派業歴

PyElly User's Manual

For Release v1.3.5.2

15 December 2015

Clinton P. Mah

Walnut Creek, CA 94595

Table of Contents

1. Introduction 5
2. The Syntax of a Language 9
3. The Semantics of a Language 13
4. Defining Tables of PyElly Rules 17
4.1 Grammar (A.g.elly) 19
4.1.1 Syntactic Rules 19
4.1.2 Grammar-Defined Words 20
4.1.3 Generative Semantic Subprocedures 21
4.1.4 Global Variable Initializations 21
4.2 Special Patterns (A.p.elly) 22
4.3 Macro Substitutions (A.m.elly) 25
5. Operations for PyElly Generative Semantics 27
5.1 Insertion of Strings 27
5.2 Subroutine Linkage 27
5.3 Buffer Management 28
5.4 Local Variable Operations 28
5.5 Local Variable Set-Theoretic Operations 30
5.6 Global Variable Operations 30
5.7 Control Structures 30
5.8 Character Manipulation 32
5.9 Selection From a Table 34
5.10 Buffer Searching 35
5.11 Execution Monitoring 35
5.12 Capitalization 36
5.13 Semantic Subprocedure Invocation 36
6. Simple PyElly Programming Examples 38
6.1 Default Semantic Procedures 38
6.2 A Simple Grammar Example 39
7. Running PyElly From a Command Line 43
8. Advanced Capabilities: Grammar 48
8.1 Syntactic Features 48
8.2 The ... Syntactic Type 50

Table of Contents

9. Advanced Capabilities: Vocabulary 53
9.1 More on the UNKN Syntactic Type 53
9.2 Breaking Down Unknown Words 54
9.2.1 Inflectional Stemming 54
9.2.2 Morphology 57
9.2.2.1 Word Endings (A.stl.elly) 57
9.2.2.2 Word Beginnings (A.ptl.elly) 59
9.3 Entity Extraction 60
9.3.1 Numbers 61
9.3.2 Dates and Times 61
9.3.3 Names of Persons (A.n.elly) 62
9.3.3.1 Explicit Name Component Patterns and Types 63
9.3.3.2 Implicit Name Components 65
9.3.4 Defining Your Own Entity Extractors 67
9.4 PyElly Vocabulary Tables (A.v.elly) 68
10. Logic for PyElly Cognitive Semantics 72
10.1 The Form of Cognitive Semantic Clauses 74
10.2 Cognitive Semantic Approaches 75
10.2.1 Fixed Scoring 75
10.2.2 Semantic Features 76
10.2.2.1 Semantic Features in Cognitive Semantic Clauses 77
10.2.2.2 Semantic Features in Generative Semantics 78
10.2.3 Semantic Concepts 78
10.2.3.1 Concepts in Cognitive Semantic Logic 79
10.2.3.2 Semantic Concepts in Language Definition Files 81
10.2.3.2.1 Conceptual Hierarchy Definition (A.h.elly) 81
10.2.3.2.2 Semantic Concepts in Grammar Rules 82
10.2.3.2.3 Semantic Concepts in Vocabulary Table Entries 82
11. Sentences and Punctuation 84
11.1 Basic PyElly Punctuation 85
11.2 Extending Sentence Recognition 87
11.2.1 Stop Punctuation Exceptions (A.sx.elly) 87

Table of Contents

11.2.2 Exotic Punctuation 88
11.3 How Punctuation Affects Parsing 88
12. PyElly Parsing and Tokenization 90
12.1 A Bottom-Up Framework 90
12.2 Token Extraction and Lookup 92
12.3 Building a Parse Tree 93
12.3.1 Context-Free Analysis Main Loop 93
12.3.2 Special Modifications 94
12.3.3 Type 0 Grammar Extensions 95
12.4 Success and Failure in Parsing 95
12.5 Parse Data Dumps and Tree Diagrams 96
12.6 Parsing Resource Limits 100
13. Developing Language Rules and Troubleshooting 102
13.1 Pre-Checks on Language Rule Files 102
13.2 A General Application Development Approach 104
13.3 Miscellaneous Tips 104
14. PyElly Applications 111
Appendix A. Python Implementation 118
Appendix B. Historical Background 123
Appendix C. Berkeley Database and SQLite 125
Appendix D. PyElly System Testing 127
Appendix E. PyElly as a Educational Tool 130

1. Introduction

PyElly is an open-source software tool for creating computer scripts to analyze and rewrite English and other natural language text. This processing 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. In particular, PyElly as a preprocessor can take care of pesky low-level details of language in text data that often get in the way of more effective exploitation.

PyElly can also be helpful if you just want to gain experience with the nitty-gritty of language processing. You can quickly write scripts to do learning 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, or even re-creating the storied Doctor simulation of Rogerian psychoanalysis.

We have been building such natural language applications since computers were only a millionth as powerful as they are today. The overall problem here remains quite challenging, however, and even the most advanced technology may still require you to work hard to develop the algorithms and linguistic knowledge to accomplish something rather basic. PyElly aims to expedite such busy work through ready-made tools and resources, all integrated in a single free open-source 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 benefit from having a diversity of reliable methods addressing the problem. Also, though PyElly is all new code, it is really a legacy system, with many core components having roots as far back as 40 years. This sounds quite ancient, but language changes slowly, and having many mature software tools can make it easier to work with text data.

The impetus for PyElly and its predecessors came from observing that many natural language systems tend to run into the same subproblems. For example, information retrieval and machine learning with text data can work better when we can reduce its words of the text into their roots. So, instead of contending with variants like RELATION, RELATIONAL, RELATIVELY, and RELATING, a system could have just RELATE. This is of course the familiar stemming problem, but available free resources to correlate such word variants have often been disappointing.

A stemmer of course is not hard to build, but it takes time and commitment to do a good job, and no one wants to repeat this from scratch for every new project. That is true of other basic language processing capabilities as well. So, it seems logical to pull together at least some kind of reusable software library here, but it would be even more helpful if we could integrate our tools and resources more closely. PyElly does that.

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

though, because of its range of natural language support, including stemming, tokenizing, entity extraction, sentence recognition, idiomatic transformation, rule-driven syntactic analysis, and semantic ambiguity handling.

Effective use of PyElly will definitely require some linguistic expertise. You will have to be able to define the details of the language processing that you want, but much of the basics here have been prebuilt in PyElly if you are working with English input. The standard PyElly distribution includes the language definition scripts for eight different example applications that you can modify to get a head start in constructing your own.

The current PyElly package consists of a series of Python modules in sixty-four source files. The code 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 GitHub under a standard BSD license; you may freely modify and extend it as needed. Though intended mainly for education, there are no restrictions on commercial usage.

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

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

PyElly rules will 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 get desired output. Creating such rules requires some trial and error, but usually should be no more difficult than setting up macros in a word processor. PyElly will get you started properly here and also provide debugging aids to help track down problems that may arise.

Many natural language system building tools, especially those in academic research, aim to address the most thorny problems in language interpretation. These are opportunities for impressive processing gymnastics and often lead to knotty theoretical papers without necessarily producing any tools for everyday use. PyElly tries here to be simple and pragmatic instead. In response to classically tough sentences like "Time flies like an arrow," it is all right just to respond with "Huh?"

PyElly is compact enough to run on mobile devices with no cloud connection, 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 number of rules actually defined. A major project may involve hundreds of grammar rules and thousands of vocabulary elements, but some useful text analyses require just a few dozen rules and little explicit domain vocabulary.

What is a grammar, and what is a vocabulary? A vocabulary establishes the range of words you want to recognize; a grammar defines how those words can be arranged into sentences of interest. You may also specify idiomatic rewriting of particular input word sequences prior to analysis as well as define patterns to make sense of various classes of unknown words. For example, you can recognize 800 telephone numbers or Russian surnames ending in -OV or -OVA without having to list them all out.

This manual will explain how to do all of this and also introduce some basics of language and language processing that every PyElly user should know. To take advantage of PyElly, you should already be able to create and edit text files for whatever system you choose to work on and set up file directories where PyElly can find your rules. You should also be comfortable with Python coding. In an ideal world, an interactive development environment (IDE) could make everything easier here, but that is yet to be.

Currently, PyElly is biased toward English input, although it can read the entire Latin-1 subset of Unicode plus some extra characters as input. That 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 nominally English data, since we often encounter terms with foreign spellings like NÉE or RÉSUMÉ.

As any beginning student of a language soon learns, its rules are often messy. Irregularities trip up someone trying to speak or write mechanically from a simple grammar. PyElly users will face the same kind of problem, but by working generally at first and dealing with exceptions by special rules as they show up, we can evolve our language processing competence to reach some useful level of parlance. There is no royal road to natural language processing, but persistence can lead to progress over time, and PyElly can help to keep you on track here in sustained efforts.

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 expected of 21st-Century high school graduates, linguistic knowledge as might be picked up from a first course in a foreign language, and willingness to learn. You should start out with simpler kinds of PyElly processing, gaining the experience to progress to more complex analyses.

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

PyElly ("Python Elly") was inspired by the Eliza system created by Joseph Weizenbaum over 50 years ago for modeling natural language conversation, but PyElly has a completely different genesis. 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 and the ideas of cognitive and generative semantics come from Vaughn Pratt's LINGOL (LISP, 1973). Frederick Thompson's REL system (1972) also influenced the design of PyElly.

The PyElly website is at https://sites.google.com/site/pyellynaturallanguage/ . This will show results of actual PyElly processing with different sets of language definition rules. The latest PyElly source files can be downloaded from https://github.com/prohippo/pyelly.git

2. 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 some context. In general, not all combinations of symbols will make a meaningful sentence; for example, "Cat the when" is nonsense in English. To define a language that you want PyElly to process, you must first identify those combinations of symbols that do make sense and then assign suitable interpretations to them.

If a language is small enough, such as the repertory of obscene gestures, we can simply list all its possible "sentences" and write down what each of them mean. Most nontrivial languages, though, have so many possible sentences that this approach is impractical. Instead one must note that languages tend to have regular structures; and by identifying those structures, a computational linguist 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 establishes 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 and words in school, but formal grammars go into much more detail. They commonly are organized as sets of syntactic rules describing the particular kinds of language structure in sentences. Such syntactic rules will provide a basis for both generating and recognizing the sentences of a language with a computer.

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

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

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

```
    W
   /|\
  / | \
X   Y   Z
```

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

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

```
X->word
X->Y
X->Y Z
```

where `X`, `Y`, and `Z` are structure types and `word` is a word or some other kind of vocabulary element.

For example, to express a more complex rule like

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

we can instead use the pair of restricted rules

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

where `T` is a unique intermediate structure introduced solely to stand for a `B` followed by a `C`.

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

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

This uses all three types of PyElly restricted grammar rules.

The structure of a sentence as implied by these 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. By convention, the tree is always upside down, and the bottom of the tree will show the individual words of the sentence. For example, the sentence "It is red" would have the following diagram here:

```
                SENTENCE
                   |
   +------------+------------+
   |                         |
 SUBJECT                  PREDICATE
   |                         |
   |                  +-------+--------+
   |                  |                |
 PRONOUN           COPULA         ADJECTIVE
   |                  |                |
   |                  |                |
  it                 is               red
```

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

Our simple grammar above so far describes only a single sentence, but we can extend the range of its coverage by adding more rules for new kinds of structures and vocabulary. For example, the new rules

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

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

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

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

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

Technically speaking, PyElly grammar rules as described here define a "context-free language." With such a grammar, it is hard to correlate the possibilities for a given structure with the possibilities for a parallel structure in a parse tree. 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 attributes 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 context-free rule here by itself will not allow us to restrict a `SENTENCE` to have only certain combinations of subjects and predicates for agreement. We could of course write explicitly correlated rules like

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

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

Even though English and other natural languages are not context-free in the theoretical sense, we still want to treat them that way for practical reasons. The advantage in doing so is that we can then apply many sophisticated techniques developed for parsing artificial computer languages, which do tend to be context-free. This is the road taken in PyElly and its predecessors.

For convenience, PyElly also incorporates semantic checking of the results of parsing and allows some shortcuts to make grammars more concise (see Section 8). 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 problematic 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 in terms of simpler structures. The challenge of defining a PyElly grammar is to capture enough of such simpler structures in grammar rules to support a proper analysis of any input sentence that we are likely to see.

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.

3. The Semantics of a Language

The notion of meaning has always been difficult to talk about. It can be complicated even for individual sentences in a language, because meaning involves not only their grammatical structure, but also where it is used and who is using it. A simple expression like "Thank you" can take on different significance, depending on whether the speaker is a thug collecting extortion money, the senior correspondent at a White House 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 practical 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 handle references to subjects like sex, golf, or the current weather.

Here we shall limit the scope of semantics even more drastically: PyElly will deal with the meaning of sentences only to the extent of being able to translate them into some 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 implement useful kinds of language processing.

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

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

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

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. Instead, we must break the problem down and take 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.

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

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

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

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

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

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

Local variables will have a limited scope such 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 either directly or indirectly. When there are multiple active declarations of a variable with a given name visible to a semantic procedure, the most recent one applies. Upon exit from a procedure, all of its local variables immediately become undefined.

Here are some semantic procedures to illustrate how a PyElly translation might actually work. Suppose that we define the five grammar rules

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

With these rules, we can implement a simple translator from English into French with the five semantic procedures below, defined respectively for each rule above. For the

time being, the commands in the procedures will be expressed in ordinary English. These commands will control the entry of text into some output area, such as a text field in a window of a computer display. Here are some possible procedures:

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

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

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

```
PRONOUN->they
```

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

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

We need more elaborate semantic procedures here to get correct agreement. This is where various other PyElly semantic commands have to come in; and in fact, we shall use local variables to pass information about number and person between the semantic procedures for our syntactic structures to govern their translations (see Section 6). Nevertheless, the overall PyElly framework of semantic procedures attached to each syntactic rule and called recursively will remain 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 at the right time. We can make the problem easier here by taking care to have lower-level structures be parts of only a few higher-level structures, but this will still require some advance planning.

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

4. Defining Tables of PyElly Rules

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

The principal PyElly rules fall into six main types: (1) grammar, (2) vocabulary, (3) macro substitutions, (4) patterns for determining syntactic types, (5) name components, and (6) 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 context of use; macros support particular users of a system, and special patterns and name components tend to be specific to given applications. Separate tables of rules make it easier to tailor PyElly processing for different environments while allowing parts of language definitions to be reused.

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

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

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

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

An important element of most language rules will be syntactic structure names, seen in Section 2. We shall also call them "syntactic types" or "parts of speech," but they will be more general than what we learned in grade school. The current implementation of PyElly in Python can handle up to 64 different syntactic types in its input files. Six of these types, however will be predefined by PyElly with special meanings.

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

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

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

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

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

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

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

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

4.1 Grammar (`A.g.elly`)

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

4.1.1 Syntactic Rules

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

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

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

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

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

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

4.1.2 Grammar-Defined Words

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

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

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

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

a. The `D:` is mandatory in order to distinguish a word rule from a grammatical rule of the form `X->Y`.
b. The underscore separators are the same as for syntactic rules. A word definition may also have both 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, number, or other text token, possibly hyphenated. Multi-word terms in an application must be defined in PyElly's external vocabulary or stitched together by grammar rules, macro substitution rules, or other mechanisms discussed in Section 9.

4.1.3 Generative Semantic Subprocedures

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

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

A subprocedure definition will have the following form:

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

a. Note the absence of any arrow, either `->` or `<-`, in the first definition line.
b. A procedure name `n` should be a unique string of alphanumeric characters without any spaces. It can be of any non-zero length. The case of letters is unimportant. Duplicate definitions for the same procedure name will be reported as an error.
c. The underscore separators are the same as for syntactic rules and word definitions and will both be mandatory for a subprocedure definition.
d. A subprocedure definition may have only generative semantics. Cognitive semantics will not apply to a subprocedure and will always be ignored if specified.

4.1.4 Global Variable Initializations

PyElly global variables in a generative semantic procedure can be set in various ways. When such variables store important parameters referenced in a particular application grammar, it is helpful to be able to define them within the definition file for that grammar. In that way, the definition will be more readable and more easily maintained. The startup initialization of global variable `x` to the string `s` is accomplished by a `I:` line in a grammar definition file:

```
I: x = s
```

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

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

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

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

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

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

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

The key concept in an FSA is that of a state, which sums up how far along a text string the FSA has so far managed to match and what patterns it should look for next. An FSA will typically have multiple states, but one and only one will always be the starting state when the FSA is looking at the front of an input text with nothing yet matched. The total number of different states must be limited, hence the finiteness of an automaton.

In any given state, a PyElly FSA will have a list of patterns with possible wildcards to check against the input text at its current position. A wildcard will be a pattern element able to match against more than one text character; for example, any digit 0-9. PyElly wildcards are similar to the wildcards used in regular expressions, but are defined specifically for natural language processing; they will be listed below. This departs from the usual operation of FSA; which typically allow no wildcards.

