the form
                        # X->Y Z.

RETURN                  # returns to caller

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

4.3 Buffer Management

Processing starts with a single output text buffer. Spawning other buffers will often be helpful to keep the output of different semantic procedures separate for additional processing before they are joined together. You can put aside the current buffer and starting processing in a new buffer and then move text back and forth between the two buffers.

```
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,
                         # deallocates the new one
```

These allow a semantic procedure to be executed for its side effects without yet putting anything into the current output buffer. 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 changed to "string2"
```

4.4 Local Variable Operations

Local variables can store a Unicode string. They are declared within the scope of a semantic procedure and will automatically disappear upon a return from the procedure.

```
VARIABLE x=string        # declares variable x with
                         # initial string value; if
                         # no value is specified,
                         # initialization is to null

SET x=string             # assigns string to local
                         # variable x of the most
                         # recent declaration

ASSIGN x z               # assigns the value of local
                         # variable x to the local
                         # variable z in their most
                         # recent declarations
```

The equal sign (=) must appear with SET even if `y` is missing. If a referenced local variable is undefined in either a `SET` or an `ASSIGN` operation, it will become automatically defined. You may write `VAR` as shorthand for `VARIABLE`; they are equivalent.

4.5 Global Variable Operations

Global variables are permanently allocated 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 how many global variables you can have. Global variables `gp0`, `gp1`, … can be set from a command line (see Section 6); others you can define yourself in generative semantic procedures.

4.6 Control Structures

Only two structures are supported: the `IF-ELIF-ELSE` conditional and the `WHILE` loop; the operations are as follows:

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

ELIF x=s                 # 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 failed

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

```
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 or `WHILE` loop. PyElly will reject a generative semantic procedure if an `END` is missing.

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

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

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 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 list here.

Within a `WHILE` loop, you may also have

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

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

4.7 Character Manipulation

These work with the current output buffer.

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

EXTRACT x < n                # drops the first n characters of
                             # the next output buffer and
                             # sets local variable x to the
                             # string of those 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 character from
                        # start of next output buffer
                        # without removing it

PEEK > x                # get a single character from
                        # end of current output buffer
                        # without removing it

DELETE n <              # deletes n characters from
                        # the start of the next output
                        # buffer

DELETE n >              # deletes n characters from
                        # the end of the current output
                        # buffer

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

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

If n is omitted for the EXTRACT operation above, it is assumed to be 1. If the < or > are omitted from a DELETE or a SHIFT, then < is assumed.

The DELETE operation also has the special variant

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

4.8 Selection From Table

This operation that uses the value of a local variable to select a string for insertion at the end of the current buffer. It is equivalent to

```
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#)
```

If the value of local variable `x` is equal to substring `vi`, then substring `si` will be inserted. If there is no match, nothing will be inserted. When `vn` is null, then `sn` will be inserted if the variable `x` matches no other `vi`.

For example, the `PICK` operation

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

is equivalent to

```
if   x=uu
    append aaaa
elif x=vv
    append bbbb
elif x=ww
    append cccc
else
    append dddd
end
```

but this is much less compact.

4.9 Searching

There is one 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 are shifted to the
                          # current buffer up to the first
                          # occurrence of string s

FIND s <                  # as above, but transferring
                          # is in the other direction
```

If no substring `s` is encountered, an entire buffer will be moved. If the < or > is omitted, then > is assumed.

4.10 Tracing

To monitor the execution of semantic procedures, you can use the command:

```
TRACE                    # show parse node information
```

When the semantic procedure for a phrase contains this command, PyElly will print to the standard error stream the starting word position of the phrase in a sentence, its syntax type code, the index number of the syntactic rule involved, and the degree of branching of the rule. When the command is put into a named subprocedure, PyElly will follow the chain of calls back to the first semantic procedure for an actual phrase.

4.11 Capitalization

PyElly is quite limited in what it can do about upper and lower case in output. There is currently a single command:

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

If you fail to do a `SPLIT` and `BACK` operation before running this command, you will get a null pointer exception, which will halt PyElly.

4.12 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 in a semantic procedure by giving the name in parentheses:

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

The subprocedure name must be defined somewhere in a PyElly `A.g.elly` file.

The null procedure call `()` is always defined; it is equivalent to executing a `RETURN`. This is used only for handling special kinds of PyElly vocabulary definitions (see Section 8).

5. PyElly Programming Examples

We are now ready to look at some simple examples of semantic procedures for PyElly syntax rules, employing the operations defined in the preceding section. Sections 7 and 8 will discuss more advanced capabilities.

5.1 Default Semantic Procedures

The notes for the Section 3.1.1 of this manual mentioned that omitting the semantic procedure for a syntax rule would result in a default procedure being associated with it. Now that that we can finally define those default procedures. A rule of the form `G:X->Y Z` has the default

```
_
  LEFT
  RIGHT
__
```

Note that a `RETURN` command is unnecessary in this procedure as it is implicit upon executing its final command. 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 and are like 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 obtained by a PyElly analysis.

In the first example of a default semantic procedure above, a call to the procedure for the left constitutent 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. You might write this as

```
_
  RIGHT
  LEFT
__
```

In the second example above of a default semantic procedure, there is only one constituent in the syntactic rule, and this will always be a left constituent by convention. We have no right constituent, and trying to call its semantic procedure with a `RIGHT` command would result in an error and halt PyElly.

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 will execute a `OBTAIN`. Either a `LEFT` or a `RIGHT` command here would result in an error.

5.2 A Simple Grammar

We now give an example of a nontrivial PyElly grammar. The problem of making subjects and predicates agree in French was discussed previously. Here we make a start at a solution by handling just the present tense of first conjugation verbs in French plus 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
  VAR NUMBER=s
  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 syntax 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:default
_
  PICK PERSON (1=ons#2=ez#3=ent#)
__
P:1cnjg
_
  IF NUMBER=s
    PICK PERSON (1=e#2=es#3=e#)
  ELSE
    (default)
    END
__
```

Semantic subprocedures `default` 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 them in several places below and so define the subprocedures just once for economy.

```
D:sing<-VERB
_
  APPEND chant    # root of verb
  (1cnjg)         # for first conjugation
__
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
      (default)
      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 it 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 small number of such verbs in any natural language.

Here is what 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. For full-scale translation, 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 various tenses other than present. You should, however, be able to envision now what a full PyElly grammar should look like; it would basically be more of what we see above.

6. Running PyElly

We have so far described how to set up definition text files to create the various tables to guide PyElly operation. This section will go into how to run PyElly for actual language analysis, but first you will have to do some preliminary setting up. That should be fairly straightforward, but you may want to get some technical help here.

To begin with PyElly was written entirely in version 2.7 Python, which seems to be the most widely preinstalled by computer operating systems. The latest version of Python is 3.*, but unfortunately, this is incompatible with 2.7. So make sure you have the right version here. Python is free software, and you can download a 2.7 release from the Web, if necessary. The details for doing so will depend on your computing platform.

After getting Python, you should next get Berkeley DB, a free database package used by PyElly to handle its vocabulary rules. Although you can get by without Berkeley DB, this will severely limit what you can do in PyElly. Get the latest release, currently 6.0, available from the Oracle website for most operating systems.

There is an additional complication with Berkeley DB. Python originally built in support for BDB, but this has been dropped in version 2.7. To use Berkeley DB in Python, you must also download and install the free third-party `bsddb3` module. Again, details here will depend on your computing platform. To save grief, go through a software management tool here instead of trying to do all the installations by individual commands from your keyboard.

Once you have Python with `bsddb3` and Berkeley DB installed, you are ready to 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 under copyright.

The Python code in PyElly currently consists of 57 modules comprising about 15,000 source lines altogether. A PyElly user really needs to be familiar with only three of the modules.

1. `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.

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

3. `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 all your `*.*.elly` language definition files to generate the various tables that will guide PyElly analysis of input data. Section 3 introduced three of them. For a given application A, we can have `A.g.elly`, `A.p.elly`, and `A.m.elly`, with only the

`*.g.elly` file mandatory. Subsequent sections of this user manual will describe the other `*.*.elly` definition files.

The PyElly tables created for an application A will be saved in two files: `A.rules.elly.bin` and `A.vocabulary.elly.bin`. These are Python pickled files. They are not really binary in that you can look at them with a text editor, but they will be hard for people to read. If the `*.elly.bin` files exist, ellyBase will compare their creation dates with the modification dates of corresponding `*.*.elly` definition files and create new tables only if one or more definition files have changed. Otherwise, the existing tables will be reloaded from the `*.elly.bin` 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 five examples of simple applications and their definition files to show you how to set everything up (see Section 12).

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

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

This will then generate the PyElly tables for the `name` application and prompt for individual sentences to analyze with them. If no application is specified in the command line, the default is `test`. Running ellyBase can be helpful in tracking down problems in a language definition because it will provide a detailed dump of grammar rules and closer monitoring of PyElly analysis. It also lets you prebuild `*.elly.bin` files so that PyElly can start up faster.

For an actual application, you normally will invoke ellyMain from a command line. The `ellyMain.py` file is a straight Python script that reads in general text and allows you to specify various options for PyElly language 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,...][ -noLang] [name] < text
```

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

The commandline flags are all optional, with the following interpretations:

`-b`	operate in batch mode with no prompting; PyElly will otherwise switch automatically to interactive mode with prompting when its text input comes from a user terminal.
`-d n`	set the maximum depth for show 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 analysis. Set n = 0 to disable parse trees. See Section 10 for more details.
`-g v0,v1,v2,..`	define the PyElly global variables `gp0`, `gp1`, `gp2`, ... for PyElly semantic procedures to have the respective specified string values `v0`, `v1`, `v2`, ...

