Seal Developer's Manual
Version 0.11

Steven Abney

July 2, 2015

Contents

Part I

Software Development

Chapter 1

Introduction

This manual describes the implementation and source-code organization of Seal, for those who might wish to modify it.

1.1 Use

1.1.1 Environment

The source directory for Seal is referred to as `$SRC`. It is usually named `seal`, and has the following toplevel listing:

```
Makefile      cx          doc          python
configure     data        examples     scripts
```

The directory into which Seal is installed is referred to as `$DEST`. It is usually `/cl`.

The examples assume the following environment variable settings:

- `PATH` includes `$DEST/bin`
- `PYTHONPATH` includes `$DEST/python`

The python examples assume that one has done:

```
>>> import seal
```

A first test:

```
>>> seal.hello()
Hello.  This is Seal ...
```

The actual output will have the version number in place of the ellipsis.

1.1.2 Installation

To install Seal, go to the `$SRC` directory and do:

```
$ make
```

There is a `configure` script, which writes configuration information into the file `CONFIG`. If `CONFIG` does not exist, `make` will call `./configure` to create it. Alternatively, one may call `./configure` manually. It takes one optional flag, the destination directory:

```
$ ./configure --prefix=/foo
```

1.2 seal

The `seal` module, defined in `$SRC/python/__init__.py`, does not contain any independent definitions, but imports a large number of definitions, giving the user access to them by doing

```
>>> from seal import *
```

1.3 seal.config

The `seal.config` module contains only four variables:

`Dest`	The destination directory in which Seal is installed (a string).
`Version`	The main version number (an integer).
`Revision`	The minor version number (an integer).
`Patchlevel`	The least-significant portion of the version number (an integer).

Chapter 2

Scripts

2.1 aconv

The script `aconv` takes a stream of bytes and prints out each byte as a number, or it does the reverse mapping. The script is called with two arguments, one of which is "`a`" and the other of which is a numeric base: 8, 10, or 16. For example:

```
$ aconv a 16 < /cl/examples/text1.utf8
0x66
0xc3
0xa1
0x20
...
0x8b
0xa
```

Note that C3 A1 is UTF-8 for U+00E1 (á).

Here is an example in the other direction:

```
$ echo '150 151 012' | aconv 8 a
hi
```

2.2 doctest

The script `doctest` provides a quick way to test python examples in Tex documentation. It creates a file suitable for the `doctest` module, and calls `doctest.testfile()` on it. Specifically, it extracts lines between "`\begin{python}`" and "`\end{python}`," and it extracts lines that begin "`%>>>`," omitting the leading percent sign. If the last extracted line is not "`>>>`," it inserts such a line as the last one.

For example, suppose the file `foo.tex` has the following contents:

```
\documentclass{article}
\begin{document}

This is an example.

\begin{python}
>>> 2 + 2
4
\end{python}
\end{document}
```

Calling `doctest` on it produces the following result:

```
$ doctest foo
1 test(s) found, all passed
```

If one changes the "4" to "5," the result is:

```
$ doctest foo
**********************************************************************
File ``/tmp/doctest.24869.test'', line 1, in doctest.24869.test
Failed example:
    2 + 2
Expected:
    5
Got:
    4
**********************************************************************
1 items had failures:
   1 of   1 in doctest.24869.test
***Test Failed*** 1 failures.
```

If the filename ends with "`.tex`," the suffix is stripped, so one can run doctest on all tex files in a directory by doing:

```
$ doctest *.tex
```

2.3 Collocations

2.3.1 Bigrams

The only trick with bigrams is that one should treat the text as "circular." That is, if the text consists of tokens $t_1, \ldots, t_n$, one should include not only the bigrams $(t_1, t_2), \ldots, (t_{n-1}, t_n)$, but also the "wrapped-around" bigram (t_n, t_1). That way, there are exactly as many bigrams as unigrams, and the distribution of unigrams in the first bigram position is the same as the distribution in the second bigram position.

Here is a simple script that produces bigrams from input containing one token per line.

```
#!/usr/bin/perl

$prev = '';
while ($token = <STDIN>) {
    chomp($token);
    if ($prev) { print("$prev\t$token\n"); }
    else { $first = $token; }
    $prev = $token;
}
print("$prev\t$first\n");
```

The script **bigrams** is similar, but it takes a **tokenized** file as input and produces **markup** containing bigrams as output.