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

Some matches will have no next state in a PyElly FSA table, but instead will specify a syntactic structure type. If an FSA has also reached the end of an input token, then a match is complete, and PyElly can assign the given structure type to the token being matched. At this point, a normal FSA would be done, but PyElly will also have to examine all the matching possibilities arising from multiple next states for an FSA.

PyElly continues until all reachable states and patterns have been checked or until the FSA runs out of input. At that point, PyElly will return a positive match length if any final state has been reached at the end of a token; 0, otherwise.

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

Here is a simple PyElly file of some pattern rules:

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

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

This recognizes 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 will have four parts as follows:

```
state    Pattern    Syntactic Type    next
```

a. The first part is an integer ≥ 0 representing a current PyElly automaton state.
b. The second part is a pattern, which may be an 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 have explicit characters and also various wildcards, which can match various substrings. PyElly wildcards will be 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 letters or digits in a sequence
&#	will match one or more digits in a sequence
&@	will match one or more letters in a sequence
^	will match a single vowel
%	will match a single consonant
'	will match an apostrophe appearing either as `'` (ASCII) or `'` (Unicode right single quotation mark) or `′` (Unicode prime)
$	will match the end of a word, but does not add to the extent of any matching

d. A pattern consisting only of `\0` (ASCII NUL) is special. It will cause an automaton to move immediately to the next state indicated without any matching. That next state must be present.
e. 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. The pattern `[a]b` will match the string `ab` or the string `b`.
f. A pattern with a wildcard other than `\0` or `$` by itself must always match at least one character; for example, the pattern `[a]*` will be rejected.
g. All final state patterns not ending with the `*` or `$` wildcards will have a wildcard `$` appended automatically.
h. The third part of a pattern line is a syntactic structure type (part of speech) like `NOUN`. A '–' here means that no type is specified. A non-final state may not specify any type.
i. The fourth part is the next state to go to upon matching a specified pattern. This will be an integer ≥ -1. A `-1` here means a final state.

For example, the pattern `###-##-####$` matches Social Security numbers, while the pattern `(###)###-####$` matches a telephone number with an area code, with no separating spaces. See Subsection 9.2.1 for more on possible number patterns. To match a character like `&` in a string, it may be specified explicitly for matching in a pattern by escaping it with a backslash character: `\&`. This will not be interpreted as a wildcard.

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

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

The main difference in PyElly macro substitution versus *Software Tools* is that substrings to be replaced can be described with wildcards along with explicit characters to match and that substrings to replace parts of the original can specify the parts of the original string that matched wildcards.

Macro substitution provides a convenient way of handling idioms, synonyms, abbreviations, and alternative spellings and of carrying out simple syntactic transformations awkward within the framework of context-free grammar rules. The general way to define a PyElly macro substitution rule is as follows:

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

a. Each macro definition is limited to a single line. Since macros will be in their own `*.m.elly` file, we need no marker at the beginning of each line.
b. The left and right sides of a substitution may have an arbitrary number of components, each being separated from the others by a blank.
c. The `P`, `Q`, `R`, `A`, `B`, `C`, and `D` are character strings containing no spaces. There must be at least one such string on the left side of a substitution and zero or more strings on the right side.
d. Upper and lower case is significant only on the right side.
e. Input words matching the pattern on the left side will be replaced by the right side.
f. The left side of a substitution rule may have patterns with wildcards; these will be the same as recognized in the special patterns described by the preceding subsection (3.2).
g. A pattern may also have '_' as a wildcard, which will match a single space character. This is not in wildcard list for the special patterns above because macro patterns can match multiple words, while special patterns can match only single words.
h. A `\\1`, `\\2`, `\\3`, and so forth, on the right stands respectively for the first, second, third, and so forth, parts of text matched by wildcard patterns on the left. PyElly allows up to nine bindings for a pattern. Each binding applies to a sequence of contiguous wildcards, except for _ and `'`, which will always be bound singly. For example, the pattern `#@abc@#`on a match will associate the first digit and letter of a match with `\\1` and the last letter and digit with `\\2`. Matching the pattern `#a'_*` with a string will associate the apostrophe in the string with `\\1` and the following space with `\\2`.

i. When any macro is matched, its substitution will be done. Then all macros will be checked again against the modified result for other possible substitutions. When a macro eliminates its match entirely, though, further substitutions will be ended at that position.
j. The order of macro definitions is significant. Those defined first in a definition file will always be applied first, possibly affecting the applicability of those defined afterward. Macros starting with a wildcard will always be checked after all others, however.
k. Macros have no associated semantic procedures; they run outside of PyElly parsing and rewriting.

Macro substitutions will be 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 careless. Nevertheless, macros can greatly simplify a language definition when you use them properly and keep their patterns fairly short.

They will be applied to the current PyElly input text buffer each time before the extraction of the next token to be processed. This can in effect override any tokenization rules in effect and can modify any stemming of words. All of that can add up to substantial overhead if you define many macros because all possible substitutions will always be tried out at each possible token position.

Here are some actual PyElly macro substitution rules from its `texting` example application, which tries to compress input as much as possible while keeping it readable:

```
*'ll -> \\1
percent* -> %
will not -> willnot
greater than or equal to -> >=
carry -ing -> with
receiv[e]* -> rcv\\1
#* @[.]m$ -> \\1\\2m
```

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

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

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

which inserts `RS` into the PyElly text input buffer after a matching comma (`,`) when followed by a participial -ING expression. PyElly will automatically recognize `RS` as the syntactic category `SEPR`, but the grammar rules for an application will then have to do something with this.

You often can use macro substitutions to handle idioms or other irregular forms that are exceptions to general language rules. They will always be applied after any inflectional stemming, but before any morphological stemming (see Subsection 9.1) with an unknown text element. Use them with care; PyElly will warn you when something is possibly dangerous, but will not stop you from doing something catastrophic.

5. Operations for PyElly Generative Semantics

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

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

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

5.1 Insertion of Strings

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

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

BLANK                    # put a space character into buffer

SPACE                    # same as BLANK

LINEFEED                 # start new line in buffer, add space

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

5.2 Subroutine Linkage

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

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

RIGHT                    # calls the semantic procedure
                         # for subconstituent structure
                         # Z for a rule of the form
                         # X->Y Z, but Y for rule X->Y
```

```
RETURN                    # returns to caller

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
```

5.3 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 their contents are finally 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,
                          # deallocating the new one
```

These in effect 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"
                          # (the divider / here may be replaced by
                          #  any char not in string1 or string2)
```

5.4 Local Variable Operations

Local variables can store a Unicode string. 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 string
```

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

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

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

SET x SP                  # set variable x as single
                          # space char; this will define x
                          # if not already defined
```

Note the absence of the equal sign (=) here. PyElly will recognize `SP`, `HT`, `LF`, and `CR` as space characters here. This form can also be used with the `IF`, `ELIF`, `WHILE`, and `BREAKIF` semantic operations described below. You may write `VAR` as shorthand for `VARIABLE`; they are equivalent.

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

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

QUEUE   q=x               # appends the entire string stored
                          # in local variable x to any string
                          # stored already in local variable q

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

The equal sign (=) must appear with `SET` and `VARIABLE` even when a second argument is missing; this is also required for `UNQUEUE` and `QUEUE`. If a lefthand local variable is undefined by a `SET` or `ASSIGN` operation, it will become automatically defined in the scope of the current generative semantic procedure.

5.5 Local Variable Set-Theoretic Operations

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

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

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

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

5.6 Global Variable Operations

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

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

GET x y                   # the inverse of PUT
```

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

5.7 Control Structures

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

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

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

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

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

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

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

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

```
IF x SP                    # check if local variable x is
                           # a space character

ELIF x SP                  #

WHILE x SP                 #
```

Instead of `SP`, you may also have `HT`, `LR`, or `CR`.

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

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

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

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

The strings to be compared against here must be separated by a comma (`,`) followed by a space. The space is essential for PyElly to recognize the listing here. The tests here can be negated with a tilde (~) also. The checking of multiple space characters `SP`, `HT`, `NL`, and `CR` as described above is not supported here.

Within a `WHILE` loop, you may also have

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

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

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

5.8 Character Manipulation

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

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

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

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

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

PEEK x <                # get a single char from
                        # start of next output buffer
                        # without removing it
```

```
PEEK > x                # gets a single char from
                        # end of current output buffer
                        # without removing it

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

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

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

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

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

If `n` is omitted for the `EXTRACT` operation, it is assumed to be 1. If the < or > are omitted from a `DELETE` or a `SHIFT`, then < is assumed; if `n` is omitted here, it means to delete or shift the entire source buffer. All deleted text can be recovered by a `STORE` command.

The `DELETE` operation also has two variants

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

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

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

5.9 Selection From a Table

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

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

The table argument is a literal string of the form

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

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

For example, the particular PICK operation

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

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

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

but the IF-ELSE form takes up multiple lines. A PICK table will also be saved in a generative semantic procedure as a Python hash object for faster lookup.

The operation

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

will append dddd for any x, including x being set to the null string.

5.10 Buffer 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 will be shifted to the
                        # current buffer up to and including
                        # the first occurrence of string s

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

Substring `s` must be given and may contain no spaces. If it `s` not found in a buffer scan, the entire contents of the buffer will be moved. If final < or > is omitted, then > is assumed. Note that `s` can include the characters < or >, which can look confusing, but should be interpreted as what to search for.

5.11 Execution Monitoring

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

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

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

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

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

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

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

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

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

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

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

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

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

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

When executed in a generative semantic procedure, `VIEW 4` will write the following kind of message to the standard error stream:

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

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

5.12 Capitalization

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

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

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

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

The restrictions for `CAPITALIZE` apply here also.

5.13 Semantic Subprocedure Invocation

If `DO` is the name of a semantic subprocedure defined with `P:` in a PyElly grammar table, then it can be called in a generative 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. This definition does not have to come before the call. When a subprocedure finishes running, execution will return to the point just after where it was called. Any local variables in the subprocedure will then become undefined.

A subprocedure call will always take no arguments. If you want to pass parameters, you must do so through local or global variables or in a buffer. Results from a subprocedure can be returned only by putting them into an output buffer or passing them back in a local or global variable.

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

6. Simple PyElly Programming Examples

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

6.1 Default Semantic Procedures

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

```
_
  LEFT
  RIGHT
__
```

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

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

```
_
  LEFT
__
```

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

```
_
  OBTAIN
__
```

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

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

```
_
  RIGHT
  LEFT
__
```

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

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

6.2 A Simple Grammar Example

We now give an example of a nontrivial PyElly grammar. The problem of making subjects and predicates agree in French came up previously in Section 3. Here we make a start at a solution by handling *elison* and the present tense of first conjugation verbs in French 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     # can be 1, 2, or 3
  VAR NUMBER=s     # singular or plural
  LEFT             # for subject
  SPLIT
  RIGHT            # for predicate
  BACK
  IF PERSON=1
    IF NUMBER=s
      EXTRACT X <  # letter at start of predicate
      IF X=a, e, è, é, i, o, u
        DELETE 1   # elison j'
        APPEND '   #
      ELSE
        BLANK      # otherwise, predicate is separate
        END
      INSERT < X   # put predicate letter back
      END
    ELSE
      BLANK        # predicate is separate
      END
  ELSE
    BLANK          # predicate is separate
    END
  MERGE            # combine subject and predicate
  APPEND !
__
```

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

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

```
G:SUBJ->PRON

__
```

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

```
D:i<-PRON

_
  APPEND je
  SET PERSON=1

__
D:you<-PRON

_
  APPEND vous
  SET PERSON=2
  SET NUMBER=p

__
D:it<-PRON

_
  APPEND il

__
D:we<-PRON

_
  APPEND nous
  SET PERSON=1
  SET NUMBER=p

__
D:they<-PRON

_
  APPEND ils
  SET NUMBER=p

__
```

These internal dictionary 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:plural

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

__
P:1cnjg

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

__
```

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

```
D:sing<-VERB

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

__
D:have<-VERB

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

__
```

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

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

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

```
YOU SING
vous chantez!

THEY SING
ils chantent!

I HAVE
j'ai!

WE HAVE
nous avons!

THEY HAVE
ils ont!
```

The example of course is extremely limited as translations go. For more substantial processing, we would also take English inflectional stemming into account, use macro substitutions to take care of irregularities on the English side like `has`, and handle other subtleties. We also have to deal with various tenses other than present as well as aspect, mood, and so forth. You should, however, be able to envision now what a full PyElly grammar should look like; it will take much more work to make complete, but would be a straight extension of what we have seen above.

7. Running PyElly From a Command Line

We have so far described how to set up definition text files to create the various tables to guide PyElly operation. This section will show you how to run PyElly for actual language analysis, but first we will have to take care of some preliminary setup. That should be fairly straightforward, but computer novices 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.

Once you have the latest version Python 2.7.* 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 free, but still under copyright.

The Python code making up PyElly currently consists of 64 modules comprising about 16,000 source lines altogether. A beginning PyElly user really needs to be familiar with only three of the modules.

> **`ellyConfiguration.py`** - defines the default environment for PyElly processing. Edit this file to customize PyElly to your own needs. Most of the time, you can leave this module alone.
>
> **`ellyBase.py`** - sets up and manages the processing of individual sentences from standard input. You can run this for testing or make it your programming interface if you want to embed PyElly in a larger application.
>
> **`ellyMain.py`** - runs PyElly from a command line. This is built on top of EllyBase and is set up to extract individual sentences from continuous text in standard input.

The ellyBase module reads in `*.*.elly` language definition files to generate the various tables to guide PyElly analysis of input data. Section 4 introduced three of them. For a given application `A`, these were `A.g.elly`, `A.p.elly`, and `A.m.elly`, with only the `A.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 automatically saved in two files: `A.rules.elly.bin` and `A.vocabulary.elly.bin`. The first is a Python pickled file, which is not really binary since you can look at it with a text editor, but this will be hard for people to read. The second is a binary database file produced by SQLite from definitions in a given `A.v.elly` (see Subsection 9.4 for an explanation).

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 PyElly language rule tables will be reloaded from the `*.elly.bin` files.

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

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

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

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

This will first generate the PyElly tables for the specified application and provide a detailed dump of grammar rules allowing you to see any problems in a language definition. The default application here will be `test` if none is specified. Resulting tables will be saved as `*.elly.bin` files that PyElly can subsequently load directly to start up faster.

After initializing, ellyBase will prompt for one sentence per input line, which it will then translate. Its output will be a rewritten sentence in brackets if translation is successful; or just `????` on failure. It will also show a parse tree of the syntactic analysis done plus a detailed summary of internal details of parsing. The optional `depth` argument above will limit how far down the reporting of parse trees will go (see `-d` for ellyMain below).

For an application with batch processing of input sentences not necessarily on separate lines, you normally will invoke ellyMain from a command line. The `ellyMain.py` file 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,…][ -p][ -noLang] [name] < text
```

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

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

`-b`	operate in batch mode with no prompting; PyElly will otherwise run in interactive mode with prompting when its text input comes from a user terminal.

`-d n`	set the maximum depth for showing a PyElly parse tree to an integer n. This can be helpful when input sentences are quite long, and you do not want to see a full PyElly parse tree. Set n = 0 to disable parse trees completely. See Section 12 for more details.
`-g v0,v1,v2,..`	define the PyElly global variables gp0, gp1, gp2, ... for PyElly semantic procedures with the respective specified string values v0, v1, v2, ...
`-p`	show cognitive semantic plausibility scores along with translated output. If semantic concepts are defined, PyElly will also give the contextual concept of the last disambiguation according to the order of interpretation by generative semantics. This is intended mainly for debugging, but may be of use in some applications (see `disambig`, described in Section 14).
`-noLang`	do not assume that input text will be in English; the main effect is to turn off English inflectional stemming (See Section 11).

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

```
PyElly v1.3.5.2, Natural Language Filtering
Copyright 2014, 2015 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 find the sentence boundaries according to its current rules and divide up the text for analysis.

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. PyElly `*.*.elly` language definition files should be in UTF-8 encoding and may contain arbitrary Unicode except in grammar symbol names. As text input to translate, however, PyElly currently accepts only ASCII and Latin-1 letters plus some additional Unicode punctuation; all other input characters will be automatically converted to spaces. The `chinese` application described in Section 14 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 save to a file 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 can also redirect. Historically, the predecessors of PyElly have been filters, which in Unix terminology means a program that reads from standard input and writes a translation to standard output.