`-noLang`	do not assume that input text will be in English; the main effect is to turn off English inflectional stemming (See Section 9).

When ellyMain starts up in interactive mode, you will see the following message:

```
PyElly version 0.1 Natural Language Filtering
Copyright 2013 under BSD open-source license by C.P. Mah
All rights reserved

reading <a> definitions
recompiling language 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 figure out where the sentence boundaries lie.

As soon as PyElly reads in a full sentence, it will try to write a translation to its output. In interactive mode, this will be after the first linefeed after the sentence because PyElly has to read a full line before it can proceed. Linefeeds will NOT break out of the ellyMain input processing loop, although two consecutive linefeeds will terminate a sentence even when punctuation is absent. End your input with an EOF (control-D on Unix and Linux, control-Z in Windows). A keyboard interrupt (control-C) will break out of ellyMain with no further processing.

All PyElly text input should be in UTF-8 encoding. Outside of `*.*.elly` files, however, PyElly currently is able to work only with ASCII and Latin-1 letters here; all other input characters here will be treated as spaces. PyElly language definition files may contain arbitrary UTF-8. The `chinese` application in Section 12 uses definition files with both traditional and simplified Chinese characters in UTF-8.

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

Here is a example of interactive PyElly translation with a minimal set of language rules (`echo.*.elly`), which defines a simple echoing application:

```
>> Who gets the gnocchi?

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

where the second line is actual output from ellyMain. Elly by default will convert upper case to lower, and will strip off English inflectional endings. You can more complete echoing by turning off inflectional stemming and morphological analysis.

By default, PyElly will look for the definition files for an application in your current working directory. You change this by editing the value for the symbol `baseSource` in `ellyConfiguration.py`. The five PyElly applications described in Section 12 are distributed in separate subdirectories under the main directory of Python source files, resources, and documentation.

PyElly `*.py` modules by default will be in your working directory, too. You can change where to look for them, but that will involve resetting environment variables for Python itself. PyElly is written as separate Python modules that must be found and linked together 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.

On the whole, you must be comfortable with computing at the at the level of command lines in order to run PyElly either by `ellyMain.py` or by `ellyBase.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.

7. Advanced Capabilities: Grammar

As noted earlier in this manual, PyElly is built around a parser for context-free languages to take advantage of extensive technology developed for computer programming languages. So far, we have stayed pretty much in the strict context-free domain, although there were some loopholes in macro substitution prior to parsing and in local variables shared by generative semantic procedures executed after parsing.

You can actually accomplish a great deal with with such facilities alone, but for more challenging language analysis, PyElly supports an array of other capabilities beyond the restrictions of context-free languages. 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 normally a topic of grammar, but PyElly breaks this out as a separate level of processing to simplify the design of the basic system. The details here will be in Section 9.

7.1 Features

PyElly currently allows for only 48 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 `grammarTable.py`, but we have a better option here.

For more flexibility in defining language rules, PyElly also supports qualification of syntactic types by syntactic and semantic features. This has the effect of multiplying the total number of syntactic types.

7.1.1 Syntactic Features

A syntactic feature is a binary tag that augments the meaning of various classes of syntactic types. Currently, PyElly allows up to sixteen syntactic features for each class of syntactic types. You can define the classes and name the features however you want. If needed, you can get more than sixteen syntactic features by redefining the variable `FMAX` in `grammarTable.py`.

The advantage of syntactic features is that you can specify grammar rules that ignore them; and so a single rule can handle what would have to be multiple parallel rules when you add new syntactic types close to an old syntactic type. This lets us make a grammar more concise.

For example, a `DEFINITE` syntactic feature allows a definite noun phrase to be specified in a syntax rule without having to introduce a new type like `DNP`. This new type would not really differ much syntactically from the generic noun phrase type `NP`, but would have to have its own complete set of grammar rules in a PyElly language definition.

With PyElly, the grammatical attribute of definiteness can be attached as feature to a syntactic type. In particular, we would have something like `NP[:DEFINITE]` instead of `DNP`, where

DEFINITE can be present or absent for an instance of an NP. Note that a grammar rule like PRED->VP NP will apply to NP[:DEFINITE] as well as plain NP. This not so for DNP.

PyElly syntactic features are expressed by an optional bracketed qualifier appended to a syntax 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 features for a specific class of syntactic types and F1, …, Fn are feature names composed of alphanumeric characters, possibly preceded by a prefix "-". It is logical to define different sets of feature names for categories of syntactic types like nouns and verbs, but the choice is up to you.

A set of syntactic features must immediately follow a syntactic type name with no spaces in between. PyElly allows spaces in the listing of individual syntactic features for easier reading, but a space before the left bracket ([) will always raise an error.

A syntax rule referring to 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 certain 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 for the DEFINITE feature.

For clarity, identifiers should be a punctuation character other than brackets or '+' or '-'. You should have at most five or six sets of syntactic features, although PyElly sets no upper limit here. Be conservative here; remember that syntactic features are supposed to simply grammar rules, not make them more complicated.

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.

7.1.2 Semantic Features

A sentence is ambiguous if it breaks down more than one way into constituent structures of a language description. This is undesirable, but usually unavoidable for any nontrivial subset of a natural language like English. PyElly therefore provides a way to evaluate the alternative decompositions and choose the most semantically plausible one. If a choice must later be rejected for any reason, the analysis will automatically take the next best choice.

PyElly ambiguity handling includes the definition of sets of up to sixteen semantic features associated with classes of syntactic structures. These deal with what words refer to, not how they fit together in a sentence. For example, the two phrases “a big idea” and “a little boy” are syntactically identical, but they will be semantically plausible only in certain contexts; one will seldom say “a big idea played Pokemon” or “a little boy was invalidated”.

A concise way to state such restrictions is to have an `ABSTRACT` semantic feature with the word “idea” and a `ANIMATE` semantic feature with the word “boy” and to have rules for “play” that favor an `ANIMATE` subject and for “invalidate” favor an abstract object. These conditions are important only when there is a choice of interpretations; if there is only one possible interpretation, PyElly will accept it regardless of its plausibility.

In PyElly, semantic features are specified in a way similar to syntactic features. That is, we have a bracketed list of feature names along with a set identifier. These should be different from the identifiers for syntactic types to avoid confusion.

Semantic features will appear in as a series of clauses immediate following the statement of a syntax rule that will constitute the cognitive semantics for a `G:` or `D:` grammar rule alluded to earlier. Each clause will have the basic form

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

The symbol “x” may be either `*L` or `*R`, and the “#” is sequence of **+**’s or **–**’s. More +’s means more plausibility; more **–**’s mean less. You can also specify this as **+***n* or **–***n*, with the obvious meaning when *n* is an integer.

A clause here is divided into two major parts by `>>`. The left part specifies conditions on the immediate subconstituents of a sentence structure in order for the clause to apply. The right part specifies actions to take if the left side is satisfied. In evaluating a given rule of grammar, its clauses will be tried in sequence until one is found to apply. It is possible that none will apply.

The prefixes `L` and `R` on the left part of a clause specify the constituent substructures to be tested, respectively left and right. The “x” prefix on the right is for specifying inheritance of features, with `*L` meaning the left subconstituent and `*R` meaning the right; that is, `*R` means to copy the semantic features of the right subconstituent into the current phrase. The feature set in the right part of the clause will indicate specific features to be set there.