The script **minskipgrams** takes the output of **mintok** and produces (x, y, d) triples, where x and y are words and d is the distance between them, with d being positive if they appear in the order $x \ldots y$, and d negative if they appear in the order $y \ldots x$. **Minskipgrams** has an obligatory argument, the maximum absolute value of d.

For example, create a file called `foo.txt` containing:

```
The rain in Spain stays mainly in the plain.
```

Then calling

```
mintok foo.txt | minskipgrams 2
```

creates output that begins:

```
in the -2
in rain -1
in spain 1
in stays 2
```

(Windows centered around the first two words are produced at the end of the file.)

When modelling bigram distributions, we pretend that the output of **minbigrams** or **minskipgrams** is an i.i.d. sample of word tuples. We assume random variables X, Y, and D returning the first word (reference word), the second word (target word), and distance from reference to target.

2.3.2 Counts

The script **counts** takes a **tabular file** as input. It interprets each line as a tuple, and produces counts for each of the distinct tuple types. Here is its contents:

```
#!/bin/sh

cat $* |
sort |
```

```
uniq -c |
fixuniq
```

(The Unix tool `uniq` prepends each line with leading whitespace followed by a number followed by a single space character; **fixuniq** replaces that with just the number followed by a tab.)

2.3.3 Pointwise mutual information

The formula is

$$\log \frac{p(x,y)}{p(x)p(y)}$$

With simple relative frequency estimates, that is:

$$\begin{aligned} & \log \frac{C(x,y)/N}{[C(x)/N] \cdot [C(y)/N]} \\ = \quad & \log N \frac{C(x,y)}{C(x)C(y)} \\ = \quad & \log N + \log C(x,y) - \log C(x) - \log C(y) \end{aligned}$$

The script **mi** takes a **counts** file as input and produces MI scores as output.

Here is an example:

```
capply mintok /cl/data/gut.corp |
bigrams |
counts |
mi |
sort -nr
```

2.3.4 Colloc

The **colloc** script can be used to compute scores using a variety of collocation measures for individual items with arbitrary count information. It takes a single argument, the name of a "command" file. The command file is a **tabular file** containing one "command" per line. The possible record types are:

description	`conc`	$d_{\min}$ $\text{count}_0 \ldots \text{count}_k$
description	`zscore`	$Nxy\ Nx\ Ny\ N$
description	`chi2`	$Nxy\ Nx\ Ny\ N$
description	`logbinratio`	$Nxy\ Nx\ Ny\ N$
description	`mi`	$Nxy\ Nx\ Ny\ N$
description	`zdiff`	$Nxy_1\ Nx_1\ Ny_1\ N_1\ Nxy_2\ Nx_2\ Ny_2\ N_2$
description	`rfratio`	$Nxy\ Nx\ Ny\ N$

Here is an example of a "command" file:

```
strong tea      conc    -2      0       1       0       8       3
strong tea      zscore  50      100     200     1000
strong tea      chi2    50      100     200     1000
strong tea      logbinratio     50      100     200     1000
strong tea      mi      50      100     200     1000
strong/powerful tea     zdiff   50      100     200     1000    15      150     200     1000
strong tea      rfratio 50      1000    10      500
```

Conc This is a measure of concentration. We have a sample of "displacements," giving the location of y with respect to x. For example, if x is "strong" and y is "tea," the displacement $+1$ indicates that we have the bigram "strong tea," the displacement -1 indicates that we have the bigram "tea strong," the displacement -2 indicates that we have "tea" before "strong" with exactly one word intervening, and so on. A minimum and maximum displacement is fixed. The variable d ranges over displacements and $C(d)$ is the number of tokens with displacement d.