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

```
>> Who gets the gnocchi?

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

where the second line is actual output from ellyMain. PyElly by default converts upper case to lower, and will strip off English inflectional endings as well as -ER and -EST. You can get stricter 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 can change this by editing the value for the symbol baseSource in `ellyConfiguration.py`. The various PyElly applications described in Section 14 are distributed in the `applcn` subdirectory under the main directory of Python source files, resources, and documentation.

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

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

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

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

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

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

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

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

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

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

8. Advanced Capabilities: Grammar

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

You can actually accomplish a great deal with such basics alone, but for more challenging language analysis, PyElly supports other capabilities beyond the confines of pure 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 also normally a topic of grammar, but PyElly breaks this out as a separate level of processing for modularity. The details on this will be discussed in Section 11.

8.1 Syntactic Features

PyElly currently allows for only 64 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 there is a more convenient option here. PyElly also lets you qualify syntactic types through syntactic features, which in effect greatly multiplies the total number of syntactic types available.

Syntactic features are binary tags serving to subcategorize syntactic types; they appeared in Noam Chomsky's seminal work *Syntactic Structures* (1957). 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 really needed, you can get more than sixteen syntactic features by redefining the variable `FMAX` in `symbolTable.py`.

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

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

```
[oF1,F2,F3,...,Fn]
```

where "o" is a single-character identifier for a set of feature names for a specific class of syntactic types and `F1`, ..., `Fn` are the actual names composed of alphanumeric

characters, possibly preceded by a prefix '–' or '+'. For clarity, set identifiers are usually punctuation characters and should never be in the set { '+' , '–' , '*' , '[' , ']' , ',' }.

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

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

A syntax rule with feature names might appear as follows:

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

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

PyElly sets have no upper limit on the number of different sets, but it is probably a good idea to have only five six. Just remember that syntactic features are supposed to simplify grammars, not make them impossibly 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.

The special feature name *LEFT (or equivalently *L) will also be automatically defined. This will work like *RIGHT, except that inheritance will be from the leftmost immediate subconstituent. You can specify both *LEFT and *RIGHT in the features for a syntax type, but usually just one will suffice. With a one-branch rule, *LEFT and *RIGHT will be the same for inheritance, though they will remain distinct as syntactic features.

A third special feature name *UNIQUE will be in all PyElly syntactic feature sets. It has the sole purpose of preventing a phrase from matching any other phrase in PyElly ambiguity checking while parsing. This and the other starred (*) special syntactic features may not be redefined, but they will be counted in the total number of syntactic features available for any given grammar.

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

8.2 The . . . Syntactic Type

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

```
...->
```

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

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

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

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

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

The `. . .` type was intended specifically to support keyword parsing, which tries to recognize a limited number of words in input text and more or less ignores anything

else. A PyElly grammar to support such parsing can always be written without **...**, but may be unwieldy. The `doctor` application for PyElly illustrates how this kind of keyword grammar would be set up; it includes syntax rules like the following:

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

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

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

```
g:sent[@*right]->ss
__
g:x->... key
__
g:key[@ 0,1]->fmly
__
g:ss[.*right]->x[@ 0, 1,-2,-3,-4,-5,-6] ...
_
  append TELL ME MORE ABOUT YOUR FAMILY
__
g:ss[.*right]->x[@ 0, 1,-2,-3,-4,-5,-6] ...
_
  append WHO ELSE IN YOUR FAMILY
__
d:mother <- fmly
__
g:...->unkn
__
g:...->unkn ...
__
```

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

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

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

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

PyElly will also block you from defining a rule like

```
g:...->...
—
```

or like

```
g:X->... ...
—
```

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

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

9. Advanced Capabilities: Vocabulary

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

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

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

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

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

9.1 More on the UNKN Syntactic Type

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

```
D:xxxx <- UNKN

_
  OBTAIN

__
```

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

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

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

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

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

9.2 Breaking Down Unknown Words

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

Consequently, many kinds of rule- and table-driven word stemming exist. It can on the whole be a rather crude instrument for text processing, but might still be helpful for language analysis in general if we can do it reliably. For English at least, it turns out to be fairly straightforward if we can work long enough at the refinement of stemming. This has resulted in two quite separate PyElly tools for analyzing the structure of words as well as dealing with unknown terms.

9.2.1 Inflectional Stemming

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

PyElly inflectional stemming currently recognizes only five endings for English: -S, -ED, -ING, -N, and -T. These each have their own associated stemming logic and also share

additional logic related to recovering the uninflected form of a word. All that logic is based on American English spelling rules and recognizing special cases. PyElly coordinates its execution through the module `inflectionStemmerEN.py`.

If an unknown word ends in -S, -ED, -ING, -N, or -T, PyElly will apply the logic for the ending to see whether it is an inflection and, if so, what the uninflected word should be. Though such logic is necessarily incomplete, it has been refined by forty years of use in various systems and is generally accurate for American spellings of most English words. For example,

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

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

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

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

To use English inflectional stemming in PyElly, setting the `language` variable in the `ellyConfiguration.py` file to `EN`. To override such stemming just for a particular word, define that word in a vocabulary table entry so that it will known in its inflected form. This does not work for `D:` internal dictionary entries.

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

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

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

This approximately translates to

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

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

9.2.2 Morphology

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

The morphology component of PyElly started out as a simple FSA stemmer that served just to remove common endings from English words, including -S, -ED, and -ING. It has now evolved to focus on non-inflectional endings and to output the actual affixes removed as well as the final root form.

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

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

PyElly morphological analysis will be applied only 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 also have to add the grammar rules to recognize prefixes and suffixes and incorporate them into an overall analysis of an input sentence.

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

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

```
abular 2 2 le.
dacy 1 2 te. 1
entry 1 4 .
gual 2 3 . 0a
ilitation 2 6 &,
ion 2 0 .
lenger 2 5 . 0e
oarsen 1 5 .
```

```
piracy 1 4 te. 1
santry 1 4
tention 1 3 d.
uriate 2 2 y.
worship 0 0 .
|carriage 0 0 .
|safer 1 5 . 0e
```

Each line of a `*.stl.elly` file defines a single pattern 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: 0= always reject this match, 1= no conditions, 2= the ending must be preceded by a consonant, and 3= the ending must be preceded by a consonant or U.
- A number specifying how many of the characters of the matched characters to keep as a part of a word after removal of a morphological suffix. A starting vertical bar (`|`) in a listed ending will count as one character here.
- A string specifying what letters to add to a word after removal of a morphological suffix. An `&` in this string is conditional addition of e in English words, applying a method defined in English inflectional stemming.
- A period (`.`) indicates that no further morphological analysis be applied to the result of matching a suffix rule and carrying out the associated actions; a comma (`,`) here means to continue morphological analysis recursively.
- A number indicating how many of the starting characters of the unkept part of a matching ending to drop to get a morphological suffix to be reported in an analysis.
- A string specifying what letters to add to the front of the reduced unkept part of a matching ending in order to make a complete morphological suffix.

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

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

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

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

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

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

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

—

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

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

—

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

The PyElly file `default.stl.elly` is a comprehensive compilation of English word endings evolving over the past fifty years and covering most of the non-foreign irregular forms listed in WordNet exception files. If there is more than one possible analysis, PyElly can make no rule, so that RENT is not reduced to REND. If you actually want to force a decision here, then you must supply your own grammar rule to do it.

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

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

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

```
contra 1 0 .
hydro 1 0 .
non 2 0.
noness 1 3.
pseudo 1 0 .
quasi 1 0 .
```

```
retro 1 0 .
tele 1 0 .
trans 1 0 .
under 1 0 .
```

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

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

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

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.

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

For prefixes here, you will need a dictionary rule like

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

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

9.3 Entity Extraction

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

The main problem with entities is that 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 when they are unsure what a given entity means exactly. This is in fact the purpose of much text that we read: to inform us about something we might be unfamiliar with. A fully competent natural language system must be able to function in this kind of situation.

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

9.3.1 Numbers

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

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

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

  1,000,000 ==> 1000000.

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

  one hundred forty-third ==> 143rd

  fifteen hundred and eight ==> 1508

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

9.3.2 Dates and Times

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

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

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

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

```
import extractionProcedure

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

You can disable date or time extraction by just removing its method name from the extractors list. The second element in each listed entry is a string syntax specification, which generally includes a syntactic type plus syntactic features to assign to a successfully extracted entity. The names of syntactic types and features here will have to be coordinated with other PyElly grammar rules.

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

mm/dd/yyyyXX

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

hh:mm:ssZZZ

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

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

In processing natural language text, we often want to identify the names of persons and of things. This can be handled in various ways within parts of PyElly already seen, but names more generally will present unique problems for both syntactic and semantic analysis. For example, entirely new names or old names with unusual spellings often show up in text, but it is hard to anticipate them in a vocabulary table or even a pattern table. Also, a name can appear in different forms in the same text: Joanne Rowling, J.K. Rowling, Rowling, Ms. Rowling.

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

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

```
X : T

=PPPP
```

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

9.3.3.1 Explicit Name Component Patterns and Types

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

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

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

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

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

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

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

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

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

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

9.3.3.2 Implicit Name Components

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

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

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

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

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

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

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

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

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

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

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

```
=aSN
=KRPLT
```

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

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

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

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

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

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

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

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

9.3.4 Defining Your Own Entity Extractors

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

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

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

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

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

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

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

Each vocabulary entry in a PyElly `*.v.elly` definition file must be a single text line taking 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"

ribulose bisphosphate carboxylase oxygenase

The '`:`' is required to let PyElly know when a term ends; no wildcards are allowed in a `TERM`, and it must start with a letter, digit, or the character '`.`' or '`"`'. Upper and lower case here will not matter, since PyElly will ignore this when matching up entries with input text.

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

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

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

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

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

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 elsewhere in a `*.g.elly` grammar rule file.

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

```
Lady Gaga : noun [^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 a vocabulary table as an encoded numerical form according to a symbol table associated with a PyElly grammar.

Syntactic features must be immediately after a syntactic category name with no space in between. Otherwise, PyElly will be unable to differentiate between syntactic and semantic features in a `*.v.elly` file. Individual syntactic or semantic features inside of brackets may be preceded by a space, however.

Unlike the dictionary definitions of words in a grammar, there are no permanent rules associated with the terms in a vocabulary table. When a term is found by lookup, PyElly automatically generates a temporary internal dictionary rule to define the term. This rule will persist 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 tried out in possible PyElly analyses, with the most plausible taken for a PyElly analysis.

Often, a vocabulary table may have overlapping entries like

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

PyElly will always take the longest matching entry in the analysis of an input sentence and ignore any shorter matches.

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

An external vocabulary entry can contain arbitrary characters that PyElly normally would take to mean that a word or other token has ended and that another has started. For example, the vocabulary entry "R&B" can be matched as a whole in text even though

PyElly would analyze "R&B" in text as separate "R", "&", and "B" without the vocabulary entry. A vocabulary entry may not match across sentences, however.

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, which includes most of the nouns, verbs, adjectives, and adverbs in WordNet 3.0. Always define `A.v.elly` if you do not want such a huge vocabulary; it may take a long time for PyElly to load on a slower computer. PyElly will always save a compiled vocabulary data base for an application `A` in the file `A.vocabulary.elly.bin`. If you change `A.v.elly`, PyElly will automatically recompile any `A.vocabulary.elly.bin` at startup. Recompilation will also happen if `A.g.elly` has changed. Otherwise, PyElly will just read in the last saved `A.vocabulary.elly.bin`.

Note that the `A.vocabulary.elly.bin` file created by PyElly must always be paired only with the `A.rules.elly.bin` file it was created with. This is because syntactic types and features are encoded as numbers in `*.elly.bin` files, which may be inconsistent when they are created at different times. If you want to reuse language rules, always start from the `*.*.elly` files. If PyElly has to recompile `A.rules.elly.bin` at startup, then it will also automatically recompile `A.vocabulary.elly.bin`.

10. Logic for PyElly Cognitive Semantics

The generative semantic part of a grammar rule tells PyElly how to translate its input into output, while the cognitive semantic parts of different grammar rules help in evaluating the plausibility of analyses. Generative semantics is always involved in producing final PyElly output; cognitive semantics is important only when PyElly has to choose between alternate analyses of the same input, but it is always run for each new phrase node created in a PyElly analysis to get a plausibility score anyway.

With large grammars, we cannot expect that every input sentence will always break down in only one way into subconstituents according to the rules of that grammar. In most languages, for example, a particular word occurrence could be assigned multiple parts of speech, and each possibility here can result in various syntactic analyses for an input sentence. Those alternate analyses can potentially lead to conflicting interpretations of a sentence that eventually have to be resolved somehow.

PyElly tries to take a wait-and-see approach in resolving such ambiguous situations. Multiple interpretations might exist at lower levels of a parse tree, but some could end up not fitting into any final analysis for an entire sentence. In this way, an ambiguity might resolve itself when placed in a bigger context. So PyElly tries to hold off on any resolution decision until it has more information.

PyElly will therefore disregard differing interpretations until it comes across two or more phrase nodes of the same syntactic type with the same syntactic features over the same segment of an input sentence. Only at that point will PyElly compare the cognitive semantic plausibility scores already computed for each alternate phrase node and then keep just one of them in continuing a full sentence analysis.

There is only one exception to this delayed resolution. When finished with processing all the tokens in a sentence, PyElly may find two or more phrase nodes of type `SENT` over the entire sentence, but with different syntactic features. It will then choose one interpretation situation according to semantic plausibility regardless of syntactic features because we cannot wait any further.

Some actual kinds of ambiguity can slip completely past PyElly. For instance, "I love rock" could be about music or landscaping. If the grammar rules of a language definition fail to produce two different syntactic analyses with co-extensive phrase nodes of the same type and same features, however, PyElly will be unaware of any alternative interpretations. You have to tell PyElly explicitly where ambiguities are possible.

Consider the following simple set of grammar rules:

```
g:sent[:*r]->x
__
g:sent[:*r]->y
__
d:wwww<-x[:f]
_
  (xgen)
__
d:wwww<-y[:g]
_
  (ygen)
__
```

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

We have two possible sentence analyses here, given the rules for inheriting syntactic features through the predefined `*R` syntactic feature described in Subsection 8.1:

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

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

The method of choosing between alternate sentence analyses or between different interpretations of individual phrases will be through a numerical plausibility score assigned to each phrase node in a parse tree. A score of 0 here will be neutral, increasingly positive will be more plausible, and increasingly negative will be more implausible. Various characteristics of a phrase node will cause its plausibility score to be incremented or decremented.

This section of the PyElly User's Manual tells how to adjust plausibility scores according to the grammar rules employed in an analysis and explains how this all can be employed to assign a score to each possible subtree in an overall PyElly analysis. You can choose to write language descriptions without such explicit scoring, but it is built into the PyElly parsing algorithm, and provides another way to get more control over how PyElly runs.

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

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

For example, suppose we have a constituent described by a grammar rule `A->X Y`. We first get a plausibility score for subconstituent `X` and another score for subconstituent `Y`; with PyElly parsing, these would be computed already. Then we run the cognitive semantic logic for the grammar rule `A->X Y`, producing an adjustment to the summed plausibility scores for `X` and `Y` to get an overall plausibility for our phrase of type `A`.

With competing analyses, if only one has the top score, PyElly chooses it and is done. If more than one has the highest score, then PyElly will arbitrarily pick one of them. PyElly will keep track of which grammar rules are involved here, however, and will make a different choice the next time an ambiguity arises with these rules.

10.1 The Form of Cognitive Semantic Clauses

PyElly currently provides three different ways to define how a grammar rule will contribute to a plausibility score: by a fixed value assigned to the grammar rule in a language definition, by logical rules relating the semantic features associated with the constituents to be combined by the grammar rule, and by measuring the semantic distance between the concepts associated with each constituent.

In the input `*.g.elly` file defining grammar rules for a PyElly language description, the cognitive semantic logic will consist of a series of single-line clauses coming after the `G:` or a `D:` line introducing each rule and ending at the first _ or __ line (see Section 4). Each clause will be a line containing the character sequence '>>', possibly with text before and after. For example, the following rule with no explicit generative semantics has three cognitive semantic clauses:

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

__
```

A '>>' divides a clause into two major parts. The left part specifies conditions on the immediate subconstituents of a phrase structure in order for the clause to apply. The right part specifies actions to take if the left side is satisfied. The '>>' is mandatory in any cognitive semantic clause. Spaces between the conditions of the left part of a clause are optional.

When evaluating the plausibility of a given rule of grammar, each of its cognitive semantic clauses will be tried in order until one is found to apply. None may apply, in which case a zero plausibility contribution is assumed. Having no conditions on the left side of a clause is equivalent to an always-true condition, and such a clause will always make any following clauses irrelevant.

The actual form of a clause will depend on which of the three kinds of plausibility contribution is being specified in the clause. You may freely mix the different kinds within the same clause or the same set of clauses, but remember that ordering does matter here; the first clause to match will always define the contribution of a grammar rule for an overall plausibility score.

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

```
?>>?
```

The `?` condition on the left side always evaluates to False, which means that right side of the clause will never be run. This will have the side-effect, however, of setting a flag that will turn on print statements when running all following cognitive semantic clauses for a given grammar rule. For example,

```
tracing phrase 0 : rule= 29 with current bias=0
cog sem at clause 4 of 4
l: phrase 1 @0: type=0 syn[00 00] sem[00 00] : bia=0 use=0
r: phrase 2 @1: type=0 syn[00 00] sem[00 00] : bia=0 use=0
incremental scoring= 1 sem[00 00]
```

This identifies the phrase node where the cognitive semantic logic is being executed, the grammar rule generating the node, the clause in effect for scoring adjustment, and the subconstituents involved. The incremental scoring is then reported along with the actual setting of semantic features for the phrase node.