Here is an example of a syntax rule with both cognitive and generative semantics. The cognitive part has three clauses for semantic plausibility:

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

_
  RIGHT
  LEFT

__
```

The cognitive clauses follow immediately after the statement of the syntax rule and are terminated by a line beginning with a single underscore. Parts of a clause may be omitted, with the following defaults applying: no inheritance, no setting or clearing of features, and neutral plausibility. Omission of conditions on the left is equivalent to an always-true condition; if both `L` and `R` tests are missing, then the `>>` may be omitted also.

The total semantic plausibility of a structure is computed as the total number of `+` ratings minus the total number of `-` ratings in its various levels of substructures. With several competing structures, the one with the highest such score will be taken as the most semantically plausible. If no cognitive semantic clauses are specified in a grammar, then every structure will have a neutral plausibility of zero as a default. With tie scores, a structure will be selected "randomly", with a different choice each time.

Note that a syntax rule without an explicit generative semantic procedure can still have cognitive semantic plausibility clauses. In this case, you can omit the single underscore to write something like

```
G:NP->ADJ UNKNOWN
  >>--
__
```

The double underscore must still of course terminate every syntax rule.

7.2 The `...` Syntactic Type

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

```
...->
```

This is called a null rule, and it is a special case in PyElly and cannot be explicitly specified in a `*.g.elly` file for any syntactic type, including `...`. In strict context-free grammars, empty phrases are always forbidden. They are allowed only in so-called type 0 grammars, the most unrestricted of all; and the languages described by such grammars tend to be avoided 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. Instead of having to write both the rules

```
z->... y
z->y
```

we can equivalently write just the one rule

```
z->... y
```

because of the implicit null rule in PyElly allowing `...` to match an empty phrase.

The `...` type is specifically to support keyword parsing, which focuses on certain words in a sentence and more or less ignores the others. A PyElly grammar for such a parser can be written without `...`, but would then require more rules. The Doctor script for PyElly illustrates how this could be done; its grammar might include syntax rules like the following:

```
ss->x[@ 0] ...
x[@*right]->... key
...->unkn ...
```

The syntactic type `key` here represents all the various kinds of key phrases to recognize; for example, “mother” and “dream”. In our rules, the different kinds of phrases will each represent the same syntactic category `key`, but having different combinations of 15 syntactic features to work with, we can encode up to 2^{15} different kinds of key phrases.

The actual responses of our script will be written as 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. Some simplified 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.

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

```
g:ss->...
—
```

A more complete Doctor script is provided as one of the test applications in the PyElly distribution. Here, we are just giving some incomplete examples of how to use the `...` syntactic type in a PyElly definition file.

The `...` syntactic type also has the restriction that you cannot specify syntactic features for it. If you put something like `...[.F1,F2,F3]` in a PyElly rule, it be treated as just `...`. This is to avoid problems that would arise because of how the PyElly parser works.

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 and is even trickier for PyElly to handle as an extension to its basic framework of context-free parsing. The various restrictions here make sentence analysis easier for PyElly while not really preventing you from doing what you need to do.

8. Advanced Capabilities: Vocabulary

So far, all of examples have been where all the words in a language are explicitly listed out within a PyElly grammar and assigned their own semantic procedures. This approach is fine when our vocabulary is small, but in most natural language applications, we need to deal with from hundreds to tens of thousands of distinct terms. Putting them all into a `*.g.elly` file is unwieldy and probably entails more keyboard entry than most people care to do.

We need to be more efficient in its language definition process. In this section, we shall look at how PyElly can help you in various ways to work with much larger vocabularies. The main problem here is how to assign a syntactic type and syntactic features to a term extracted from text read in by PyElly and to assign it appropriate semantic procedures. In a context where the latter is fairly simple translation, we want this to be as compact as possible.

PyElly provides us a grab bag of tricks here that may come in handy in various contexts. We have already seen some of these tricks in previously in this manual, but this section will describe them in more detail. There will be three main topics: applying word analysis to the PyElly builtin `UNKN` type, recognition of entities like numbers, time, and dates, and the use of PyElly vocabulary tables, which are maintained as external databases.

A vocabulary table is the final fallback for defining terms and phrases to be recognized by PyElly. In the real world, however, we cannot always rely having everything predefined in just one place. For example, a system often has to deal with misspellings, unusual alternate spellings, slang, jargon, new coinage, arbitrary alphanumeric identifiers, and names made up mainly to facilitate web searches. So PyElly is set up to allow you to be as flexible as possible here.

The bulk of PyElly vocabularies will typically be in vocabulary tables. These allow for defining multi-terms terms, which cannot be done with `D:` grammar rules. Separate vocabulary tables also make it easier to reuse a grammar with different vocabularies. The only disadvantage is that term definition options will be limited in vocabulary tables; this is to facilitate semiautomated generation and to save space.

8.1 Working With the `UNKN` Syntactic Type

When PyElly cannot identify a word through its available means, that word is assigned a predefined PyElly structure type `UNKN`. In effect, PyElly generates a temporary rule of the form:

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

This rule is in effect only for the current sentence being processed, and like all `D:` rules, it can only be for a single word extracted from input text. You can then supply grammar rules that describe how to interpret such unknown words with respect to known ones. 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, but will favor a noun interpretation. PyElly will then try to analyze a sentence with both interpretations, and if both succeed, it will take the more plausible one.

You can also deliberately define a word as unknown

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

if this is ever necessary, or you can force a term recognized by PyElly to be treated as unknown to allow you to reduce the number of grammar rules you have to write:

```
G:UNKN->DATE
__
```

Do this only if you are avoiding grammar rules for the `DATE` syntactic type. Otherwise PyElly `DATE` rules can end up giving you a ambiguous interpretation of a sentence and so behave unpredictably.

8.1.1 Inflectional Stemming

Given an `UNKN` word in English text, PyElly will automatically remove `-S`, `-ED`, and `-ING` endings to see if this produces a known word. If this is still unknown, then you can provide grammar rules that infer syntactic information from the endings that had been removed. For example,

```
D:-ED <- ED
__
G:VERB[|ED]->UNKN ED
  >>+
__
```

You can completely disable this English inflectional stemming by setting the `language` variable in the `ellyConfiguration.py` file to something other than EN. To override such stemming for a particular word, just define it in a PyElly grammar or a vocabulary so that it is no longer unknown.

In the example above, we are fairly sure of having a verb because of the `-ED` inflection; but there is always the problem of names like “Michele Shocked”. English inflectional stemming is the

oldest and most highly evolved of all PyElly components; but there are always exceptions, and you will be responsible to taking care of what your applications are likely encounter.

PyElly inflectional stemming operates at the level of English spelling rules and recognizes many common special cases; it is generally quite accurate for common English words. For example,

```
winnings ==> win -ing -s

placed   ==> place -ed

judging  ==> judge -ing

cities   ==> city -s

bring    ==> bring
```

If words with such inflections will appear in input sentences and inflectional stemming is turned on, then the rules of grammar for an application will have to take this into account. To begin with, you will have to define each inflectional ending with syntactic information just as we do with grammatical function words.

```
D:-s <- S
__
D:-ed <- ED
__
D:-ing <- ING
__
```

Note the hyphens (`-`) on the endings being defined; PyElly stemming puts them in automatically. The syntactic types for endings are arbitrary; call them whatever you want.

If inflectional stemming produces a known word, then your grammar rules will also have to explain the endings; for example,

```
G:VERB->VERB S
__
G:VERB->VERB ED
__
```

and so forth. Current inflectional stemming is for English only. If inflections are not explicitly handled in a grammar, sentences with such endings may not parse, since inflectional stemming will always be turned on by default.

PyElly inflectional stemming rules are written in their own special language of nested conditional statements. These currently are loaded from the text files `Stbl.sl`, `EDtbl.sl`, `INGtbl.sl`, `rest-tbl.sl`, `spec-tbl.sl`, and `undb-tbl.sl` when PyElly starts up. The operation of these rules is rather complex and will not be covered in this manual, although

someone familiar with writing code for string analysis can figure out what is going on locally in a particular rule file.

Here is a segment of code from `Stbl.sl`, which tells PyElly what to do for a possible `-S` ending when it is preceded by an `IE`.

```
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.

All PyElly inflectional stemming is at such levels of detail, with many special cases built in (e.g. SORTIES, COOKIES, MOVIES). At some point, you may want to edit some of this logic to make PyElly inflectional stemming work better for you or even to write a new inflectional stemmer yourself, perhaps for an inflected language other than English.