We define:

$$n = \sum_d C(d)$$

$$\bar{d} = \frac{1}{n} \sum_d C(d) d$$

$$\texttt{conc} = \frac{1}{n-1} \sum_d C(d)(d - \bar{d})^2$$

For the remaining measures, we define random variables X and Y taking the value 1 for bigrams containing the target word in the appropriate position, and 0 otherwise. This yields a contingency table N with:

	$Y=0$	$Y=1$	
$X=0$	$N\bar{x}\bar{y}$	$N\bar{x}y$	$N\bar{x}$
$X=1$	$Nx\bar{y}$	Nxy	Nx
	$N\bar{y}$	Ny	N

z-score We define:

$$\begin{aligned} p &= \frac{Nx}{N} \cdot \frac{Ny}{N} \\ \mu &= Np \\ \sigma^2 &= Np(1-p) \\ \texttt{zscore} &= \frac{Nxy - \mu}{\sigma} \end{aligned}$$

z-score of differences We are given two bigrams and we are interested in their relative strength of affinity.

$$\begin{aligned} p_1 &= \frac{Nx_1}{N_1} \cdot \frac{Ny_1}{N_1} \\ p_2 &= \frac{Nx_2}{N_2} \cdot \frac{Ny_2}{N_2} \\ \sigma_1^2 &= N_1 p_1 (1 - p_1) \\ \sigma_2^2 &= N_2 p_2 (1 - p_2) \\ \texttt{zdiff} &= \frac{Nxy_1 - Nxy_2}{\sqrt{\sigma_1^2 + \sigma_2^2}} \end{aligned}$$

χ^2 test A contingency table E of expected counts is constructed, with

$$E\alpha\beta = \frac{N\alpha}{N} \cdot \frac{N\beta}{N} \cdot N = \frac{N\alpha \cdot N\beta}{N}$$

where $\alpha \in \{x, \bar{x}\}$ and $\beta \in \{y, \bar{y}\}$. Then

$$\texttt{chi2} = \sum_{\alpha,\beta} \frac{(N\alpha\beta - E\alpha\beta)^2}{E\alpha\beta}$$

Likelihood ratio of binomials For a given bigram (x, y):

$$p_0 = p(y) = \frac{Ny}{N} \qquad p_1 = p(y|x) = \frac{Nxy}{Nx} \qquad p_2 = p(y|\bar{x}_1) = \frac{N\bar{x}y}{N\bar{x}}$$

$$b(k; n, p) = \binom{n}{k} p^k (1 - p)^{n-k}$$

$$\texttt{logbinratio} = \log \frac{b(Nxy; Nx, p_0) b(N\bar{x}y; N\bar{x}, p_0)}{b(Nxy; Nx, p_1) b(N\bar{x}y; N\bar{x}, p_2)}$$

Relative frequency ratio Let Nxy_1 be the count of (x, y) in Corpus-1 (size N_1), and let Nxy_2 be its count in Corpus-2 (size N_2).

$$\texttt{rfratio} = \frac{Nxy_1/N_1}{Nxy_2/N_2}$$

Pointwise mutual information For a given bigram (x, y):

$$p(x, y) = \frac{Nxy}{N} \qquad p(x) = \frac{Nx}{N} \qquad p(y) = \frac{Ny}{N}$$

$$\texttt{mi} = \log \frac{p(x, y)}{p(x)p(y)}$$

2.4 Xmltxt

The executable `xmltxt` converts XML to plaintext. It suppresses nodes whose category is `head`, `script`, or `style`. It takes zero or more filenames, and prints its output to stdout.

```
$ xmltxt /cl/data/bible/rsv/01-genesis.html > gen.txt
```

Chapter 3

General purpose: `seal.misc`

The module `seal.misc` contains a collection of generally useful functions. For this chapter, we assume the following includes:

```
>>> from seal.misc import *
>>> from seal.io import ex
```

3.1 General

3.1.1 `hello`

We have already seen the function `hello()`, which just prints version information.

```
>>> hello()
Hello.  This is Seal ...
```

3.1.2 `mean`

The function `mean()` returns the arithmetic mean of two numbers.

```
>>> mean(2,4)
3.0
```

3.1.3 `matches`

The function `matches()` takes an object and a dict. The dict is interpreted as a description, in which the keys are attributes and the values are required values. The return value is `True` or `False`, indicating whether the object matches the description.

```
>>> class Point (object):
...     def __init__ (self, x, y):
```

```
...         self.x = x
...         self.y = y
...     def L1_norm (self):
...         return abs(self.x) + abs(self.y)
...
>>> p = Point(2, -4)
>>> matches(p, {'y': -4, 'x': 2})
True
>>> matches(p, {'y': 0})
False
>>> matches(p, {'foo': 'bar'})
False
```