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

10.2 Cognitive Semantic Approaches

PyElly cognitive semantics shows up mainly in the grammar rule logic for evaluating the plausibility of phrase nodes in an analysis, but appears in various other places as well. There are three different approaches possible: fixed scoring, semantic features, and semantic concepts.

10.2.1 Fixed Scoring

The simplest and most common kind of cognitive semantic clause will assign a fixed positive or negative score unconditionally to a grammar rule in order to favor or disfavor a phrase analysis based on the rule. Such clauses may take one of the following forms:

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

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

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

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

```
>>+0
```

a special case of fixed scoring. Note that the "+" is necessary here if you actually want to be explicit here about a zero score.

10.2.2 Semantic Features

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

```
[&ANIMATE,MOBILE]
```

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

As with syntactic features, you may have up to 16 semantic feature names, with the rules for legal names being the same. Unlike syntactic features, however, they will have only one predefined feature name: `*CAPITAL` or equivalently `*C`, which indicates that a phrase is capitalized and so is probably a name or a proper noun. This name is reserved in every semantic feature set.

10.2.2.1 Semantic Features in Cognitive Semantic Clauses

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

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

The symbol "`o`" is a feature set identifier; "`x`" may be either `*L` or `*R`, and the "`#`" is a fixed scoring as in Subsection 10.1 above; for example, `+++` or `-3`.

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

The prefixes `L` and `R` on the left part of a clause specify the constituent substructures to be tested, respectively left and right descendant. Either can be omitted, but you probably want to specify at least one, if you want something different from fixed scoring.

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

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

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

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

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

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

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

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

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

10.2.2.2 Semantic Features in Generative Semantics

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

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

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

10.2.3 Semantic Concepts

PyElly is a currently being used experimentally in applying conceptual information from WordNet to infer the intended sense of ambiguous words in English text. This now a cognitive semantic approach in PyElly.

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

(WordNet is produced manually by professional lexicographers affiliated with the Cognitive Science Laboratory at Princeton University and is an evolving linguistic resource now at version 3.1. [George A. Miller (1995). WordNet: A Lexical Database for English. Communications of the ACM Vol. 38, No. 11: 39-41.] This notice is required by the WordNet license.)

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

For disambiguation experiments in PyElly, trying to work with all the synsets of WordNet 3.1 is too cumbersome. So we instead have been focusing on concepts from a small subset of WordNet synsets related to interesting kinds of ambiguity in English. We can identify each such concept as an 8-digit decimal string combining the unique

WordNet offset for its corresponding synset plus a single appended letter to indicate its part of speech. For example,

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

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

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

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

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

10.2.3.1 Concepts in Cognitive Semantic Logic

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

10.2.3.1.1 Concepts in the Left Side of Cognitive Semantic Clauses

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

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

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

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

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

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

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

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

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

10.2.3.1.2 Concepts in the Right Side of Cognitive Semantic Clauses

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

```
>>++ CONCEPT
```

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

```
>>+0 CONCEPT
```

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

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

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

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

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

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

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

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

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

10.2.3.2 Semantic Concepts in Language Definition Files

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

10.2.3.2.1 Conceptual Hierarchy Definition (A.h.elly)

This specifies all the concepts in a language definition and their semantic relationships. You can define everything arbitrarily, but to ensure consistency, start from some existing language database like WordNet. Here are some entries from `disambig.h.elly`, a conceptual hierarchy definition file based on WordNet 3.1 concepts for a PyElly example application:

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

The ">" separates two concept names to be interpreted as a link in a conceptual hierarchy, where the left concept is the parent and the right concept is a child. In this particular definition file, each concept name is an offset in a WordNet 3.1 part of speech

data file plus a single letter indicating which part of speech (`n`, `v`, `a`, `r`). Both offset and part of speech are necessary to identify any WordNet concept uniquely.

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

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

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

10.2.3.2.2 Semantic Concepts in Grammar Rules

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

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

_
  APPEND xxxx-C

__
```

Similarly with a regular syntax rule:

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

_
  RIGHT
  SPACE
  LEFT

__
```

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

10.2.3.2.3 Semantic Concepts in Vocabulary Table Entries

For a vocabulary table entry, we extend the plausibility field in an `A.h.elly` input file to allow appending a concept name separated by a "`/`" (See Subsection 9.4 above). Omitting a concept name here will be equivalent to a null concept.

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

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

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

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

11. Sentences and Punctuation

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

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

This sentence contains six periods (`.`), but only the final one stops the sentence. It is not hard to recognize such exceptions, but this is yet more detail to take care of on top on already complex tasks of language analysis.

PyElly divides text into sentences with its ellySentenceReader module, which employs a simple sequential punctuation-checking algorithm to detect sentence boundaries in text. While doing this, PyElly also normalizes each sentence to make subsequent processing easier. The algorithm is simple and tends to find too many sentences, but we can help it out by providing some supporting modules with more smarts.

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

The approach of PyElly here is to provide sentence recognition a notch or two better than what one can cobble together just using Python regular expressions or the standard sentence recognition methods provided by libraries in a language like Java. If you really need more than this, then there are other resources available; for example, NLTK can be trained on sample data to discover its own stop exceptions. Builtin PyElly sentence recognition should be adequate for most of its potential 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 recognized by PyElly, and replaces single new line characters with spaces. The ellyCharInputStream and ellySentenceReader modules both operate at the character level and together will divide input text into individual sentences for PyElly processing.

A single input line could contain multiple sentences, or a single sentence may extend across multiple input lines. There is also no limit on how long an input line may be; it could be an entire paragraph terminated by a final linefeed as found in many word processing files. PyElly can also read text divided into short lines terminated by

linefeeds, carriage returns, or carriage returns plus linefeeds. It currently will not splice back a hyphenated word split across two lines, however.

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

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

11.1 Basic PyElly Punctuation

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

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

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

All predefined PyElly single- and multi-character punctuation will be associated with the syntactic type `PUNC`. If you want to make your own system of punctuation, define your own syntactic types here and use those in your grammar and vocabulary rules. You can even reuse `PUNC`, but remember that this will come with predefinitions.

PyElly also will qualify the syntactic type `PUNC` with syntactic features under the specific ID `[| ]` and semantic features under the specific ID `[! ]`. You should not use these feature names in your own grammar and vocabulary rules unless you really understand the trouble you might get into; the names currently are as follows:

syntactic feature	Indication
START	can start a sentence
STOP	can stop a sentence
***L**	is a left bracket or quotation mark (special use of predefined feature name)
***R**	is a right bracket or quotation mark (special use of predefined feature name)
QUO	is quotation mark
COM	is comma
EMB	can be included in a bracketed expression

semantic feature	Indication
BRK	can divide a sentence without ending it

Your language definition files should expect these builtin definitions. In particular, if you plan on defining your own syntactic features under the ID `[| ]`, then you will have to make their names different, and there will be fewer free feature slots available.

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

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

You can also use PyElly macro substitution to change the form of punctuation before it is looked up. This is a good way of handling ellipsis; transform three periods with or without intervening spaces into a single character: . . . → … .

11.2 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.

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

When PyElly starts up an application `A`, its stopException module will try to read in a file called `A.sx.elly`, or failing that, `default.sx.elly`. This file specifies various patterns for when a text character should not treated as normal sentence termination.

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 actual exception patterns from `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 list.

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 patterns where it makes a difference what character precedes the match. In the case of `DR.`, the preceding character probably does not matter; but in the case of `@.`, it will. This is why the pattern here needs to be `~@.`.

11.2.2 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. This is not hard, though.

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

11.3 How Punctuation Affects Parsing

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

For example, you might want to put `DR.` into your vocabulary table, perhaps as the syntactic type `TITLE`. Since this will take three characters from an input stream, including the period, PyElly will no longer see the punctuation here. PyElly tokenization will always take longest possible match when multiple PyElly rules can apply; a token including punctuation will probably be longer than anything else.

Identifying punctuation in an input sentence is just the start of PyElly analysis, however. The grammar rules for a PyElly application will then have to describe how to fit the punctuation into the overall analysis of a sentence and how eventually to translate it. This will be entirely your responsibility; and it can get complicated.

In simple text processing applications taking only a sentence at a time, 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, you can have a grammar rule like

```
g:UNKN->PUNC
—
```

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

When you have to work with sentence punctuation, you will need at least one grammar rule like

```
g:SENT->SENT PUNC[|STOP]
—
```

for handling stop punctuation terminating a sentence, although the syntactic feature reference is often unnecessary. The setting of syntactic features by the PyElly punctuationRecognizer module can be ignored when no grammar or vocabulary rules refer to them.

PyElly parsing will fail if any part of a sentence cannot be put into a single coherent syntactic and semantic analysis; and punctuation handling will be a highly probable point of failure here.

12. PyElly Parsing and Tokenization

Parsing and tokenization are usually invisible in PyElly operation, which should help to simplify the development of natural language applications. Still, we do sometimes need to look under the hood, either when something goes wrong or when efficiency becomes an issue. So this section will take a deep dive into how PyElly analyzes a sentence, a procedure that has evolved to become rather complex.

PyElly follows the approach of compiler-compilers like YACC. Compilers are the indispensable programs that translate code written in a high-level programming language like Java or C++ into the low-level machine instructions that a computer can execute directly. In the early days of computing, all compilers were written from scratch; and the crafting of individual compilers was complicated and slow. The results were often unreliable.

To streamline and rationalize compiler development for a proliferation of new languages and new target machines, compiler-compilers were invented. These provided prefabricated and pretested components that could be quickly customized and bolted together to make new compilers. Such standard components typically included a lexical analyzer based on a finite-state automaton and a parser of languages describable by a context-free grammar.

Using a compiler-compiler of course limits the options of programming language designers. They have to be willing to work with the constraint of context-free languages; and the individual tokens in that language (variables, constants, and so forth) have to be recognizable by a simple finite-state automaton. Such restrictions are significant, but being able to develop a reliable compiler in weeks instead of months is so advantageous that almost everyone can accept the tradeoffs.

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

12.1 A Bottom-Up Framework

A parser analyzes an input sentence and builds a description of its structure. As noted earlier, this structure can be represented as a kind of tree, where the root of the tree is a phrase node of the syntactic type `SENT` and the branching of the tree shows how complex structures break down into simpler structures. A tool like PyElly must be able to build such trees incrementally for a sentence, starting either at the bottom with the basic tokens from the sentence or at the top by putting together different possible structures with `SENT` as root and then matching them up with the parts of the sentence.

One can debate whether bottom up or top down is better, but both should produce the same parse tree in the end. We can 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 show up in a top-down analysis. PyElly does this through a true/false matrix m(`X`,`Y`) telling whether a syntactic type `Y` could eventually satisfy a goal of `X` at parse position; it is automatically compiled when loading a grammar into PyElly

LINGOL and subsequently PyElly both take this restricted bottom-up approach. Doing so is quite efficient, and the various resulting subtrees can provide helpful information when parsing fails or when a translation turns out wrong. It is also more convenient for computing plausibility scores with PyElly cognitive semantics.

The PyElly bottom-up algorithm revolves around a queue that lists the newly created phrase nodes of a parse tree. These still need to be processed to create the phrase nodes at the next higher levels of our tree. Initially, the queue is empty, but we then read the next token in an input sentence and look it up to get some new bottom-level parse tree nodes to prime our queue.

PyElly parsing then runs in a loop, taking the node at the front of its queue and applying its grammar rules to create new nodes to be appended to the back of the queue for further action. This procedure keeps going until the queue finally empties out. At that point, PyElly will then try to read the next token from a sentence to refill the queue and proceed as before. Parsing will stop after every token in sentence has been seen.

There is one circumstance when a new node will not be added to the end of a queue. If there was already a phrase node of the same syntactic type with the same syntactic features built up from the same sentence tokens and if the new node does not have the `*UNIQUE` syntactic feature, PyElly will note an ambiguity here and will attach the new node as an alternative to the already processed node instead of queueing it separately for further tree building.

This consolidation of new ambiguous nodes serves to reduce the total number of nodes generated for the parsing of a single sentence. Otherwise, PyElly would have to build parallel tree structures for both the old node and the new node without necessarily any benefit. The `*UNIQUE` syntactic feature does allow you to override the handling of ambiguities here if you really want to do so.

In any event, PyElly immediately computes the plausibility score of each new phrase as it is generated in parse tree building. Whenever an ambiguity is found, PyElly will find the alternative with the highest plausibility and use it in all later processing of a sentence. All the other alternatives will, however, be retained for reporting, for possible backup on a semantic failure, or for adjusting biases to insure that the same rule will not always be taken when there are multiple rules with the same semantic plausibility.

12.2 Token Extraction and Lookup

PyElly token lookup is complicated because it happens in many different ways: external vocabulary tables, FSA pattern rules, entity extraction, the internal dictionary rules for a grammar, and builtin rules like those for punctuation recognition. These possibilities must also interact with macro substitution, inflectional stemming, and morphological analysis; and so it can be hard to understand what is going on here.

PyElly sees a sentence as a sequence of tokens, each a single word or word fragment, a number, a phrase, a complex entity, or punctuation. PyElly parsing goes from left to right in a sentence, applying various language rules to get the extent of the token at the next position in a sentence. For an application `A`, the full lookup procedure is currently as follows:

1. If number rewriting is enabled, rewrite any spelled-out number in the current sentence position as digits plus any ordinal suffix like -ST, -RD, or -TH.
2. Apply macro substitution rules at the current position.
3. Look up the next input text in the external vocabulary table for `A`; put matches into the PyElly parsing queue as parse tree leaf phrase nodes if consistent with top-down parsing expectations at the current position according to the current grammar rules for `A`.
4. Try also to match up the next input with the FSA pattern table for `A`; queue up matches as leaf phrase nodes at the current position if they are consistent with top-down parsing expectations.
5. Try entity extraction at the current position; put matches as phrase nodes into the PyElly parsing queue if consistent with top-down expectations at the current position.
6. If steps 2, 3, or 4 have queued phrase nodes, keep only the phrases with the longest extent; discard all phrase nodes of shorter extent for subsequent processing.
7. If any queued phrases are longer that the next simple input token, we are done with the generation of leaf phrase nodes and ready to start the main parsing loop.
8. Otherwise, extract the next input token from the PyElly input buffer with inflectional stemming and macro substitution.
9. Look up the input token as a single word in both external vocabulary table and the internal dictionary for `A`. Queue up a phrase node for any matches here if consistent with top-down expectations.
10. If there are any queued phrases, we are done and ready to go into the PyElly main parsing loop.

11. Otherwise, morphologically analyze our current single-word token with the rules defined for `A`. Look it up in the external vocabulary table and in the internal dictionary for `A`. If found and consistent with top-down expectations, queue up phrase nodes for each match.

12. If there are any queued phrase nodes, we are done.

13. Otherwise, check if the next token is standard punctuation. If so and the punctuation is consistent with top-down expectations, enqueue a phrase node of syntactic type `PUNC` and stop.

14. If all else fails, then create a phrase node of `UNKN` type for the shortest next token. This will be without any top-down consistency check.

The lookup process will produce a queue of at least one phrase node for the next token for a round of parse tree building. This will continue until input is exhausted or when no more phrase nodes can be generated.

12.3 Building a Parse Tree

Given a way to put bottom-level 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's main loop, and the two following subsections will describe PyElly extensions to that algorithm.

12.3.1 Context-Free Analysis Main Loop

At each step in parsing, we first enqueue the lowest-level phrase nodes for the next piece of an input sentence, with any ambiguities already identified and resolved. Then for each queued phrase node, we go through a process of determining all the ways that the node will fit into a parse tree currently being built. This is called "ramification" in PyElly source code commentary.

For newly enqueued phrase node, PyElly ramification will go through three steps when the syntactic type of the node is `X`:

1. Look for rules of the form `Z->Y X` that have earlier found a `Y` and set a goal of an `X` in the current position. For each such goal found, create a new phrase node of type `Z`, which will be at the same starting position as phrase `Y`.

2. Look for rules of the form `Z->X`. For each such rule, create a new node of type `Z` at the same starting position and with the same extent in a sentence as `X`.

3. Look for rules of the form `Z->X Y`. For each such rule, set a goal at the next position to look for a `Y` to make a `Z` at the same starting position as `X`.

A new phrase node will be vetoed in steps 1 and 2 if inconsistent with a top-down algorithm. This uses the same true/false derivability matrix employed in token lookup.

Each newly created node will be queued up for processing by taking the three steps above. When all the phrase nodes ending at the current sentence position have been ramified, PyElly parsing advances to the next position.

The main difference between PyElly basic parsing here and similar bottom-up context-free parsing elsewhere is in the handling of ambiguities. Artificial languages generally avoid any ambiguities in their grammar, but natural languages are full of them and so we have to be ready here. In PyElly, the solution is to resolve ambiguities outside of its ramification steps.

PyElly sees an ambiguity only when two phrase nodes of different types or features 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 have to be resolved at some point, but if they are marked with different syntactic categories or have different syntactic features, PyElly will have to wait.