8.1.2 Morphology

Words in natural language often are built up from other words by the addition of prefixes or suffixes. For example, in English, the word "unrealizable" might be broken up into the components UN-, REAL, -IZE, and -ABLE. Here the -IZE suffix changes the adjective 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 kind of analysis falls into the area of linguistics called morphology. It can get quite complicated because of historical derivation and cross-language borrowing; but for practical natural language processing in PyElly, we can just proceed in a straightforward way here. Analyzing an unknown word into its morphological components will help in assigning it a syntactic type.

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`. These are akin to the grammar, macro substitution, and word pattern tables already described. There are 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 words that otherwise would be assigned 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 have to add appropriate grammar rules to recognize prefixes and suffixes and to incorporate them into an overall analysis of an input sentence.

8.1.2.1 Word Endings

PyElly suffix analysis will be done after any removal of inflectional endings. 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 and actions upon matching. Its format 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; the actions associated with an ending 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: 1= none, 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 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 from English inflectional stemming.
- A period (`.`) indicating 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.
- A number indicating how many of the starting characters of the unkept part of a matching ending is to drop to get a morphological suffix to be reported.
- A string specifying what letters to add to the front of the unkept part of a matching ending in order to make a complete morphological suffix.

In applying such patterns in the analysis of a word, PyElly will always take the action for the longest match. For example, if something at 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 case of the LENGER rule above, PyElly will accept a match for it at the end of word only if it is preceded by a consonant in the word. If so, the rest of the action specified by the rule is to add 5 characters of the matched ending to the resulting word with a suffix removed. With the remaining part of a matched ending, PyElly will then drop no characters, but add an E in front to get the actual suffix removed.

So the word CHALLENGER will be analyzed as follows:

`CHA`**`L`** `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)

The period (`.`) in the action for LENGER specifies no further morphological analysis. If there had been a comma (`,`), then PyElly could produce a sequence of different suffixes by reapplying its rules to the word resulting from a previous analysis. This can continue indefinitely, with the only restriction being that PyElly will block the removal of any ending if it fails to leave at least two characters in the remaining part of a word. This is to prevent a short word like `DENT` from being analyzed as `D ENT`.

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

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

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

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

A grammar would of course have to be ready to deal with a possible succession of morphological suffixes.

8.1.2.2 Word Beginnings

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.
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. As with suffixes, the longest matching prefix will be taken, and 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 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.

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.

A word of course may be decomposed into suffixes, prefixes, and inflections. For example, NONFUNCTIONING becomes NON+ +FUNCT -ION -ING with the tables from above. A grammar would then have to stitch these parts back together.

To begin with, you will need a rule like

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

and you should by now be able to fill the other required grammar rules yourself.

8.2 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 as Flight VX 84, 888-CAR-TALK, 2.718281828, NASDAQ APPL, or orotidine 5'-phosphate.

The main problem with entities is that we are likely to have almost none of them in a predefined vocabulary. People seem to handle them in stride while reading text, however, even we are unsure what a given entity means exactly. This is in fact the purpose of much text: to talk about something a reader 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. Yet in a tool like PyElly, some basic entity extraction support can be quite valuable.

8.2.1 Numbers

PyElly has no predefined `NUM` syntactic type. If you need to recognize numbers in text input, then you must define them yourself through special patterns in file `*.p.elly` as described in Section 3.2. Otherwise, numbers appearing in text be treated as type `UNKN`. The PyElly predecessor written in C did have compiled code for number recognition, but this covered only a few possible formats.

The current PyElly scheme lets us readily change the patterns to which numbers have to conform. Section 3.2 provides an example of finite-state automaton logic able to deal with integers and numbers with fractional parts in decimal notation. You can readily modify this logic as needed for a particular natural language application.

PyElly does have some builtin capabilities for normalizing numbers in text, however, so that you need fewer patterns to recognize them.

- Automatic stripping out of commas in numbers:

  `1,000,000` ==> `1000000`.

- An automatic mapping of spelled out numbers to a numerical form:

  `one hundred forty-third` ==> `143rd`

Note that you still will need patterns to recognize the rewritten numbers here. You can turn off all such rewriting by setting the variable `ellyConfiguration.rewriteNumbers` to `False`.

8.2.2 Dates and Times

Dates and Times could be handled as patterns, but their forms can vary so much that this would be a major chore with a finite-state automaton. For example, here are two dates:

```
the Fourth of July, 1776

2001/9/11
```

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

To make these methods available, they have to be listed in the `ellyConfiguration.py` module. Here is the 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 it from the `extractors` list. The second element in each list entry is a string syntax specification, which generally includes a syntactic type plus syntactic features to assign to a successfully extracted entity. This has to be coordinated with a grammar.

The `date` and `time` methods above are built into PyElly. These will do normalizations 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, `09:22:17EST`. 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. You will often want to convert these dates and times to other forms. The `XX` epoch indicator in a date and the `ZZZ` zone indicator in a time may be omitted.

8.2.3 Other Entities

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

1. The name can be anything legal in Python for method names.

2. The method should be defined at the level of a module, outside of any Python class.

3. The method takes a single argument, a list of individual Unicode characters taken from the current text being analyzed. PyElly will provide that list. The method may alter the list, but you will have be careful in how you do this if you want the changes to persist after returning from the method. You will definitely need some Python expertise here.

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 an entity is not at the start of an input list, return 0.

5. 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.

6. 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 3.2.

7. Install a new method by editing the `extractors` list in the PyElly file `ellyConfiguration.py`.

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.

8.3 PyElly Vocabulary Tables

PyElly can keep large vocabulary tables in external files maintained with the free BSD Database package, which is separate from Python. If you want to use vocabulary tables, you will have to download and install BSD Database on your computing platform along with the third-party bsddb3 Python module that provides an interface between Python applications and BSD Database. Doing this is fairly straightforward, but depends on the target platform and so will not be described here. You may need some expert technical support.

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

An entry in a PyElly vocabulary table consists of a single line in one of the following formats:

```
TERM : SYNTAX

TERM : SYNTAX =TRANSLATION

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

TERM : SYNTAX (procedure)

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)
```

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

```
Lady Gaga

Lili St. Cyr

Larry O'Doule

"The Robe"

ribulosebisphosphatecarboxylaseoxygenase
```

The '`:`' is required to let PyElly know when a term ends; no wildcards are allowed in a `TERM`. `SYNTAX` is just the usual PyElly syntactic type plus optional syntactic features.

SEMANTIC-FEATURES are the bracketed semantic features for cognitive semantics; PLAUSIBILITY is an integer value for scoring a phrase formed from a vocabulary entry. Both SEMANTIC-FEATURES and PLAUSIBILITY may be omitted, but if either is present, then the other must also be so.

The final part of a vocabulary entry is optional and can take one of three forms as shown above. A TRANSLATION is a literal string to be used in rewriting a vocabulary entry; the '=' is mandatory here. The x=Tx y=Ty z=Tz form is actually a generalization of TRANSLATION; it maps to the generative semantic operation

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

It is possible here that 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 matches none of the other specified options.

A (procedure) in parentheses is a call to a generative semantic subprocedure defined 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 [^celeb] =Stefani Joanne Angelina Germanotta
Lili St. Cyr : noun[:name] [^celeb] 0
horse : noun FR=cheval ES=caballo CN=馬 RU=лошадь
twerk : verb[|intrans] [^sexy] (xxxx)
```

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

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

A term may have multiple vocabulary table entries; for example,

```
bank : noun [^institution]
bank : noun [^geology]
bank : verb[|intrans]
```

If the word BANK shows up in input text, then all of these entries will be incorporated in a possibly ambiguous PyElly analysis.

Often, a vocabulary table may have overlapping entries like

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

PyElly will always take the longest entry and ignore any shorter matches.

For a given application `A`, the PyElly `ellyMain` module will look for a vocabulary table definition in the text file `A.v.elly`. If this is missing, the file `default.v.elly` is taken; this is listing of most of the nouns, verbs, adjectives, and adverbs in WordNet 3.0, which totals to more that 155,000 entries in all. Define an empty `A.v.elly` file if you do not want this huge vocabulary, which will take a long time to read in and compile.

PyElly uses BSD Database to create a compiled vocabulary data base for an application `A`, which will be stored in the file `A.vocabulary.elly.bin`. If you change `A.v.elly`, PyElly will automatically recompile `A.vocabulary.elly.bin` at startup. Recompilation will also happen if `A.rules.elly.bin` 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 tables, work with the `*.*.elly` files.

9. Sentences and Punctuation

Formal grammars typically describe the structure of single sentences. PyElly accordingly is set up to analyze one sentence at a time through the `ellyBase` module. In typical text, however, sentences are all jumbled together, and we somehow have to divide them up properly before we can do 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 stops the sentence. It is not hard to recognize such exceptions, but this is yet more detail to take care of on top on an already complex language analysis.

PyElly divides text into sentences with its `ellySentenceReader` module, which employs a sequential punctuation-checking algorithm to detect sentence boundaries in text. While doing this, PyElly also normalizes each sentence so as to make subsequent processing easier. The simplicity of the algorithm will make it tend to get too many sentences, but we can help it out by providing some supporting modules with more smarts.

Currently, the PyElly `stopException` module works with a user-provided 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 quite workable.

The approach of PyElly here is to provide sentence recognition only a notch or two better than what one can cobble together just using Python regular expressions. If you really need more than this, then there are other resources available (e.g. Python NLTK). The `ellySentenceReader` module should be quite 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 readable line by line with the Python readline() method. The `ellyCharInputStream` module is a filter that removes extra white space, substitutes for Unicode characters not in the Latin-1 subset, and replaces single new line characters with spaces. The `ellyCharInputStream` and `ellySentenceReader` modules both operate at the character level and serve together to convert input text into individual sentences for PyElly processing.

The `ellySentenceReader` module currently recognizes five kinds of sentence punctuation: period (`.`), exclamation point (`!`), question mark (`?`), colon (`:`), and semicolon (`;`). By default, these will be taken as a sentence stop whenever followed by a whitespace character. A blank line consisting of two new line characters together will also terminate a sentence without any punctuation. The procedure here is deliberately minimal.

9.1 Extending Sentence 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 sentence 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.

9.1.1 Stop Punctuation Exception

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 text character should not treated as normal sentence punctuation.

The patterns 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 character for the immediate left context of *p*, and *r* is the immediate right context character of *p*. The vertical bar (|) marks the start of a right context; if it is missing, the right context is assumed to be any nonalphanumeric character.

The *l* and *r* parts of a pattern may be Elly wildcards for matching. Those currently recognized in `stopException` are

_	matches a single whitespace character or beginning or end of text
@	matches a single letter
#	matches a single digit
~	matches a single nonalphanumeric character

A left context may have any of these wildcards; a right context can recognize only the whitespace wildcard (_). All nonwildcard characters in a pattern must be matched exactly, except for letters, which will be matched irrespective of case.

Here are some examples of patterns in `default.sx.elly`:

```
~@.|_
DR.
MR.
MRS.
U.S.S.|_
U.S.|_
```

The first pattern picks up initials, which consist of a single letter followed by a period and a space character. The other patterns match personal titles and work as you would expect them to. The file `default.sx.elly` has an extensive list of stop exceptions that might be helpful for handling typical text. You can of course supply your own stop exceptions here.

You should note that ordering makes a difference in the listing of patterns here. PyElly will always take the first match, which should do the right thing. You should, however, watch out for for patterns where it makes a difference what character precedes the precedes the match. In the case of `DR.`, the preceding character probably does not matter; but in the case of `@.`, it does. This is why the pattern needs to be `~@.`.

9.1.2 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.

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 always see standard punctuation.

9.2 Punctuation Syntax

The input sentences processed by ellyBase will currently include all punctuation, including those recognized by stopException as not breaking a sentence. When PyElly breaks a sentence into parts for analysis, its punctuation by default will be taken as single character tokens. PyElly will assign common English punctuation to the predefined syntactic type `PUNC` unless you provide vocabulary rules or `D:` grammar rules or pattern rules specifying otherwise.

For example, you might want to put `DR.` into you vocabulary table, perhaps as the syntactic type `TITLE`. Since this takes three characters from an input stream, including the period, PyElly will no longer see the punctuation here. The rule here is always to take longest possible match when multiple PyElly rules can apply.

The identification of punctuation as a sentence part is just the first step for PyElly analysis. The grammar rules for a PyElly application will then have to describe how to fit 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, you might choose just to ignore all punctuation, but in others, punctuation occurrences in sentence will provide important clues about the boundaries of phrases in text input. In the former case, one can have a grammar rule like

```
g:UNKN->PUNC
__
```

or alternatively, define macro substitutions that will make all punctuation marks disappear from input text.

In the latter case, one needs at least one grammar rule like