It is possible that a parsing ambiguity may found for a phrase node after it has been ramified. This is no problem if that node has a higher plausibility than the new phrase node producing the ambiguity; but if the new phrase is more semantically plausibility, then it must replace the old phrase, and the plausibility scores of all of its ramifications must also be adjusted upward to reflect the replacement. Such changes of the plausibility of other phrase nodes may in turn require adjustment of their ramifications. PyElly handles all of this automatically.

12.3.2 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 the introduction of syntactic features as an extra condition on whether or not a rule is applicable for some aspect of ramification.

On the right side of a rule like `Z->X` or `Z->X Y`, you can specify what syntactic features must and must not be turned on for a queued phrase node of syntactic type `X` to be matched in steps 2 and 3 above and for a queued phrase node of type `Y` to satisfy a goal based on a rule `Z->X Y` in step 1. This 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 first of the resulting phrase nodes so that only step 3 of ramification will be taken for the root part and only step 1 will be taken for each inflection part. A parse tree will therefore grow more slowly than otherwise expected, making for faster parsing and less overflow.

12.3.3 Type 0 Grammar Extensions

The introduction of the PyElly `...` syntactic type complicates parsing, but handling the type 0 grammar rules currently allowed by PyElly turns out to require only two localized changes to its core context-free algorithm.

1. Just before processing a new token at the next position of an input sentence, generate a new phrase node for the grammar rule `...[.1]->` . Enqueue the node and get its ramifications immediately.
2. Just after processing the last token of an input sentence, generate a new phrase node for the grammar rule `...[.2]->` . Enqueue it and get its ramifications immediately.

Those reading this manual closely will note that the two rules here have syntactic features associated with `...`, which Section 8 said was not allowed. That restriction is still true, and that is because PyElly reserves the syntactic features of `...` to make the type 0 logic handling work properly as done above.

The difficulty here is that the `...` syntactic type is prone to producing ambiguities. This will be especially bad if the PyElly parser cannot distinguish between a `...` phrase that is empty and one that includes actual pieces of a sentence. So PyElly itself keeps track by using syntactic features here, but keeps that information invisible to users.

The solution will propagate up the syntactic feature `[.1]` to indicate an empty phrase due to case 1 and the feature `[.2]` to indicate an empty phrase due to case 2. Though invisible, a grammar will still need to guide this explicitly through setting `*LEFT` or `*RIGHT` in rules for syntactic feature inheritance when a rule involves `....`

12.4 Success and Failure in Parsing

For any application, PyElly automatically defines the grammar rule:

```
g:SENT->SENT END
```

—

This rule will never be realized in an actual phrase node, but the basic PyElly parsing algorithm uses this rule to set up a goal for the syntactic type `END` in phase 3 of ramification. After a sentence has been fully parsed, PyElly will look for an `END` goal at the position after the last token extracted from the sentence. If no such goal is found, then we know that parsing has failed; otherwise, we can then run the generative semantics for the `SENT` phrase node that generated the `END` goal just found.

There may be more than one `END` goal in the final position, indicating that their respective generating `SENT` phrase nodes were not collapsed as an ambiguity. PyElly can still compare their cognitive semantic plausibility scores, select the most plausible, and run its generative semantic procedure to get the interpretation for a sentence. This is equivalent to making actual phrase nodes based on a `SENT->SENT END` rule, which will trigger PyElly ambiguity handling as just described.

Failure in parsing gives us no generative semantic procedure to run, and our only recourse then is to dump out intermediate results and hope that someone can spot some helpful clue in the fragments. If the failure is due to something happening in semantic interpretation, though, PyElly can automatically try to recover by backing up in a parse tree to look for an ambiguity and selecting a different alternative at that point.

12.5 Parse Data Dumps and Tree Diagrams

PyElly can produce dumps of parsing data, including all the complete or partial parse trees built up for a sentence. In a successful analysis, this helps in verifying that PyElly is running as expected. In a failed analysis, the parse trees will provide clues about what went wrong. For example, you can see where the building of a parse tree had to stop and whether this was due to a missing rule or bad input text.

All this information is written to the standard error stream. Such output originally was an informal debugging aid, but has proved so useful that it is now integral to PyElly operation. The most important part of parse data dumps are the trees. These will be presented horizontally, with their highest nodes on the left and with branching laid out vertically. For example, here is a simple 3-level subtree with 4 phrase nodes:

```
sent:0000──ss:8001┬noun:8000 @0 [nnnn]
   6 =  3    4 =  2│   1 =  1
                   └verb:0000 @1 [vvvv]
                       2 = -1
```

Each phrase node in a tree display will have the form

```
type:hhhh
   n =  p
```

Where `type` is the name of a syntactic type truncated to 4 characters, `hhhh` is hexadecimal for the associated feature bits (16 are assumed), `n` is phrase sequence number indicating the order in which it was generated, and `p` is the numerical plausibility score computed for the node. The nodes are connected by Unicode drawing characters to show the kind of branching in grammar structures.

To interpret the feature bits `hhhh` here, you should look at the encoding of feature names produced by grammarTable.py. The feature encoded as 0 will be the leftmost bit and will show up as the hexadecimal `8000`; the feature encoded as 1 will be `4000`.

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

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

This node above has no syntactic features turned on; its node sequence ID number is 6, and its plausibility score is +3. Similarly, the node for type `ss` at the next level is

```
 ss:8001
  4 =  2
```

The actual sentence tokens for a PyElly will be in brackets on the far right, preceded by its sentence position, which starts from 0. In the example above, the tokens are the "words" `nnnn` and `vvvv` in sentence positions 0 and 1, respectively. Every parse tree branch will end on the far right with a position and token, plus a semantic concept if your grammar includes them.

With analysis of words into components becoming separate tokens, 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 elements HELLO and period (.), as you would expect. Note that the hidden syntactic flags of the `. . .` type do show up in the displayed tree here; just ignore them.

All the examples here show complete sentences that might be chosen for semantic interpretation. When a PyElly analysis fails, however, you may want to see all the results of parsing, including rejected ambiguities and dead ends. PyElly will do this dump according to the sequence numbers of phrase nodes, which indicate their order of generation.

For a full dump, PyElly looks for the node with the highest sequence number not yet shown in any parse tree for the current dump. PyElly then dumps the subtree under that node as done above for the subtree under `sent` and loops back in this fashion until all phrase nodes are accounted for. For a long sentence, we often will have tens of thousands of phrase nodes, but each node will show up in only one subtree.

In addition to all the trees and subtrees, a PyElly full dump will also show the goals at the final position in a sentence analysis, all grammar and internal dictionary rules applied plus the phrases nodes generated, and all ambiguities found in the process. This information should allow you to reconstruct how PyElly parsed a sentence.

For example, suppose that we have the following trivial grammar, which allows for sentences consisting of either a `NOUN` plus a `VERB` or a `VERB` plus a `NOUN`:

```
# trivial grammar
p:do

_
  left
  space
  right

__
g:sent->noun verb

_
  (do)

__
g:sent->verb noun
 >>-

_
  (do)

__
g:noun->noun sufx

_
  (do)

__
g:verb->verb sufx

_
  (do)

__
d:dog  <- noun

__
d:dog  <- verb

__
d:-s   <- sufx

__
d:bark <- verb

__
d:bark <- noun

__
```

Here is an example of a full dump for an analysis of a short sentence with ellyBase:

```
> dogs bark.

parse FAILed!
dump all

dumping from phrase 8 @0: typ=0 syn[00 00] sem[00 00] : bia=-1 use=0
 sent:0000┬verb:0000┬verb:0000 @0 [dog]
    8 = -1│   4 =  0│   1 =  0
          │         └sufx:0000 @1 [-s]
          │             2 =  0
          └noun:0000 @2 [bark]
              6 =  0

dumping from phrase 7 @0: typ=0 syn[00 00] sem[00 00] : bia=0 use=0
 sent:0000┬noun:0000┬noun:0000 @0 [dog]
    7 =  0│   3 =  0│   0 =  0
          │         └sufx:0000 @1 [-s]
          │             2 =  0
          └verb:0000 @2 [bark]
              5 =  0

rules invoked and associated phrases
rule    2: 7
rule    3: 8
rule    4: 3
rule    5: 4
rule    6: 0
rule    7: 1
rule    8: 2
rule    9: 5
rule   10: 6

3 final goals at position= 3
 goal 8: sufx typ=6 for [phrase 5 @2: typ=5 syn[00 00] sem[00 00] : bia=0 use=0] rul=5
 goal 9: sufx typ=6 for [phrase 6 @2: typ=4 syn[00 00] sem[00 00] : bia=0 use=0] rul=4
 goal 10: end typ=1 for [phrase 7 @0: typ=0 syn[00 00] sem[00 00] : bia=0 use=0] rul=1

9 phrases altogether

ambiguities
sent  0000:  7 (+0/0)  8 (-1/0)

4 raw tokens= [[dog]] [[-s]] [[bark]] [[.]]
9 phrases, 11 goals
```

Parsing fails here for the input "dogs bark." because our simple grammar expects no punctuation at the end of a sentence. In the full dump, we first see the two subtrees for the first three tokens of the input sentence. The last position for which goals are defined here is at 3, and these are listed. In the listing of ambiguities we have phrases 7 and 8, both identified as a `SENT` type without no syntactic features being set; phrase 7 is at the front of the listing because it has the higher semantic plausibility score as result of the cognitive semantics for rule 3.

If a semantic concept is defined for a phrase at a leaf node in a parse tree, a PyElly tree dump will show the concept immediately after the bracketed token at the end of an output line. For example, the augmented printout for tree the first tree from the example above might become

```
sent:0000┬noun:0000┬noun:0000 @0 [dog] 02086723N
   4 =  0│    2 =  0│    0 =  0
         │          └sufx:0000 @1 [-s]
         │              1 =  0
         └verb:0000 @2 [bark] 01049617V
             3 =  0
```

where `02086723N` and `011049617V` are WordNet-derived concept names as described in Subsection 10.4.1 above. If no concept is defined for a leaf node, then the tree output will remain the same as before. This is the case for the suffix -S here.

12.6 Parsing Resource Limits

PyElly is written in Python, a scripting language that can be interpreted on the fly. In this respect, it is closer to the original LINGOL system written in LISP than to any of its 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 main memory, PyElly should be able to handle sentences containing over a hundred tokens with little difficulty. Writing grammar tules to describe huge sentences efficiently may take nontrivial effort, however, requiring extra care in controlling all the many combinations of possible ambiguous interpretations here.

When grammar rules allow for high degrees of ambiguity, the total number of phrase nodes allocated in generating a complete parse for a sentence can grow exponentially with the number of distinct tokens in the input. This can result in noticeably slow processing or even apparent crashes. Be careful especially when using the predefined `*unique` syntactic feature or when many words have to be typed syntactically as `UNKN`.

In the worst case, PyElly will generate all possible parse trees for a sentence. These are reduced somewhat by immediate resolution of ambiguities of the same syntactic type and with the same syntactic features, but sentences with many competing ambiguous interpretations will clog up PyElly processing. Inflectional stemming or morphological analysis will of course produce extra tokens beyond the count of words and punctuation for a sentence.

The PyElly `ellyConfiguration` module as of v1.2.2 imposes a 50,000 node limit on the total number of phrases generated for a sentence parse tree. This is quite generous, for you will see a big parsing slowdown when there are only 10,000 phrase nodes. When a PyElly sentence analysis hits the cutoff, PyElly will report an overflow and break off processing of the sentence; it will produce no translation and show no parse trees.

If you really must, you can raise the phrase node limit by editing the `phraseLimit` parameter definition in the file `ellyConfiguration.py`. This probably will not help much, however. Overflow is a warning that your grammar and vocabulary really need to be rethought and tightened up. A better approach here might be to use syntactic features

to restrict the applicability of syntax rules and to define more terms explicitly, especially ones composed of multiple words.

As of v1.3.5, you can also use special control characters to limit the number of phrase nodes generated for a PyElly parse tree. Such a control character in PyElly input will have no output translation; it serves only as a separator to make the analyses of two segments of input text independent. Currently, the only recognized separator is the Unicode character u'\u001E', or ASCII RS (record separator). It will automatically be defined in the internal dictionary of all PyElly grammars as the syntactic type `SEPR`.

To be effective, `SEPR` must never be in any grammar rules leading to ambiguity in a sentence analysis and is best used at the level of the `SENT` syntactic type. In that case, it will allow you to divide a single long, complex sentence into syntactically independent parts to lessen the combinatorial explosion of phrase nodes arising from ambiguities. Your grammar rules will be responsible, however, for stitching back the results of such separated analyses to get proper output for the original input sentence.

Because ASCII control characters other than HT, CR, and LF are normally absent in natural language text, PyElly will always filter them out in its normalization of input. An ASCII RS can be inserted into text input, however, by PyElly macro substitutions, when it is specified as `\\S` in the result part of a macro rule (see Subsection 4.3). You can of course change PyElly Python code to define other control characters.

Outside of phrase and goal nodes, the main defined resource restrictions in PyElly are on the total number of syntactic types (64) and the total number of different syntactic features or of different semantic features for a phrase node (both 16). These are fixed to allow for preallocation of various arrays used by the PyElly parser in order to run faster. You can change these limits also, but will have to do so in the PyElly Python code; the current numbers should be quite enough for ordinary applications.

13. Developing Language Rules and Troubleshooting

PyElly rewrites input sentences according to the rules that you provide. A natural language application can involve up to eight different `*.*.elly` text files for rule definition, with some containing hundreds or even thousands of rules. There are plenty of ways to go wrong here; and so we all have to be systematic in developing PyElly applications, taking advantage of all the tools available when problems arise.

Even when trying to do something simple, you need to be constantly alert and cultivate good habits. In general, the best way to use PyElly is to approach a solution bit by bit, trying to take care of just one sentence at a time. Let PyElly to check everything out for you at each step and go no further until everything is satisfactory. Remember that a change in your rules can break previous analyses. Always assume the worst.

Application building will never be a slamdunk, but remember that you are already a natural language expert! Despite enormous advances in hardware and software, an intelligent young child nowadays still knows more about natural language than Siri or Watson. If you can harness some basic analytical skills and add some programming chops to this innate expertise, then you should do well with PyElly. Just set some clear goals and proceed slowly and carefully.

Start with the simplest sentences requiring the fewest rules. Once these can be handled successfully, move on to more complex sentences, adding more rules as needed to describe them. With a modular PyElly rule framework, you should be able to build on your previous rules without having to change them constantly. This is one big advantage of processing sentences recursively around the syntactic structures of sentences.

When testing out a new sentence, you should not only verify that PyElly is producing the right output, but also inspect its parse tree dump to see that it is what you expect from your current grammar rules. If everything is all right, then add the sentence and its translation to a list to be run in regression testing with `ellyMain.py` later. This testing will take only seconds, but must be done.

13.1 Pre-Checks on Language Rule Files

As your language definition files get longer, PyElly can help to verify that each of them is set up correctly and makes sense by itself before you try to bring everything together. This can be done by running the unit tests of the modules to read in definition files and checking the acceptability of rules. Running `ellyBase.py` or `ellyMain.py` will also do this to an extent, but you can sort out issues more easily when looking at only one definition file at a time. For example, with application `X`, you can run any or all of the following unit tests from your command line:

```
python grammarTable.py X

python vocabularyTable.py X

python patternTable.py X

python macroTable.py X

python nameTable.py X

python conceptualHierarchy.py X
```

Each command will read in the corresponding `X.*.elly` file, check for errors, and point out other possible problems. If a table or a hierarchy can be successfully generated, PyElly will also dump this out entirely for inspection, except for external vocabularies, which are usually too big for such examination.

The vocabulary, pattern, macro, and name table unit tests will also prompt you for additional examples to run against their rules for further verification of correct lookup or matching. This can be helpful when debugging a problem with a PyElly language definition and a specific problematic sentence.

PyElly error messages from language definition modules will always be written to sys.stderr and will have a first line starting with "`** `". They may be followed by a description line starting with "`*  `" showing the input text triggering the problem. For example, here is an error message for a bad FSA rule in its assigning a part of speech:

```
** pattern error: bad link
*  at [ 0 *bbbb* ZED start ]
```

PyElly will continue to process an input rule definition file after finding an error so as to catch as many definition problems as possible in one pass. No line numbers are given in error messages because PyElly normalizes all its input to make processing more simple for definition modules. This will strip out comments and will eliminate any blank lines, thus changing line numbers.

Once each separate table has been validated in isolation, you can run `ellyBase.py` to load everything together. Its unit test will run a cross-table check on the consistency of your syntactic categories and of your syntactic and semantic features. For application `X`, do this with the shell command

```
python ellyBase.py X
```

This is also how you would normally test the rewriting of individual test sentences, but the information provided by PyElly from the loading of language rules is a good way to look for omissions or typos in your language rules, which can be hard to track down otherwise. Running `ellyMain.py` will skip this detailed kind of checking.

13.2 A General Application Development Approach

For those wanting more specifics on setting up PyElly language definition files, here is one reasonable way to build a completely new application `X` step by step.

1. Set up initially empty `X.g.elly`, `X.m.elly`, `X.stl.elly`, `X.n.elly`, and `X.v.elly` files. For the other PyElly language definition files, taking the defaults should be all right as least for a start.

2. Select some representative target sentences for your PyElly application to rewrite. Five or six should be enough to start with. You can add more as you go.

3. Write `G:` grammar rules in `X.g.elly` to handle to handle one of your target sentences; leave out the cognitive and generative semantics for now. Just check for correctness of the rules by running the PyElly module `grammarTable.py` with `X` as an argument.

4. Add the words of a target sentence as `D:` internal dictionary rules in `X.g.elly` or as vocabulary entries in `X.v.elly`. Run `grammarTable.py` or `vocabularyTable.py` with `X` as an argument to verify correctness of language definition files as they change.

5. Run PyElly module `ellyBase.py` with `X` as an argument to verify that your language definition files can be loaded together. Enter single target sentences as input and inspect the parse data dumps to check that analyses are correct. Ignore any generated output for now.

6. Write the generative semantic procedures for your grammar rules and check for correctness by running `grammarTable.py`. If you have problems with a particular semantic procedure, copy its code to a text file and run the PyElly module `generativeProcedure.py` with the name of that text file as an argument.

7. When everything checks out, run `ellyBase.py` with `X` as an argument and verify that PyElly translates a target sentence as you want.

8. When everything is working for your target sentence, repeat the above from step 2 with more sentences. Always test your new system against all previous target sentences to ensure that everything is still all right after any change of language rules. You can create `X.main.txt` and `X.main.key` files to automate such testing.

13.3 Miscellaneous Tips

This subsection is a grab bag of advice about developing nontrivial PyElly grammars and vocabularies. It is distills experience going back to the PARLEZ system, the original PyElly ancestor. As with any software development effort, you must always expect to make mistakes in a language definition; but try to avoid making the same ones again when building up every new application.

1. PyElly is a simple system of about sixteen thousand lines of Python code. It is set up to translate natural language strings into other kinds of strings and nothing more. In other words, it will not by itself let you replicate Watson or Siri. Go ahead and push it to its limits, but be aware that PyElly is not magic and that you can do so much in limited amount of time.

2. The PyElly `ellyMain.py` module is the better choice to rewrite batches of raw text data because of its sentence recognition. If you are working with only one input sentence at a time, run `ellyBase.py`. This is more friendly for interactive processing and also provides more diagnostic information for its translations.

3. PyElly analysis always revolves around sentences, but remember that you can define them however you want and need not follow what you learned in 8th grade. It is all right to break sentences at colons and semicolons or even at subordinate conjunctions. This can greatly simplify your grammar rules.

4. In developing a grammar, less is better. You are more likely to get into trouble with more syntactic categories and more rules. In many applications, for example, you can ignore language details like gender, number, tense, and subject-verb agreement. Avoid defining rules for what does not matter to you.

5. Get the syntax of a target input language right before worrying about the semantics. PyElly automatically supplies you with stubs for both cognitive and generative semantics in grammar rules. You can replace these stubs later with full-fledged procedures and logic after you can successfully parse sentences.

6. Natural language always has regular and irregular forms. Tackle the regular first in your grammar rules and make sure you have a good handle on them before taking on the irregular. The latter can often be handled by macro substitutions: for example, just change DIDN'T to DID NOT or if tense is unimportant, DO NOT.

7. Writing semantic procedures for a grammar is a kind of programming. Therefore, follow good software engineering practices. Divide a large project into smaller parts that can be finished quickly and individually tested. Test thoroughly as you go; never wait until your language rules have all been written out.

8. Make your generative semantic procedures short, fewer than twenty lines if possible; this will make it easier to verify the correctness of your code visually. If a long procedure is unavoidable, run it first in a separate file with the unit test for the PyElly module `generativeProcedure.py`. Throw in some `TRACE`, `SHOW`, and `VIEW` commands temporarily to monitor what that code is doing.

9. Be liberal with named generative semantic subprocedures to shorten large blocks of code. A subprocedure may have as few as only two or three commands. Calling such a subprocedure could actually take more memory than equivalent inline commands,

but clarity and ease of maintenance will trump efficiency here. Common code used in multiple procedures should always be broken out as a named subprocedure.

10. Group the rules for syntactic types into multiple levels where the semantic procedures at each level will do similar things. Possible successive levels here might be (1) sentence types, (2) subject and predicate types, (3) noun and verb phrase types, and (4) noun and verb types with inflections. This is also a good way to organize the definition of local variables for communication between different generative semantic procedures.

11. Macro substitutions will usually be easier to use than syntax rules plus semantic procedures, but they have to be fairly specific about the words that they apply to. Syntax rules are better for patterns that apply to broad categories of words.

12. Macro substitution rules can be dangerous. Watch out for infinite loops of macro substitutions, which can easily arise with `*` wildcards. Macros can also interact unexpectedly; in particular, make sure that no macro is reversing what another is doing, which can also lead to infinite loops.

13. Try to avoid macros in which the result of substitution in longer that the original substring being replaced. These are sometimes necessary, but be careful here; PyElly will warn you if it comes across such a macro, but will allow it.

14. 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.

15. PyElly macro substitution comes after transformations of spelled out numbers, but before external vocabulary lookup, and entity extraction. It comes again just before the next token is taken from its input buffer after any inflectional stemming and again just before the next token is taken with morphological prefixes and suffixes split off. This will allow macros to undo any general word analysis in special cases. Be careful here; macros actions can undo, but cannot themselves be undone.

16. Macro substitutions and transformations can change the spelling of words in sentences being processed. Make sure that your internal dictionary grammar and external vocabulary rules take this into account.

17. Avoid macros for matching literal phrases like "International Monetary Fund." Unless you need the wild card matching supported by macros, use vocabulary tables instead. This will be faster.

18. Macros can be slow because all rules will be checked again after any successful substitution except when entire match is deleted by a rule. Each substitution will involve much copying and recopying of text.

19. Vocabulary building should be the last thing you do. You should define at least a few terms to support early testing, but hold off on the bulk of your vocabulary. Adding vocabulary is easy; adding grammar is complicated with more side effects.

20. For large ambiguous grammars, predefine as many vocabulary terms as possible in order to keep parsing reasonably fast. Use `ellySurvey.py` to find out what tokens appear in the text data you want to process and what definitional status they have with current language definition rules. The file `default.v.elly` (WordNet 3.1) is a good source of vocabulary rules with parts of speech for words.

21. Syntactic and semantic features can be quite helpful. The former lets you be more selective about which rules to apply to an analysis; the latter lets you better choose between different interpretations in case of ambiguities. These can reduce the total number of syntax rules, but will add overall complexity to your grammar.

22. PyElly will allow a syntactic category to be associated with only one set of syntactic feature names. Different syntactic categories may share the same set of syntactic features, however.

23. Features are encoded in a grammar table rule and in a parse tree node only as *n* anonymous bits. The same feature name may be in different feature sets, but will not necessarily refer to the same bit in a node. This can make for bad surprises when trying to inherit syntactic and semantic features through the `*L` or `*R` mechanisms; always inherit when nodes have the same set of feature names.

24. You must always say whether feature inheritance in a parse tree is from a left or a right descendant. There is nothing by default for either syntactic or semantic features. All the features from a designated descendant will be inherited, but you can always explicitly turn off particular inherited features afterward.

25. To dump out the entire saved grammar rule file for application `A`, run `grammarTable.py`. with `A` as an argument. This will also show generative and cognitive semantics, which `ellyBase.py` omits in its diagnostic output.

26. Keep the PyElly parse data dumps enabled in PyElly language analysis and learn to read them. This will be the easiest and most valuable way to obtain diagnostic information when your language rules are not working as expected (usually the case). Full dumps will show all subtrees generated for any ambiguous analyses, even those not incorporated into actual PyElly output, plus other parsing information.

27. PyElly will cut a tree display off at 25 levels of nodes by default. You can adjust this limit in the ellyMain and the ellyBase command lines, but deeper trees will be hard to interpret when output is broken up to fit into the maximum width of a display.

28. If you run into a parsing problem with a long input sentence, try to get the same problem it with a shorter version of that sentence. This will help to isolate the issue and also make your parse dumps easier to read.

29. When a parse fails, the last token in the listing shown in a parse tree dump will show approximately where the failure occurred. It may be a few tokens before, however. Check for a token for which no phrase node was generated. Look also at the last set of goals generated in a bottom-up parsing and see what token position they occur in.

30. If you are working with English input and have not defined syntax rules for handling the inflectional endings -S, -ED, and -ING, a parse will fail on them. The file `default.p.elly` will define these as the syntactic category `SUFX`, but you still need something in your grammar rules to work with them.

31. To verify the execution of a generative semantic procedure, put in a `TRACE` command. This will write to the standard error stream whenever it is run. In a procedure attached to a phrase node, it will show the syntactic type and starting position of that phrase node and the grammar rule governing the node. In a named subprocedure, PyElly will show the first attached generative procedure calling the subprocedure.

32. To see the value of local variables during execution for debugging, use the `SHOW` generative semantic command, which writes to the standard error stream. Remember that both local and global variables will always have string values, with an empty string possible. Local variables may be defined further up in a parse tree.

33. To see the contents of your current and your next output buffer at a given point in a generative semantic procedure, use the `VIEW` command. This is also a good way to learn what the various PyElly semantic commands do.

34. Minimize the number of `TRACE`, `SHOW`, or `VIEW` commands. Being overwhelmed with too much instrumentation can be as problematic as having too little information. Clean up such commands when no longer needed.

35. 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 unless specified as a single term by entity extraction or external vocabulary lookup. An underscore or an apostrophe is not normally a word break.

36. Quotation marks can be quite troublesome, since this may involve special Unicode forms as well as ambiguous uses of the ASCII apostrophe character. PyElly predefines quotation marks, including Unicode variations for formatted text, but these all must still be properly handled by your grammar rules.

37. Default vocabulary rules for punctuation can be overridden by `D:` rules defined by a PyElly application, but only when their cognitive semantics make these alternate interpretations preferred by PyElly.

38. Macro substitution will not apply across sentence boundaries. To override punctuation otherwise seen as a sentence stop, use the special PyElly stop exception rules described in Subsection 11.1.1. Note that macros can already deal with embedded periods and commas, since these will be non-stopping punctuation.

39. Ambiguity is often seen as a problem in language processing, but PyElly embraces it. Deliberate ambiguity can simplify a grammar. For example, the English word IN can be either a preposition or a verb particle. Define rules for both usages with

appropriate cognitive semantic plausibility scores and let PyElly figure out which one to apply.

40. Try always to give ambiguous alternatives different plausibility scores in external vocabulary rules as well as grammar rules; otherwise PyElly will switch between them arbitrarily when parsing different sentences in the same session. This is probably not what you want.

41. When assigning plausibility scores to rules, try to keep adjustments either mostly positive or mostly negative. Otherwise, they can cancel each other out in unexpected and often unfortunate ways. Plausibility for a phrase is computed by adding up all the plausibility scores for a phrase and all its immediate subconstituents.

42. To see what is going on with cognitive semantic logic for a particular phrase node, put `?>>?` as the first clause. This will turn on tracing for subsequent clauses in a rule and write to the standard error stream to verify that the logic is being executed and identify which clause is actually used to compute plausibility. This is also good for tracking the generation of new phrase nodes and seeing what grammar rule is used.

43. When a problem is tied to a particular input sentence, run the sentence by itself with `ellyBase.py` to get more diagnostic information. This will show what grammar rule is tied to each phrase node and what ambiguities it is associated with.

44. A common failing is when the wrong interpretation gets the highest plausibility score. To remedy this, identify the phrase nodes contributing most to that score and check their ambiguous counterparts for a better interpretation. This should tell you where plausibility scoring needs adjustment in grammar or vocabulary rules. You may have to go up or down in a parse tree to find which ambiguities to focus on.

45. The predefined `*unique` syntactic feature allows you to control where PyElly ambiguity resolution happens. A phrase marked with that feature will remain unresolved in PyElly parsing even when there is another phrase of the same syntactic type and with the same syntactic features generated over the same set of input sentence tokens. This feature cannot be inherited.

46. Ambiguity will produce many possible parsings of a sentence, often leading to exponential growth in total processing. A sentence with four points of ambiguity, each with two possible interpretations, has sixteen possible analyses. The problem worsens in longer sentences with more room for ambiguities. To minimize this problem, avoid input tokens taken to be `UNKN`, and be sparing with the syntactic feature `*unique` in grammar rules. When reasonable, recognize multi-world phrase as a single token. Use RS control characters to partition analyses if possible.

47. The PyElly name recognition capability is based on the idea that we can list out the most common names in a particular domain of text input and that other names will be lexically and phonetically similar and can be contextually inferred. This will help reduce the number of `UNKN` tokens seen by PyElly and will help in recognizing surnames after the appearance of a full name.

48. The `name.n.elly` definition file is actually quite diverse, being based on U.S. Census data, but if you are working primarily with foreign names of one particular kind like Arabic, Chinese, or Russian, you probably want to extend your tables with sample names from other sources. Do not expect to get away with only a little work here. Current PyElly phonetic signatures reflect American pronunciation and may have to be adjusted to make inferences about foreign names more reliable.

49. Experiment. PyElly offers an abundance of language processing capabilities, and there is often more than one way to do something. Find out what works best for you.

50. Fix any language definition problems due to typos first. It is easy to mistype names of syntactic categories or names of semantic or syntactic features. Always check the complete listing of grammar symbols in ellyBase diagnostic output to verify that there are no unintended ones due to typos.

51. PyElly's use of '`#`' as a marker for comments in rule files and as a wildcard matching numeric characters '`0`' through '`9`' is a hazard. To be safe, use a backslash (`\`) to escape a single '`#`' wildcard in a rule file so as to make it unambiguous. Always put a space after the first '`#`' marking a comment.

52. Whenever a grammar or a vocabulary is complex, something will almost always be wrong with it. Check underlying analyses to make sure the PyElly is really doing what you expect; it is not enough for just the final output to look correct. Watch out especially for red flags like ambiguous phrases with the same plausibility score.

53. You need to test thoroughly any language definition being developed. Always collect at least a dozen sentences plus their expected translations to test PyElly processing for a given application. Repeat some sentences in your test set to check for possible variations in the resolution of ambiguities.

54. In natural language processing, you have to be systematic and committed for the long haul; always rerun your test sets after any significant change in language definition rules. Even slight changes in your language definition rules can result in PyElly being unable to handle test sentences that it could translate previously.

55. Being able to rewrite *n* sentences successfully will not guarantee that you can handle the next sentence. You can only increase the probability of success here by trying out a PyElly application on a large, diverse set of sample sentences.

56. You may have to scrap some earlier language definitions to make a clean start on handling especially problematic language features. Expect to learn from failures and always be ready to change your processing strategies as necessary.

An adult learning a second language will discover how hard it is hard to achieve even a minimum level of fluency. The same is true for building a non-trivial natural language system, even in the age of Siri, Watson, and Google Translate. Tools like PyElly can take you a good part of the way to credible functionality, but you still need to choose between the various options available and fill in all the messy details.

14. PyElly Applications

PyElly by itself is no silver bullet for natural language processing despite its builtin capabilities. Within its limitations, however, you can still produce some quite useful results quite quickly with text data. A good candidate application for PyElly implementation should 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 offers the most builtin support.

2. Your intended output will be translations of fairly short input sentences into 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 might be expected in a simple dictionary.

4. You understand what your translation would involve and accomplish at least on a broad scale and mainly 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 comfortable with trial and error development of language definition files and are willing to put up with idiosyncratic non-commercial software.

Familiarity with the Python language will help here, but is not critical. You will, however, definitely have to write the code for PyElly cognitive and generative semantics. This is nontrivial work, but it will be in a highly restricted programming language, which should be straightforward for someone with basic coding experience.

To build a PyElly application `A`, you first need to define its rules through language definition files. In the extreme case, these would include `A.g.elly` (grammar), `A.v.elly` (vocabulary), `A.m.elly` (macro substitutions), `A.p.elly` (syntactic type patterns), `A.n.elly` (name components), `A.ptl.elly` (prefix removal rules), `A.stl.elly` (suffix removal rules), `A.h.elly` (semantic concept hierarchy), and `A.sx.elly` (stop punctuation exceptions). Only `A.g.elly` is mandatory; the rest can be either be empty files or omitted. If you omit other definition files, the respective files for `default` will be loaded instead.

Here are three fairly simple application projects you can try to build 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 part of speech tagger for English words just using morphological analysis.

The PyElly distribution includes examples of actual applications. They fall into two classes: those used for debugging and validation only and those realizing potentially useful functionality. PyElly integration testing consists of running three debugging and validation applications plus all of the functional applications on preset test data files and checking that the results are as expected (See Appendix D).

Below, we shall list which the language definition files are provided with each example application. You can look at the rules in these files to see how to write them for your own PyElly applications.