```
g:SENT->SENT PUNC

__
```

for handling stop punctuation terminating sentences, plus other rules for handling punctuation like "`,`" "`(`", "`)`", or single and double quotation marks, which may be internal to a sentence. You will have to decide how much work you want to do here.

Elly parsing will fail if all the parts of a sentence cannot be put into a single coherent analysis; and punctuation handling will be a highly probable failure point here.

10. PyElly Parsing

Parsing is usually invisible in PyElly operation. This is helpful in that it simplifies the task of setting up 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 technical look at the mechanics of PyElly parsing, which is quite highly evolved.

PyElly copies 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 interpret directly. In the early days of computing, all compilers were written from scratch; and the crafting of individual compilers was complex and slow, and the results were often unreliable.

To streamline and rationalize compiler development for the proliferation of new languages and new target machines, compiler-compilers were invented. These provided prefabricated and pretested components that could be quickly 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 work under the constraint of context-free languages; and the individual tokens in that language (variables, constants, and so forth) have to be recognizable by a finite-state automaton. Such restrictions are significant, but being able to develop a working compiler in weeks instead of months is so advantageous that almost everyone can live with them.

The LINGOL system of Vaughn Pratt adapted compiler-compiler technology to help build natural language processors. Natural language is not context-free, but life is more simple if we can parse them as if they were context-free and then take care of context sensitivities through other means like local variables in semantic procedures attached to syntax rules. PyElly follows the LINGOL model and carries it even further.

10.1 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 more simple structures. PyElly must build such a tree incrementally, starting either at the bottom with the basic tokens from a sentence or at the top by putting together different possible structures with `SENT` as root that we then have to match up with a given input sentence.

One can debate forever about whether bottom up or top down is better, but both should produce the same parse tree in the end. We can in fact 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 also be generated top-down. This approach was taken by LINGOL and by PyElly. Going bottom-up can often provide more helpful information when parsing fails, a common occurrence in computational linguistic analysis.

The PyElly bottom-up algorithm revolves around a queue listing the phrase nodes of a parse tree that still need to be processed to get the phrase nodes in the next higher level of a tree. Initially, this queue is empty, but we then read the next token in an input sentence and look it up to get bottom-level nodes for its possible interpretations. These new phrase nodes are then inserted into a queue to prime it.

PyEllying parsing then runs in a loop, taking the first node in its queue and applying its grammar rules to create new nodes, which will be put into the back of the queue for further processing. This procedure keeps running until the queue becomes empty again, although a queue will often get longer as result of taking off a node and processing it. With an empty queue, PyElly will then read the next token from a sentence and do the same thing as before.

There is one circumstance when a new node is not put into a queue. If there is already a node of the same syntactic type with the same syntactic features built up from the same sentence tokens, PyElly notes an ambiguity. The new node is then attached to the old node, and nothing further is done. Otherwise, PyElly would have to repeat everything for the old node with the new node without really accomplishing anything useful.

When an ambiguity is found, PyElly will execute the cognitive semantics for each of the nodes involved to compute plausibility scores. It will then put the most plausible node into a parse tree being built while still keeping the other nodes around for reporting, for possible backup on a semantic failure for a more plausible node, or for adjusting biases to insure that the same rule will always be taken when there are multiple rules with the same cognitive semantic plausibility.

10.2 Token Lookup

PyElly token lookup is complicated because of the many different ways that we can do this: external vocabulary tables, pattern rules, entity extraction rules, the dictionary rules built in a grammar table. This is further complicated by macro substitution rules, inflectional stemming, and morphological analysis. To take full advantage of these capabilities, you have to be aware of how the various lookup possibilities interact within PyElly.

The PyElly parser moves from left to right in a sentence taking one piece at a time. The extent of that piece is often a single word or a single punctuation mark, but it can be include multiple words and word fragments. PyElly will in general take the longest piece here to be the next token for a sentence analysis.

Here is how PyElly currently does lookup to get the next token in a given application `A` when at a particular position in an input sentence:

1. If the number rewriting option is turned on, try to rewrite a spelled out number in current sentence position as digits plus any ordinal suffixes like -ST, -RD, or -TH.

2. Look up the next piece of text in the external vocabulary table for `A`; put matches into the PyElly parsing queue with syntactic information and type and features.

3. Try to match up the next piece with the pattern table for `A`; put matches into the PyElly parsing queue.

4. Try entity extraction at the current position; put matches into the PyElly parsing queue.

5. If steps 2, 3, or 4 have found pieces of a sentence that are recognizable, take only the longest piece as the next token for sentence analysis and remove all queued nodes corresponding to any shorter pieces of the sentence.

6. Use the default PyElly procedure for extracting the next one-word token at the current sentence position. This may automatically do inflectional stemming and macro substitution.

7. If we already have a token from previous steps that is longer than one word, then stop any further lookup. Proceed instead to the main PyElly parsing algorithm.

8. Look up the next token in the external vocabulary table and in the internal dictionary for a grammar. If found, add phrase nodes for the token to the PyElly parsing queue. There will be multiple nodes here if there are multiple vocabulary table entries or dictionary rules.

9. If we already have any queued nodes to process, then proceed to main PyElly parsing loop.

10. Otherwise, try to break off any morphological endings from the current unknown token. If this succeeds put the reduced token and its suffixes back into the current sentence position and go back to Step 6.

11. Check if the next token is known punctuation. If so, enqueue a node for the punctuation syntactic type `PUNC` and proceed to main PyElly parsing.

12. If all else fails, then create a node of `UNKN` type for the next token and proceed to main PyElly parsing.

10.3 Building a Parse Tree

Given token lookup to put phrase nodes into the PyElly parsing queue, we are now ready to build a parse tree from the bottom up. The basic algorithm here is from LINGOL, but the same procedure shows up in other bottom-up parsing systems as well. The next subsection will cover the details of the basic algorithm, and the two subsections after that will describe how PyElly extends that algorithm.

10.3.1 Context-Free Analysis Core

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 go through a process of determining all the ways that the node will fit into a parse tree being built by PyElly. This is called “ramification” in PyElly source code commentary.

For newly enqueued phrase node, basic PyElly parsing will go through three steps:

1. If the syntax type of the node is `X`, look for partially satisfied rules of the form `Z->Y X` that have earlier set a goal of an `X` in the current position. For each such goal found, create a new 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 extent in a sentence as `X`.

3. Look for rules of the form `Z->X Y`. For each one, 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 a top-down algorithm would not have generated it. Each newly created node will be added to the the PyElly parsing queue for processing with these three steps again. When all the phrase nodes for the current sentence position have been taken care of, the parser moves 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 avoid any ambiguities in their grammar, but natural languages are full of them and we have to be ready here. In Pyelly, the solution is to resolve ambiguities outside of its ramification steps.

It is technically not an ambiguity when two phrase nodes of different types cover the same words in a sentence. For example, the single word THOUGHT could be either a noun or the past tense of a verb. This will probably lead to an ambiguity further up in a parse tree, but at the lowest level, PyElly does not have to do anything yet.

10.3.2 Special Modifications

Except for ambiguity handling, basic PyElly parsing is quite generic. We can be more efficient here by anticipating how grammar rules for natural language differ from those for context-free artificial languages. The first departure from the core algorithm is in the matching of syntactic features, which need not be exact.

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 extra checking has to be added to the basic PyElly parsing algorithm, but it is straightforward to implement.

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 two resulting nodes so that only step 3 of ramification will be taken for the root part and only step 1 will be taken for the

inflection part. A displayed parse tree will therefore be smaller than one might expect, which means faster parsing.

10.3.3 Type 0 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 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 `...[.0]->` . 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 `...[.1]->` . Enqueue it and get its ramifications.

Those reading this manual closely will note that the two rules here have syntactic features associated with `...`, which Section 7 said was not allowed. In fact, someone writing rules in a `*.elly` file cannot do this, and that is because PyElly needs those syntactic features to make logic above work properly.

The difficulty here is that the `...` syntactic type is prone to producing ambiguities. This situation 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 uses syntactic features with `...` here, but keeps it all invisible to users.

PyElly will also have to be able to distinguish between phrases not of type `...`, but built up from `...` subphrases that are empty or non-empty. The problem again is with inappropriate ambiguities, and the solution here is to propagate upward the syntactic feature `[.0]` that indicates an empty phrase due to case 1 and the syntactic feature `[.1]` that indicates an empty phrase due to case 2. That propagation will of course be invisible, but someone will need to guide it explicitly through setting the `*RIGHT` syntactic feature.

10.4 Success and Failure in Parsing

PyElly automatically defines the grammar rule:

```
g:SENT->SENT END
—
```

This rule will never be realized in an actual phrase node, but the basic PyElly parsing algorithm use this rule can set up goals for the syntactic type `END` in the normal course of processing. 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, it has succeeded, and the analysis that we want for an input sentence is the structure built up for the `SENT` phrase node that generated the `END` goal just found.

It is possible that there will be more than one `END` goal. This will not be an ambiguity in the PyElly sense, for these would already have been tied together in the course of PyElly sentence analysis. We actually can have multiple phrase nodes of type `SENT` here, but with different syntactic features. PyElly can still look at their cognitive semantic scores, however, and select the one with the highest plausibility and run its generative semantic procedure.

Failure in parsing means that there no generative semantic procedure to run and we no recourse except to dump out intermediate results and hope for some clue emerging. If the failure is due to something discovered in semantic interpretation, though, we can try to recover by backing up in a parse tree to look for an ambiguity and selecting a different alternative at that point.

10.5 Parse Tree Diagrams

If a sentence is unparsable or its semantic interpretation fails, PyElly can be set up to show the trees for all the syntax structures found in bottom-up parsing. This will show you where the building of a parse tree had to stop, which will tell you at least whether the problem is with a rule of some kind or with bad input text being encountered.

PyElly displays its parse trees and subtrees by writing to the standard error stream. This originally was an informal debugging aid, but has proved to be useful that it has been integral to PyElly operation. Trees will presented horizontally, with their highest nodes on the left and with branching shown vertically. For example, here is a simple 3-level subtree with 4 phrase nodes:

```
sent:0000──ss:8000┬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 phrase sequence number indicating the order in which it was generated, and `p` is a numerical the plausibility score computed for the node. The nodes are connected by Unicode drawing characters indicating the kind of branching.

In the above example, the top-level node here for type `sent` is

```
sent:0000
   6 =  3
```

This node above has all syntactic features turned off; it has the node sequence number of 6 and a plausibility score of 3. Similarly, the node for type `ss` at the next level is

```
ss:0001
 4 =  2
```

The actual sentence tokens for a PyElly will be in brackets on the far right, preceded by its sentence position, which always starts from 0. In the example above, these 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.

With analysis of words into components, 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 are shared by two actual sentence pieces HELLO and period (`.`).

All the examples here show complete sentences that presumably were chosen for semantic interpretation. If a PyElly analysis fails, however, someone will want to see all the results of parsing, including rejected ambiguities and dead ends. This can be accomplished by exploiting the sequence numbers of phrase nodes, which indicate their order of generation.

For a full dump, PyElly starts from the node with the highest sequence number and shows the tree below the node. All nodes shown nodes are marked as already seen. Then PyElly goes to the phrase node with next high sequence number that has not yet been seen and shows the subtree below this node. This continues until every unseen phrase node and its subtree has been dumped. Each node will appear only once in the dump.

In addition to trees and subtrees, a PyElly full dump will also show all goals generated in a sentence analysis and all ambiguities found in the process. This information should allow you to reconstruct how PyElly went about doing its parse of a sentence. Here is an example

```
> Dogs eat.
```

```
dumping from phrase 5 @0: type=0 [00 00] :0 use=0
with 4 tokens
sent:0000┬noun:0000┬noun:0000 @0 [dog]
   5 =  0|   3 =  0|   0 =  0
         |         └sufx:0000 @1 [-s]
         |             2 =  0
         └verb:0000 @2 [eat]
             4 =  0

dumping from phrase 1 @0: type=4 [00 00] :0 use=0
with 4 tokens
noun:0000 @0 [dog]
   1 =  0

0 goals at final position= 4

6 phrases altogether

ambiguities
noun  0000:  0 (+0/0)  1 (+0/0)

raw tokens= [[dog]] [[-s]] [[eat]] [[.]]
6 phrases, 5 goals
```

This is a full dump for an analysis where we have the grammar rule `SENT->NOUN VERB`, the `NOUN` DOG defined in two ways, and the `VERB` EAT. Parsing fails here for the input “Dogs eat.” because we made no provision for punctuation at the end of a sentence. In the dump, we first see the subtree for the first three tokens of the input sentence, followed by the subtree for the other interpretation of DOG. There are no goals at position 4 here. In the listing of ambiguities we have phrases 0 and 1, both identified as a `NOUN` type without syntactic features set.