Currently, five applications are used for debugging or validation only:

default (`.g`,`.m`,`.p`,`.ptl`,`.stl`,`.sx`,`.v`) - not really an application, but a set of language definition files that can be substituted if your PyElly application does not specify one. These include rules for sophisticated morphological stemming and vocabulary rules covering most of the terms in WordNet 3.0.

echo (`.g`,`.m`,`.p`,`.stl`,`.v`) - a minimal application that echoes its input as analyzed by PyElly into separate tokens. It will, however, show the effect of inflectional stemming in English on words and entity transformations. You can disable that stemming in your `ellyMain.py` command line (see Section 7).

```
input:  Her faster reaction startled him.
output: her fast -er reaction 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 only simple phrase structures with a minuscule vocabulary. This essentially replicates various testing done to validate PyElly through its early alpha versions.

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

stem (`.g`,`.m`,`.p`,`.v`) - to check that PyElly inflectional stemming is properly integrated with both internal and external vocabulary lookup. This can be quite tricky because such stemming automatically happens in multiple PyElly modules; it all needs to be checked out in integration testing with a set of examples covering cases of interest.

```
input:  Dog's xx xx xx.
output: dog-'s xx xx xx.
```

bad (`.g`,`.h`,`.m`,`.n`,`.p`,`.sx`,`.v`) - deliberately malformed language rules to test PyElly error detection, reporting, and recovery. This is a big part of checking out PyElly, but no grammar or vocabulary table will be generated because of the malformed rules, and so PyElly will be unable to translate any input here. The files `bad.main.txt` and `bad.main.key` are defined for use with `doTest`, but are empty.

```
input:  - -
output: - -
```

The second, more substantial, class of applications are mostly derived from various demonstrations written for PyElly or its predecessors. These have nontrivial examples of language definitions, illustrate various PyElly capabilities, and provide a basis for broad integration testing. Most of the applications are only prototypes, but you can flesh them out for more operational usage by adding more vocabulary and grammar rules.

indexing (`.g`,`.p`,`.ptl`,`.v`) - to check removal of purely grammatical words (stopwords), stemming, morphological analysis, and dictionary lookup. Since it obtains roots of content words from arbitrary input text, it could be used to predigest input English text for information searching, statistical data mining, or machine learning systems. This application was written for PyElly.

```
input:  We never had the satisfaction.
output: - - - - satisfy -
```

Note that each non-content word and word fragment in input will be replaced with a hyphen (-) in output to indicate the extent of PyElly processing explicitly.

texting (`.g`,`.m`,`.p`,`.ptl`,`.stl`,`.v`) - with a big grammar and nontrivial generative semantic procedures. This implements a more or less readable text compression similar to that seen in mobile messaging. It is a demonstration written originally for the Jelly predecessor of PyElly and shows how a full-fledged compression application might be approached.

```
input:  Government is the problem.
output: govt d'prblm.
```

doctor (`.g`,`.m`,`.ptl`,`.stl`,`.v`) - This has a big grammar with extensive ambiguity handling required. It uses the PyElly `...` syntactic type to set up grammar rules to implement Weizenbaum's Doctor program for Rogerian psychoanalysis, incorporating the full keyword-based script published by him in 1966. The language definition rules were first written for the nlf predecessor of PyElly and then adapted for Jelly and now PyElly.

```
input:  My mother is always after me.
output: CAN YOU THINK OF A SPECIFIC EXAMPLE.
```

chinese (.g,.m,.ptl,.v) - a test of PyElly Unicode handling, translating simple English into either traditional [tra] or simplified [sim] Chinese characters. Both the grammar and the vocabulary of this application are still being worked on.

```
input:  they sold those three big cars.
output: [sim]他们卖了那三辆大汽车.
output: [tra]他們賣了那三輛大汽車.
```

In actual operation, only one form of Chinese output will be shown at a time. You get traditional character output when ellyMain.py is run with the flag -g tra . The default is simplified output as with the option -g sim. The current integration test is with traditional characters. Work on this application started in Jelly, but was filled out in PyElly where most of the rule development was done.

querying (.g,.m,.ptl,.stl,.v) - heuristically rewrites English queries into SQL commands directed at a structured database of Soviet Cold War aircraft organized into multiple tables. This is a reworking of language definition files for the very first nontrivial application written for the PARLEZ and AQF predecessors of PyElly; it was updated in PyElly to produce SQL output.

```
input:  how high can the foxbat fly?
output: from Ai a,AiPe b
        select ALTD
        where NTNM=foxbat,a.NTNM=b.NTNM
        ;
```

Table and field names are abbreviations: Ai is "aircraft," AiPe is "aircraft performance, ALTD is "altitude," NTNM is "NATO name," and so forth. The original AQF system aimed to make such names transparent to database users as well as to hide the mechanics of query formation.

marking (.g,.m,.p,.v) - rewrite raw text with shallow XML tagging, a candidate for the canonic PyElly application. This shows how someone might preprocess text data for easier mining. It has the most complex grammar and vocabulary rules of all PyElly example applications and especially notable in its cognitive semantics.

```
input:  The rocket booster will carry two satellites into orbit.
output: <sent>
        <nclu><det>the</det><noun>rocket booster</noun></nclu>
        <vclu><aux>will</aux><verb>carry</verb></vclu>
        <nclu><num>2</num><noun>satellite -s</noun></nclu>
        <nclu><prep>into</prep><noun>orbit</noun></nclu>
        <punc>.</punc></sent>
```

This application is still evolving in its language definition rules. It is a big part of the PyElly integration test suite with a large and growing input set of sample text from the Worldwide Web. That test data has been quite challenging and has already been helpful in uncovering many bugs and other problems in PyElly code.

name (.g, .m,.n,.p,.ptl,.stl,.v) - identify personal names in part or in whole from raw text by lookup and by inference. This capability could eventually be merged with **marking**, but it is useful to keep separate as an integration test validating the special name recognition modules now available in the PyElly entity extraction framework as of release v1.1.

```
input:  John Adams married Abigail Smith of Weymouth in 1764.
output: "John Adams"
        "Abigail Smith"
```

The application uses a tule table based on 2010 U.S. Census data. It recognizes the 1,000 most common surnames, the 900 most common male names, and the 1,000 most common female names; and it can infer other name components from context and gauge their plausibility. This heuristic logic is still experimental.

disambig (.g,.h,.stl,.v) - disambiguation with a PyElly conceptual hierarchy by checking the semantic context of an ambiguous term. This is only a demonstration and not yet a full application like the others listed here. It was written mainly as an integration test focusing on cognitive semantics. Its output is a numerical scoring of semantic relatedness between pairs of possibly ambiguous terms in its input and showing their intersection in a conceptual hierarchy along with the actual WordNet 3.1 concepts assigned to them by PyElly.

```
input:  bass fish.
output: 11 00015568N=animal0n: bass0n/[bass] fish0n/[fish]
```

This uses the PyElly output option to show the plausibility of a translation along with the translation itself, which is right of the second colon (:) in the output line. The plausibility will be left of that colon. The output score here is 11, which is high, and the output also includes the concepts associated with a sentence analysis. The example above shows that the intersection of `bass` and `fish` is under the WordNet concept `00015568n`, which has the label `animal0n`.

These eight example applications show the range of possibilities for simple translations of natural language. PyElly is no one-trick pony. All the processing is still rather basic, but it is quite competent and can help you to produce helpful results over a broad range of text data.

In integration testing, the **echo**, **test**, **stem**, **indexing**, **texting**, **doctor**, **chinese**, **querying** , **marking**, **name**, and **disambig** applications each have input files `*.main.txt` for ellyMain to translate. The expected results here are given by the corresponding files `*.main.key`. To run an actual test with a PyElly application, you can use the bash shell script `doTest` in the PyElly distribution (see Appendix D).

Other PyElly applications possible in the short term with the current and future versions of PyElly are:

`translit` - transliterate English words into a non-Latin alphabet or into syllabic or ideographic representation.

`editing` - detect and rewrite verbose English into concise English, correcting common misspellings.

`anonymizing` - remove identifying elements from text: names, telephone numbers, addresses, and so forth.

`nomenclature` - identify standard chemical names in text. This is to allow for more accurate processing of technical text, as such names are often mangled otherwise.

`blended` - a super application extending the language definition rules of **`marking`** with the name recognition of **`name`**, and the big vocabulary of **`default`**.

As seen in the example applications already implemented, PyElly can handle many different kinds of natural language processing. So far, their grammar rules have been only in the hundreds and their vocabulary rules only in the low thousands, but these can be extended further with the investment of a few months or even a few weeks of work.

Eventually, someone might write an application reading input text other than in English. This data currently would still have to be encoded in the ASCII and Latin 1 blocks of Unicode, but that should allow for French, Spanish, German, Czech, or Hungarian input. New rules for inflectional and morphological analysis will be needed, as PyElly does this so far only for English.

On the whole, PyElly is a throwback as a natural language system in that it involves the manual crafting of large numbers of language rules in contrast to current practice favoring unsupervised automatic machine learning. A rule-based approach can be advantageous, however, if someone else has already worked out most of the rules. You might as well exploit prior work to get a head start in system building, even though you may still rely on machine learning to achieve your primary functionality.

The intent behind PyElly is to give users more options in how they approach their text data. If you are interested in only a part of your data or want your data in a cleaner form, then PyElly can transform it to suit your purposes. For example, to focus on the ingredients in recipes without regard to their proportions, remove quantity specifications before running a deep-learning neural network on your recipes.

A PyElly application will usually be unable to translate all its text data perfectly, but it really needs only to be good enough to give better results than what you might get with totally raw data. PyElly is still evolving and will be going through more testing and experimentation, but this is standard practice in software engineering nowadays, where development goals are expected to be temporary way stations.

A PyElly application will not pass the Turing test anytime soon. PyElly by itself lacks logical and common sense processing capabilities, which will limit its range of possible language understanding. The plan, though, is to continue in building up its repertory of basic linguistic support short of full artificial intelligence, aiming for solid computational competence in text handling that could help in data analysis right now.

The entire PyElly package is free and compact enough for home use. It operates at a more detailed level as compared to other natural language toolkits, but this should help students and other novices to get a better feel for the nuts and bolts of natural language processing and explore about how key capabilities work. Natural language is often hard to handle, and one has to learn where the bumps on the road lie.

More experienced information users can also use the flexible rewriting capabilities of PyElly to get a better handle on uncontrolled text data. PyElly includes many prebuilt resources for working with English text, and its `marking` example application is a good starting point for defining rules for handling unrestricted English text data. Give PyElly a try on your favorite text corpora. Any criticisms or suggestions here will be welcome, and of course, anyone may freely improve on the PyElly open-source Python code.

Appendix A. Python Implementation

This appendix is for Python programmers. You can run PyElly without knowing its underlying implementation, but at some point, you may want to modify PyElly or embed it within some larger information system. The Python source code for PyElly is released under a BSD license, which allows you to change it freely as needed. You can download it from

`https://github.com/prohippo/pyelly.git`

PyElly was written in Python 2.7.5 under Mac OS X 10.9 and 10.10; it will not run under earlier versions of Python because of changes in the language. To implement its external vocabulary tables, PyElly v1.2+ no longer requires the Berkeley Database (BDb) database manager or the bsddb3 third-party BDb Python API wrapper. The PyElly code with BDb in release v1.1 or earlier should be discarded because a GPL copyleft license would be required for it, greatly complicating freedom of use. Vocabulary table lookup code has also evolved greatly in PyElly v1.2 and v1.3.

Currently, the PyElly v1.3+ source code consists of 64 Python modules, each a text file named with the suffix `.py`. All modules were written to be self-documenting through the standard Python `pydoc` utility. When executed in the directory of PyElly modules, the command

```
pydoc -w x
```

will create an `x.HTML` file describing the Python module `x.py`.

The code was written neither for speed of execution nor for space efficiency. This is normal in Python development practices, however, because it is an interpreted language, and it is consistent with PyElly's emphasis on quickly putting together a broad range of functionality for doing useful language processing right now. Although the code has become fairly stable through extensive testing, it remains experimental and still keeps many debugging print statements that can be reactivated by uncommenting them.

The algorithms underlying PyElly have become somewhat intricate after decades of tinkering. Section 12 of this manual does describe the bottom up parsing approach of PyElly, but other important aspects of the system have to be gleaned from source code. The most notable of these are wildcard string matching, macro substitution, the non-deterministic finite automaton for token pattern matching, compiling PyElly code for generative and cognitive semantics, and multi-element vocabulary table lookup.

Here is a listing of all 64 current PyElly Python modules grouped by functionality. Some non-code definition and data files are also included below when integral to the operation or builtin unit testing of the modules there. They are shown in the dark-shaded rows of the tables.

Inflectional Stemmer (English)	
`ellyStemmer.py`	base class for inflection stemming
`inflectionStemmerEN.py`	English inflection stemming
`stemLogic.py`	class for stemming logic
`Stbl.sl`	remove -S ending
`EDtbl.sl`	remove -ED ending
`Ttbl.sl`	remove -T ending, equivalent to -ED
`Ntbl.sl`	remove -N ending, a marker of a past participle
`INGtbl.sl`	remove -ING ending
`rest-tbl.sl`	restore root as word
`spec-tbl.sl`	restore special cases
`undb-tbl.sl`	undouble final consonant of stemming result

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`	manipulating text input with macro substitutions
`macroTable.py`	for storing macro substitution rules
`patternTable.py`	extraction and syntactic typing by FSA with pattern matching

Parsing	
`symbolTable.py`	for names of syntactic types, syntactic features, generative semantic subprocedures, global variables
`syntaxSpecification.py`	syntax specification for PyElly grammar rules
`featureSpecification.py`	syntactic and semantic features for PyElly grammar rules
`grammarTable.py`	for grammar rules and internal dictionary entries
`grammarRule.py`	for representing syntax rules
`derivabilityMatrix.py`	for establishing derivability of one syntax type from another so that one can make bottom-up parsing do nothing that top-down parsing would not
`ellyBits.py`	bit-handling for parsing and semantics
`parseTreeBase.py`	low-level parsing structures and methods
`parseTreeBottomUp.py`	bottom-up parsing structures and methods
`parseTree.py`	the core PyElly parsing algorithm
`parseTreeWithDisplay.py`	parse tree with methods to dump data for diagnostics

Semantics	
`generativeDefiner.py`	define generative semantic procedure
`generativeProcedure.py`	generative semantic procedure
`cognitiveDefiner.py`	define cognitive semantic logic
`cognitiveProcedure.py`	cognitive semantic logic
`semanticCommand.py`	cognitive and generative semantic operations
`conceptualHierarchy.py`	concepts for cognitive semantics

Sentences and Punctuation	
`ellyCharInputStream.py`	single char input stream reading with unread() and reformatting
`ellySentenceReader.py`	divide text input into sentences
`stopExceptions.py`	recognize stop punctuation exceptions in text
`exoticPunctuation.py`	recognize nonstandard punctuation
`punctuationRecognizer.py`	define single-character punctuation defaults

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

Entity Extraction	
`entityExtractor.py`	runs Python entity extraction procedures
`extractionProcedure.py`	some predefined Python entity extraction procedures
`simpleTransform.py`	basic support for text transformations and handling of spelled out numbers
`dateTransform.py`	extraction procedure to recognize and normalize dates
`timeTransform.py`	extraction procedure to recognize and normalize times of day
`nameRecognition.py`	identify personal names
`digraphEN.py`	letter digraphs to establish plausibility of possible new name component
`phondexEN.py`	get phonetic encoding of possible name component
`nameTable.py`	defines specific name components

Top Level	
`ellyConfiguration.py`	define PyElly parameters for input translation
`ellySession.py`	save parameters of interactive session
`ellyDefinition.py`	language rules and vocabulary saving and loading
`ellyPickle.py`	basic loading and saving of Elly language definition objects
`interpretiveContext.py`	handles integration of sentence parsing and interpretation
`ellyBase.py`	principal module for processing single sentences
`ellyMain.py`	top-level main module with sentence recognition
`ellySurvey.py`	top-level vocabulary analysis and development tool
`dumpEllyGrammar.py`	methods to dump out an entire grammar table

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

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 list of cases to handle
morphologyTest.txt	to support unit test with prefix and suffix tree logic plus inflectional stemming
sentenceTestData.txt	to support unit test of sentence extraction
testProcedure.*.txt	to run with the generativeProcedure.py unit test to verify correct implementation of generative semantic operations
*.main.txt	Input text for integration testing
*.main.key	Expected output text for integration testing with provided input

All `*.py` and `*.sl` files listed above are distributed together in a single directory along with `*.main.*` integration test files. The `*.txt` files for unit testing will be in a subdirectory `forTesting`.

The first v0.1beta version of the Python code in PyElly was all written in 2013 with some preparatory work done in November and December of 2012. This was an extensive reworking and expansion of the Java code in its Jelly predecessor, making it no longer compatible with Jelly language definition files. PyElly v1.0 moved beyond beta status as of December 14, 2014, but development is still ongoing.

The emphasis in PyElly is now moving away from adding on new Python modules and moving towards better reliability and usability. Existing modules will continue to evolve, but mainly to provide better support for building real-world applications to process unrestricted natural language text. This will involve the eventual construction of big grammars and big vocabularies, which should be the ultimate test of any natural language tool.

Appendix B. Historical Background

The natural language tools in PyElly have evolved greatly in the course of being completely written or rewritten five times in five different languages over the past forty years. Nevertheless, it retains much of the flavor of the original PDP-11 assembly language implementation of PARLEZ. Writing such low-level code forced simplicity in software architecture, but this has been quite advantageous in porting the system to different target computing platforms.

The PARLEZ system, for example, had its own stripped-down custom programming language for generative semantics because nothing else was available at the time. That solution, however, has been serviceable for general rewriting of natural language text and so has been carried along with only a few changes and additions in systems up to and including PyElly. And, yes, arithmetic is still unsupported.

PyElly does depart in major ways even from its immediate predecessor Jelly, however.

- The inflectional stemmer was reorganized to simplify its set of basic operations and to eliminate internal recursive calling. Stemming logic is now set out in editable text files for loading at run-time. The number of special cases recognized in English has been expanded. The -N and -T inflectional past tense endings were added.
- Morphological analysis from Jelly was enhanced to allow proper identification of removed prefixes and suffixes as well as returning stems (lemmas). This results in a true analytic stemmer in PyElly, more appropriate to a general natural language tool. The number of suffix cases recognized in English was greatly expanded in PyElly to cover many WordNet exceptions and other English irregular forms.
- The syntactic type recognizer was changed to employ an explicit finite-state automaton where transitions are made when an initial part of an input string matches a specified pattern at a state. A special null pattern was added to give more flexibility in defining automata.
- New execution control options were added to generative semantics. Local and global variables were changed to store string values, and list and queue operations were defined for local variables. Deleted buffer text can now be recovered in a local variable. Support for debugging and tracing was expanded.
- Semantic concepts were added to cognitive semantics for ambiguity handling. This makes use of a new semantic hierarchy with information derived from WordNet.
- Vocabulary tables were made more easily manageable by employing the SQLite package to manage persistent external data. An external vocabulary rule is now limited to a single input line to facilitate generating definitions en masse.
- Support for recognizing and remembering personal names or name fragments is now an optional part of PyElly entity extraction. This includes a large prebuilt name table based on 2010 U.S. Census data.

- A new interpretive context class was introduced to coordinate execution of generative semantic procedures and consolidate data structures for parsing and rewriting.
- Handling of Unicode was improved. UTF-8 is now employed in loading of all language definition files and in interactive PyElly input and output.
- Ambiguity resolution was completely overhauled with improved cognitive semantics.
- Sentence and punctuation processing is cleaner and more comprehensive. Unicode formatted punctuation is now recognized.
- The PyElly command line interface was reworked to support new initialization and rewriting options.
- Error handling and reporting has been greatly broadened for the definition of language rules. Warnings have also been added for common problems in definitions.
- New unit tests have been attached to major modules. Integration tests with important example applications were developed to provide comprehensive code validation.
- New example applications have also been written to show the range of PyElly processing. These also serve as integration tests exercising the range of PyElly natural language capabilities; older applications from Jelly and earlier systems have been converted to run in PyElly.
- Many bugs from Jelly or earlier were found by extensive testing and fixed.

Jelly is now superseded and retired. This in part reflects the growing importance of scripting languages like Python in software development and education.

PyElly is by no means perfect or complete and might be rewritten in yet another programming language as computing practices change. The goal here, however, is less a long-term utopian system than an integrated set of reliable natural language processing tools and resources immediately helpful to students and others in building practical natural language applications. Many PyElly tools and resources will seem old-fashioned to some technologists; but most have had time to mature and prove their usefulness. There is no point in continually having to reinvent or relearn such capabilities.

Going forward, we want to implement and demonstrate a robust capability to process arbitrary English text in a nontrivial way. Currently, this has been manifested in the context of the `marking` example application introduced after the v1.0 release of PyElly. The language definition problem here is quite complex and has been helpful in exercising almost all of PyElly except for conceptual hierarchies.

This PyElly User's Manual revises, reorganizes, and greatly extends the earlier one for Jelly, but still retains major parts from the original PARLEZ Non-User's Guide, which was once printed out on an early dot-matrix printer. Editing for clarity, accuracy, and completeness continues. Check https://github.com/prohippo/pyelly.git for the latest PDF for the manual.

Appendix C. Berkeley Database and SQLite

Berkeley Database is an open-source database package available by license from Oracle Corporation, which in 2006 bought SleepyCat, the company holding the BDb copyright. BDb was the original basis for PyElly external vocabulary tables, but has been replaced by SQLite because of changes in Oracle licensing policy for BDb.

The current PyElly v1.2+ vocabulary tables with SQLite should incur no noticeable performance penalty despite having to access all persistent data through an SQL interface instead of function calls. SQLite will allow PyElly to remain under a BSD license, since SQLite is included in the Python 2.7.* and 3.* libraries. It does not have to be downloaded separately.

If running with Berkeley Database is really important, you can download it yourself and return to the former PyElly vocabulary table code in v1.1. The latest versions of BDb do come with a full GNU copyleft license, though, with possibly entailing unattractive legal implications. The apparent intent of Oracle here is to compel many users of BDb to buy a commercial license instead of using free open-source code.

The Python source of `vocabularyTable.py` in the previous PyElly v1.1 release has NOT been updated, however, to display a GNU copyleft license as required for use of Bdb. That will not happen unless there is a reason to fork off a specific BDb version of PyElly in the future.

Downloading Berkeley Database and making it available in Python is a complex process depending on your target operating system. You will typically need Unix utilities to unpack, compile, and link source code. For background on Berkeley Database, see

http://en.wikipedia.org/wiki/Berkeley_DB

For software downloads, you must go to the Oracle website

http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/index.html

to get the latest Berkeley Database distribution file. The instructions for doing so on a Unix system can be viewed in a browser by opening the Berkeley Database documentation file:

```
db-*/docs/installation/build_unix.html
```

The installation procedure should be fairly straightforward for anyone familiar with Unix. An actual MacOS X walkthrough of such compilation and installation can be found on the Web at

https://code.google.com/p/tonatiuh/wiki/InstallingBerkeleyDBForMac

To access Berkeley Database from Python, you must next download and install the bsddb3 package from the web. This is available from

https://pypi.python.org/pypi/bsddb3

The entire installation procedure turns out to be quite complicated, however, and difficult to carry out directly from a command line. The problem is with dependencies where a module A cannot be installed unless module B is first installed. Unfortunately, such dependencies can cascade unpredictably in different environments, so that one fixed set of instructions cannot always guarantee success.

To avoid missteps and all the ensuing frustrations, the best approach is use a software package manager that will trace out all module dependencies and formulate a workable installation path automatically. On MacOS X, several package managers are available, but the current favorite is `homebrew`. See this link for general details:

http://en.wikipedia.org/wiki/Homebrew_(package_management_software)

As it turns out, `homebrew` will also handle the installation of Berkeley Database and the upgrading of Python on MacOS X to version 2.7.5 (recommended for bsddb3). If you have a MacOS system with Xcode already installed, you can follow these steps to download the `homebrew` package and use its `brew` and `pip` commands to get Berkeley Database:

```
# get homebrew
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

# get latest Python
brew install python —framework

# get BdB
brew install berkeley-db
sudo BERKELEYDB_DIR=/usr/local/Cellar/berkeley-db/5.3.21/ pip install bsddb3
```

(This procedure is subject to change. See the latest pertinent webpages for the most current information.)

This web page explains what is going on here:

http://stackoverflow.com/questions/16003224/installing-bsddb-package-python

The `homebrew` package manager is helpful because it maintains a shared community library of tested installation "formulas" to work with. These resources are specific to MacOS X, however, making `homebrew` inapplicable to Windows or even Linux or other Unix operating systems. If you are running on a non-MacOS X platform, you have to turn to other software package managers; see

http://en.wikipedia.org/wiki/List_of_software_package_management_systems

Some of these managers implement parallels to `homebrew` commands, but you will have to check what parts are actually equivalent.

Appendix D. PyElly System Testing

After any major change to PyElly source code, you should thoroughly validate the resulting system. Check that every Python module compiles with no errors and run the current suite of unit and integration tests in the PyElly package.

The following PyElly Python modules have builtin unit tests, executed with the command `python M.py`, where `M` is the module name:

```
cognitiveDefiner        cognitiveProcedure      conceptualHierarchy
conceptualWeighting     dateTransform           derivabilityMatrix
dumpEllyGrammar         ellyBase                ellyBits
ellyBuffer              ellyBufferEN            ellyCharInputStream
ellyDefinition          ellyDefinitionReader    ellySentenceReader
ellySurvey              ellyWildcard            entityExtractor
featureSpecification    generativeDefiner       generativeProcedure
grammarRule             grammarTable            inflectionStemmerEN
macroTable              morphologyAnalyzer      nameRecognition
nameTable               parseTree               parseTreeBase
parseTreeBottomUp       parseTreeWithDisplay    patternTable
phondexEN               prefixTreeLogic         punctuationRecognizer
simpleTransform         stemLogic               stopExceptions
substitutionBuffer      suffixTreeLogic         syntaxSpecification
timeTransform           treeLogic               vocabularyTable
```

Most of these unit tests are self-contained with predefined input test data, but some also will read sys.stdin to get additional input for testing:

```
ellyBase                ellyBufferEN            ellyCharInputStream
entityExtractor         inflectionStemmerEN     macroTable
morphologyAnalyzer      nameRecognizer          nameTable
patternTable            phondexEN               stemLogic
substitutionBuffer      suffixTreeLogic         vocabularyTable
```

Running your own examples with the unit test of one of these modules will let you track down a specific problem more easily. You can enter as many inputs as you want; just type a `<RETURN>` by itself to terminate the input loop here. Manually entered test examples will be optional for pre-release PyElly validation, however.

The following PyElly modules have specific test input files included in the standard distribution from GitHub. Their associations are as follows:

```
cognitiveDefiner:          cognitiveDefinerTest.txt
ellySentenceReader:        sentenceTestData.txt
generativeDefiner:         generativeDefinerTest.txt
generativeProcedure:       testProcedure.*.txt
morphologicalAnalyzer:     suffixTest.txt
```

In unit testing, these files are either specified by name in a commandline argument or read from redirected standard input. See the Python code for each particular PyElly module to see how to do this.

For integration testing, the `doTest` shell script with argument `A` will run `ellyMain.py` with preselected parameters for application `A` while reading input from `A.main.txt`. In a full test, run `doTest` with each of the PyElly applications having language definition files in the `applcn` subdirectory of the PyElly download package:

```
./doTest echo
./doTest test
./doTest stem
./doTest indexing
./doTest texting
./doTest doctor
./doTest chinese
./doTest querying
./doTest marking
./doTest name
./doTest disambig
```

After processing `A.main.txt`, you can compare the actual translated output of `doTest` with the corresponding `A.main.key` file as follows:

```
./doTest A > A.out
diff -bB A.out A.main.key
```

When PyElly is running correctly, the matchup will not always be exact because of extra output produced by `ellyMain.py`, but it should be fairly obvious where actual disagreement exists with `A.main.key`. For example, `diff` might produce

```
1,3d0
< system= querying
< standard input from file already
<  135 rules
```

when running with output from the `querying` integration test. Here, the mismatch in output is unimportant because it is only with incidental status reporting and not from any actual differences in PyElly translation.

Integration testing will also serve as regression testing to verify that a new version of PyElly can still do what it used to. This is important to do frequently. Otherwise, problems in code can quickly compound, making any debugging quite difficult.

After any major change to PyElly code or to language definitions for a test application or its integration test input, you should also update the `*.main.key` files to reflect any intended differences these will make in PyElly translations.

With `doTest`, you can also process input from a particular file `x.txt`. while running the same language definition files for an application `A`. Do this by redirecting the standard input for `doTest` as follows:

```
./doTest A < x.txt
```

The `marking` integration test in the PyElly package now includes the extra file `marking.more.main.txt`, which can be run in the above way to extend the test cases of `marking.main.txt`. A corresponding `marking.more.main.key` allows for comparison of results against expected output. This splitting off of test data is to keep the running time of `doTest` from getting too long with any single batch of input.

The suite of integration test applications serves to cover the broad range of PyElly processing demonstrated historically in PyElly as well as in its predecessors. The test instances are still limited, however, so that success here is in no guarantee that a given language definition will work with all future instances. We can only achieve a reasonable level of competence at some point and then continually strive to do better.

The `marking` integration test currently is the most important. It has the most grammar and vocabulary rules and has about as many test instances as all the other integration tests combined. Its input consists of actual text collected from across the Web, not sentences cooked up with specific characteristics. Such data has been a challenge to process and so has been a good way to shake down both PyElly code and language definition rules.

In upcoming PyElly work, the most logical strategy is to keep enlarging the scope of integration testing. Fortunately or unfortunately, finding sentence examples to break PyElly processing is still disconcertingly easy when running with language definitions for current applications. So everything always has to be tentative when building an application; but then that is how human beings naturally learn a language anyway.

Focus going forward will be on the `marking`, `chinese`, and `name` example applications. The main goal here is to see how far PyElly can be pushed while still remaining workable. So far, experience here with `marking` has been encouraging; despite many challenging issues and uncovered bugs, PyElly resources have been adequate, although this necessitated adding control characters for avoiding phrase node overflow.

To see if we can reach some kind of convergence with application rules, we have already obtained about fifty percent more test data for the `marking` test set. Handling this is will probably require ever more grammar and vocabulary rules to push PyElly processing capacity. PyElly has to prove here that it is not limited to running only small toy systems.

A new numbered version of PyElly will be released only after it passes all current unit and integration tests and the `pylint` tool has checked every modified Python file for common problems in source code. Any changes in how a particular release of PyElly works will also have to be described in an updated PyElly User's Manual; this should be done even for releases with no change in version number.

Appendix E. PyElly as a Educational Tool

PyElly approaches natural language processing (NLP) as a class of text data problems to be solved by developing descriptions of the structure of a language. It requires that a user be aware of syntax, semantics, morphology, and even phonology in the data—all aspects of classical linguistics. Such expertise is specialized, however, and the desire to avoid it has spurred the popularity of unsupervised statistical machine learning in NLP, which tries to infer the structure of a text data stream automatically.

Machine learning has been quite effective in many areas of NLP, especially automatic translation of text from one human language to another. It generally does make life easier for data scientists working only occasionally with text, but also has some limitations. A statistical model typically will be limited; for example, it may deal only with subsequences of n consecutive symbols for a small n, or operate combinatorially on K defined equivalence classes of symbols in text for a K of at most a few thousand.

Such simplified modeling will allow for handling of many significant NLP problems, but produces only a partial picture of language. To take full advantage of all the information available in digital text data, we want a machine to be able to read it almost as well as a human would. To do so, NLP in the end has to address issues of syntax, semantics, morphology, and phonology somehow; and much of that is difficult to achieve on a with simplified statistical models of data.

PyElly is old-fashioned and does no automatic machine learning. Instead, it relies on a user providing an organized set of language rules to guide proper rewriting of input text for a particular processing application. To use 1980's terminology, it is "rule-based." This approach can be challenging to carry out, but much about natural language has already been codified systematically by linguists in the past century and a half, and any educated speaker of a language also knows a great deal about its workings.

Even in our age of petaflop machines and clouds of virtual resources, hard-won linguistic knowledge can still benefit an NLP system when exploited right at the start of a project rather than being learned or relearned by a machine over the course of the project. Computational linguistics lets us be smart about language. In the longer run, if we are less savvy about natural language than automata trained on a corpus or two of data, then how can we hope to evaluate the processing done by the automata?

PyElly provides a framework in which to manage many different kinds of language definition rules in analyzing and rewriting text. We will have to dive into the details of a target language, but in return, we can see what is actually going on in our text analyses and gain flexibility in crafting processing strategies. Such practical experience is indispensable for someone trying to learn the ropes of NLP. After all, children need to learn multiplication by hand before becoming reliant on hand calculators all the time.

NLP can be really hard when we try to go much beyond keyword analysis. With PyElly, a student has to confront the difficulties in NLP head-on, but can choose to guide processing more actively when necessary. The issues are never buried in the entrails of

some deep neural network, where humans often are unable to relate to what it is actually being learned. Many people of course will be perfectly happy to work with black boxes, but at least a few of us need to be more curious here.

PyElly encourages students to play with language and facilitates the building of toy applications like pig Latin translation. Such simple NLP will of course be of little interest for academic research or for commercial exploitation, but it is excellent for learning by doing. It could conceivably even be fun.

A good way to employ PyElly educationally is to have students take on individual or group projects requiring only a few hundred rules at most. This would exclude something like the PyElly `marking` example application, but something like `chinese` or `texting` is quite doable. Avoid projects involving extensive background knowledge, as doing the research here could take over the whole project; for example, familiarity with Cold War Soviet military aircraft is needed to build the `querying` application.

As always, the best advice with PyElly is to start with something simple and gradually evolve to something more complex. Gratification may take a while. Students should expect to make many mistakes and to spend plenty of time in debugging, reworking, and clarifying their language descriptions. That is probably where most of the important NLP learning will happen anyway.