10.6 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 most recent predecessors written in Java, C, or FORTRAN. PyElly takes full advantage of Python object-oriented programming and list processing with automatic garbage collection.

Unless you are running on a platform with extremely tight memory, PyElly should be able to handle sentences containing hundreds of tokens with no difficulty. Writing a grammar to describe such huge sentences will take considerable effort, however.

The main restrictions in PyElly are the ones on the total number of syntactic types (48) and the total number of different syntactic features for a phrase node (16). These are fixed to allow for preallocation of various arrays used by the PyElly parser and faster operation. You can changed them in the PyElly Python code, but they should be enough for ordinary applications.

11. Developing Rules and Troubleshooting Problems

PyElly should make it easier to build natural language applications, especially in English, but there are still a myriad of details to take care of, and you have to be able visualize how an application can be broken down into semantic procedures attached to individual grammar rules. This can often be perplexing and challenging, especially you are still learning the mechanics of natural language processing.

First, a word of reassurance: you are already a natural language expert! Even a young child knows more about language than the most powerful computer in the world. If you could just harness to the expertise some basic analytical skills and some programming chops, then you should do well with PyElly.

This section is a grab bag of advice about developing nontrivial PyElly grammars and vocabularies based on experience going back to the PARLEZ system, the oldest PyElly predecessor. These is no magic formula, but you can at least learn to avoid the major pitfalls and to make PyElly work for you instead of vice versa.

- PyElly is a simple system of only about fifteen thousand lines of Python code. It is designed to translate certain kinds of strings into other kinds of strings and do nothing more. In other words, you should not expect to be able to replicate Watson or Siri overnight. So go ahead and be ambitious, but stay realistic about how much you can accomplish in single project.

- PyElly analysis revolves around sentences, but remember that you define sentences however you want and need not follow established ideas. Try to make sentences out of smaller pieces of text in order to make a PyElly grammar more manageable.

- To see how PyElly behaves with a single input sentence given a language definition, run `ellyBase.py` directly instead of going through `ellyMain.py`. This works better interactively and also provides more diagnostic information.

- In developing a grammar, keep everything as simple as possible, because you will be more likely to run into trouble as the number of syntactic types and the number of rules increase. In many cases, you can ignore details like number, tense, and subject-verb agreement.

- To see what is currently in a grammar, run the module `grammarTable.py` with your application identifier as an argument, just as you would run `ellyBase.py`. This will give you even more diagnostic information, however.

- To see what is currently in a vocabulary, run the module `vocabularyTable.py` with your application identifier as an argument,

- 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, which you can replace later.

- When you build a grammar with semantic procedures, you are in fact programming. Therefore, follow good software engineering practices. Divide a large project into smaller parts that can be finished quickly and tested. Test as much as you can along the way; never put off all your testing until after all your language rules have been written.

- Natural language typically has regular and irregular forms. Tackle the regular forms first in your grammar rules and make sure you have a good handle here before taking on irregular forms. The latter will often can be handled by macro substitutions: for example, change SLEPT to SLEEP -ED.

- The semantic procedures for grammar rules are inherently modular. Try to spread out PyElly translation work to as many separate procedures as you can. Put common code of multiple procedures into named subprocedures.

- It can be helpful to group syntactic types into multiple levels where the semantic procedures at each level will do similar things. This is a good way to organize the definition of local variables for communication between different semantic procedures.

- Macro substitutions will usually be easier to use than syntax rules plus semantic procedures, but they have to be quite specific about the words that they apply to. Syntax rules are more appropriate for patterns that apply to general categories of words.

- Macro substitution rules can be quite dangerous if you are not careful. Watch out for infinite loops of macro substitutions, which can easily arise with `*` patterns. Multiple macros can also interact unexpectedly; make sure that no macro is reversing what another is doing.

- The ordering of rules in a macro substitution table is important. Rules further up in a list can change the input text that a macro further down the list is looking for.

- PyElly is set up so that macro substitutions take place just before the next word token is taken from its input buffer. When inflections and morphological prefixes and suffixes are split off from a word, the results will always be put back into the input buffer so that macros will have a chance to undo the splitting in special cases.

- Vocabulary building should be the last thing you do. You should define at least terms to support early testing, but hold off on the bulk of your vocabulary. You will do a better job here if do it after you are fairly sure of what your grammar rules are going to look like.

- For speed, avoid macros for matching literal phrases like “International Monetary Fund.” Unless you need the wild card matching supported by macros, just stick to vocabulary tables.

- Syntactic and semantic features are most useful for reducing the total number of syntactic rules, but add complexity to your grammar. Use different sets of feature names in different contexts for clarity, but you will be responsible for using the right set in every situation. Once specified, all features are stored as anonymous bits in PyElly rules; and PyElly cannot check for correctness. Watch out here for the inheritance of features due to the `*R` syntactic feature.

- Be liberal with named semantic subprocedures. It can be useful for them to have as few as two or three commands when they can be called by more than once. A subprocedure call could actually end up take more space than making its commands inline, but clarity will trump efficiency here.

- To see what an `A.g.elly` grammar looks like, run the unit test for the `ellyBase.py` module with `A` as an argument. This will list all currently defined symbols for syntax structure types, syntax rules, and internal dictionary entries. To see semantic procedures and named sub procedures, run the unit test for the `grammarTable.py` module with `A` as an argument. To dump out the entire saved grammar rule file, run `dumpEllyGrammar.py`. with `A` as an argument.

- PyElly can be more efficient in parsing if you define your syntax rules to be right-recursive. This means to use rules of the form `x->y x` instead of `x->x y`. Almost all natural languages are right-recursive; the main exception is Japanese. With the `...` syntactic type, try especially hard to avoid a rule like `...->... x`. in favor of `...->x ….`

- If you run into a parsing problem with a long sentence, try shortening it while still keeping the same problem. A PyElly parse tree display will be easier to read for debugging when a sentence is shorter.

- When a parse fails, the last token in the listing shown with a parse tree dump will show you where the failure occurred.

- Leave the PyElly tree dumps enabled and learn to read them. This will be the most valuable diagnostic information you can easily get if your language rules are not working as you expect.

- If you are working with English input and have not defined syntax rules for handling the inflectional endings -S, -ED, and -ING, then a parse may fail on them. The file `default.p.elly` will define these as `SUFX`, but you probably want something else.

- To follow the execution of a semantic procedure, insert a TRACE command into it. This will write to the standard error output stream.

- Punctuation is tricky to handle. Remember that a hyphen will normally be treated as a word break; for example, GOOD-BYE currently becomes GOOD, - , and BYE. An underscore or an apostrophe is not a word break, though. Please keep this in mind when writing syntax rules or macro substitutions.

- Macro substitution does not apply across sentence boundaries. To override punctuation otherwise seen as a stop, use the special PyElly stop exception rules described in Section 9. Note that macros do recognize embedded periods and commas, which are non-stopping punctuation.

- Macros are powerful, but can slow down processing significantly. This is because all macros have to be rechecked after any successful substitution except the null one, and actual substitutions involve much string copying.

- Ambiguity is often seen as a problem in language processing, but PyElly embraces it. Sometimes deliberate ambiguity can simplify a grammar. For example, a word like IN can be either a preposition or a verb particle. Define rules for both usages and let PyElly figure out which one to apply. Give ambiguous alternatives different plausibility scores; otherwise PyElly will switch between them in parsing different sentences, which is probably not what you want.

- 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 (i.e. bad) ways because plausibility for a phrase is computed by recursively adding up all the plausibility scores for all its constituents and subconstituents.

- Experiment. PyElly offers an abundance of language processing capabilities. Try them to find out what works for you.

The Jelly predecessor of PyElly also included Perl scripts that checked language definition files for common problems. Changes in PyElly have now eliminated the need for many of these checks, but such “lint” scripts may be brought back later.

12. PyElly Applications

PyElly by itself is no magic bullet for natural language despite its broad range of builtin capabilities. Yet what it does, it does well, and this can often be quite useful. A good PyElly application would meet the following conditions:

1. Your input data is UTF-8 Unicode text consisting of either Latin-1 or ASCII characters and divisible into sentences. This need not necessarily be English, but that is where PyElly has the most builtin capabilities.

2. Your intended output will be arbitrary Unicode in UTF-8 encoding, not necessarily in sentences.

3. No world knowledge is required in the translation of input to output except for what can be found in a published dictionary.

4. You can describe the translation process at least in words and just need some support in automating it.

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 everything else being treated as the `UNKN` syntactic type.

6. Your computing platform has Python 2.7.* installed. This will be needed both to develop your language rules and to run your intended application.

7. You are working experimentally and are willing to put up with idiosyncratic non-commercial software.

Familiarity with the Python language will help here, but is not mandatory. You will, however, definitely have to be able to write code for PyElly cognitive and generative semantics. This is nontrivial work, but should be straightforward for anyone who has had any training at all in actual programming.

Here are three fairly simple projects you can try as a way of getting to know PyElly:

- A translator from English to pig Latin.

- A bowdlerizer to replace objectionable terms in text with sanitized ones.

- A contextual spelling corrector.

The PyElly distribution includes various examples of applications originally developed to be run as part of system integration testing. You can use their language definition files as models for building your own application. For each listing below, we identify the language definition files currently associated with it.

default (.g,.m,.p,.ptl,.stl,.sx,.v) - not really an application, but a set of language definition files that will be substituted if a particular application does not specify one. This includes rules for sophisticated morphological stemming and vocabulary definitions for most of the terms in WordNet 3.0.

echo (.g,.m,.p,.v) - a minimal application that echoes its input as analyzed by PyElly into separate tokens. It shows the effect of inflectional and morphological stemming.

```
input:  Her reaction startled him.

output: her react -ion startle -ed him.
```

test (.g,.m,.p,.ptl,.stl,.v) - for basic testing with a vocabulary of short fake words for faster keyboard entry; its grammar defines simple phrase structures.

```
input:  nn ve on september 11, 2001.

output: nn ve+on 09/11/2001.
```

indexing (.g,.p,.ptl,.v) - to check stemming and stop word removal, but can be used to get roots of content words from input text. It could be used as a front end to predigest input text for information retrieval, statistical data mining, or machine learning systems.

```
input:  We never had the satisfaction.

output: - - - - satisfy -
```

texting (.g,.m,.p,.ptl,.stl,.v) - a test with a big grammar and nontrivial generative semantic procedures. This implements a kind of readable text compression similar to that seen in mobile messaging.

```
input:  Government is the problem.

output: govt d'prblm.
```

doctor (.g,.m,.ptl,.stl,.v) - a test with a big grammar with extensive ambiguity handling required. This emulates Weizenbaum's Doctor program for Rogerian psychoanalysis.

```
input:  My mother is always after me.

output: CAN YOU THINK OF A SPECIFIC EXAMPLE.
```

chinese (.g,.ptl,.stl,.v) - a test of Unicode input and output. It demonstrates some rudimentary Chinese grammar with both traditional [tra] and simplified [sim] characters.

```
input:  they sold those three big cars.

output: [sim]他們卖了那些三辆大汽车.

output: [tra]他們賣了那些三輛大汽車.
```

(Traditional output will be selected when ellyMain is run with the flag **-g tra** . The default is simplified output.)

For integration testing, the test, indexing, texting, doctor, and chinese applications each have input files *.main.txt to be run with ellyMain. The expected translations by each application are given by the files *.main.key. The chinese test should be run with the flag **-g tra** .

Other PyElly applications not yet implemented, but possible in the short term with the current version of PyElly are:

querying - rewrite natural language queries in SQL. This will a reworking of a system originally developed with the AQF predecessor of PyElly.

tagging - rewrite text with XML tagging.

name - extract personal names from input text. This may require some additional entity extraction support.

translit - transliterate English words into a non-Latin alphabet.

PyElly currently incorporates a conceptual hierarchy capability for use in cognitive semantics to resolve ambiguity. This has been written and tested at the unit level, but has not yet been tested at the system level or documented in this manual. An application taking advantage of conceptual hierarchies based on WordNet 3 is in the works.

disambig - disambiguate terms with a conceptual hierarchy by checking the semantic context of an ambiguous term.

As can be seen, there are many possibilities here, some even with potential commercial value, though that is incidental. Also there is nothing to preclude further extension of PyElly to support a wider range of applications. Something for an extended PyElly over the longer term might be

madlib - an implementation of the popular party game.

This is all just to give you an idea of what PyElly can do. It is old-fashioned as technology goes, but give it a try and see whether it can help you learn something about natural language processing or provide some useful analytic support for some of your actual projects.

A. Technical Appendix

This appendix is for Python programmers. Although 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 do anything with it needed.

First of all, PyElly was written in Python 2.7.5 under Mac OS X 10.9. To support its external vocabulary tables, PyElly also requires the Berkeley DB open-source database manager and the bsddb3 third-party Python package for accessing Berkeley DB.

Currently, the PyElly source code consists of 57 modules, each a 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`.

Here is listing of all PyElly modules grouped by functionality. Some non-python definition and unit test data files are included in a group when they are integral to the modules there.

Characters and Wildcards	
`ellyChar.py`	methods for working with ASCII plus Latin-1 alphabet as Unicode
`ellyWildcard.py`	text wildcards for pattern matching

Inflectional Stemmer	
`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
`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

Tokenization	
ellyToken.py	class for linguistic tokens in PyElly analysis
ellyBuffer.py	for manipulating text input
ellyBufferEN.py	manipulating text input with English inflection stemming
substitutionBuffer.py	for manipulating text input with macro substitutions
macroTable.py	for storing macro substitutions
patternTable.py –	extraction and syntactic typing by FSA with pattern matching

Rule Definition Support	
ellyDefinitionReader.py	read input text with reformatting and normalization
definitionLine.py	for parsing Elly -> or <- rules

Parsing	
symbolTable.py	for names of syntactic types, syntactic features, procedures, global variables
syntaxSpecification.py	interpret syntax specification
featureSpecification.py	interpret syntactic (and semantic) features
grammarTable.py	for grammar rules and internal dictionary entries
grammarRule.py	for representing rules
derivabilityMatrix.py	for establishing derivability of one syntax type from another
ellyBits.py	bit-handling for parsing implementation
parseTreeBase.py	low-level parsing structures and methods
parseTreeBottomUp.py	bottom-up parsing structures and methods
parseTree.py	top-level parsing with core PyElly parsing algorithm
parseTreeWithDisplay.py	with methods to dump parse tree data for diagnostics

Semantics	
generativeDefiner.py	define generative semantic procedure
generativeProcedure.py	generative semantic procedure
cognitiveDefiner.py	define cognitive semantic procedure
cognitiveProcedure.py	cognitive semantic procedure
semanticCommand.py	cognitive and generative semantic codes
conceptualHierarchy.py	support for ISA relation tests in 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 exceptions in text
exoticPunctuation.py	recognize nonstandard punctuation
punctuationRecognizer.py	recognize standard punctuation as default

Morphology	
treeLogic.py	decision logic base class for affix matching
suffixTreeLogic.py	for handling suffixes
prefixTreeLogic.py	for handling prefixes
morphologyAnalyzer.py	handle morphological analysis of tokens

Entity Extraction	
entityExtractor.py	runs extraction procedures
extractionProcedure.py	predefined PyElly extraction procedures
simpleTransform.py	basic support for text transformations and handling spelled out numbers
dateTransform.py	recognize and normalize dates
timeTransform.py	recognize and normalize times of day

External Database	
vocabularyTable.py	interface to external vocabulary database
vocabularyElement.py	external vocabulary record

Top Level	
ellyConfiguration.py	define PyElly processing parameters
ellySession.py	save parameters of interactive session
ellyDefinition.py	language rules and vocabulary handler
interpretiveContext.py	handles integration of sentence parsing and interpretation
ellyBase.py	module for sentence processing
ellyMain.py	top-level main module with sentence recognition
dumpEllyGrammar.py	methods to dump parts of a grammar table

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 test for building of generative semantic procedures
cognitiveDefinerTest.txt	to support unit test for building of cognitive semantic procedures
suffixTest.txt	to support comprehensive unit test with default suffix tree logic
morphologyTest.txt	to support unit test with prefix and suffix tree logic
sentenceTestData.txt	to support unit test of sentence extraction

Author's Note.

The Python code in PyElly was written in 2013 with some minor preparatory work done in November and December of 2012. It is an extensive reworking and expansion of the Java code in its Jelly predecessor and is now incompatible with Jelly. The current version has been tested extensively, but it still should be considered as BETA.