If a value specification is a list, then the actual value can be any member of the list.

```
>>> matches(p, {'x': [0,1,2]})
True
```

If the named attribute is a method, then it is called to get the value that is compared to the description's value.

```
>>> matches(p, {'L1_norm': 6})
True
```

A `None` in the description functions as a wildcard. It matches any object.

```
>>> matches(p, {'foo': None})
True
```

3.1.4 Index

An `Index` is a dict that associates multiple values (a list) with each key. For example:

```
>>> index = Index()
>>> index['hi']
[]
>>> index.add('hi', 10)
>>> index['hi']
[10]
>>> index.add('hi', 42)
>>> index['hi']
[10, 42]
```

The method `count()` returns the number of items for a given key:

```
>>> index.count('hi')
2
```

The method `values()` returns the concatenation of all the lists.

```
>>> index.add('bye', 20)
>>> sorted(index.values())
[10, 20, 42]
```

An iterator is available as `itervalues()`.

One can also delete items:

```
>>> index.delete('hi', 10)
>>> index['hi']
[42]
```

3.2 Strings

3.2.1 Decimal-hex conversion

To convert hex to decimal:

```
>>> int('03bb', 16)
955
```

To convert decimal to hex:

```
>>> hex(955)
'0x3bb'
```

3.2.2 Unicode characters

To get the Unicode character corresponding to a given code point:

```
>>> chr(0x0041)
'A'
```

To convert a code point to a string:

```
>>> ord('z')
122
>>> hex(_)
'0x7a'
```

To get the name of a Unicode character:

```
>>> import unicodedata
>>> unicodedata.name(chr(0x00e1))
'LATIN SMALL LETTER A WITH ACUTE'
```

One can also go in the reverse direction, but it is necessary to know the precise name of the character:

```
>>> aacute = unicodedata.lookup('latin small letter a with acute')
>>> hex(ord(aacute))
'0xe1'
```

3.2.3 UTF-8 conversion

One can convert into and out of UTF-8 as follows:

```
>>> koppa = b'\xd2\x80'.decode('utf8')
>>> hex(ord(koppa))
'0x480'
>>> koppa.encode('utf8')
b'\xd2\x80'
```

(The code point U+0480 is a Cyrillic character that looks like a sickle, or a C in which the bottom curve is bent straight down. Its name is Cyrillic Capital Letter Koppa.)

3.2.4 as_ascii

The function `as_ascii()` returns a string containing only ASCII characters. If the input is a regular string, it returns it unmodified. If the input contains non-ASCII characters, they are replaced with escape sequences, surrounded by braces.

```
>>> as_ascii('h\u00ff')
'h{ff}'
```

If one adds `use_names=True`, non-ASCII unicode characters are instead handled by inserting the character name (if available) or code, in braces.

```
>>> as_ascii('h\u00ff', use_names=True)
'h{LATIN SMALL LETTER Y WITH DIAERESIS}'
```

If the input is not a string, `as_ascii()` calls `str()` on it.

```
>>> as_ascii(10)
'10'
```

3.2.5 deaccent

The function `deaccent` converts a Unicode string to ASCII in a lossy way. It replaces characters in the Latin-1 range with corresponding ASCII characters, where natural correspondences exist. Characters without a natural ASCII counterpart are simply deleted. ASCII control characters other than space, tab, newline, and carriage return are deleted. The return value is an ASCII string.

3.2.6 as_boolean

The function `as_boolean()` converts the strings `'True'` and `'False'` to the corresponding boolean values. Given anything else, it signals an error.

```
>>> as_boolean('False')
False
```

3.2.7 trim

The function `trim()` takes two arguments: a field width and a string. It first calls `as_ascii()` on the string, and then it truncates it at the field width.

```
>>> trim(6, u'L\u00ffcra')
'L{ff}c'
```

3.3 Lists

The functions `as_list()`, `repeatable()`, `concat()`, `unique()`, and `cross_product()` produce lists as output.

3.3.1 as_list

The function `as_list()` converts any item x to a list.

- If x is `None`, it returns the empty list.
- If x is a list, it returns x itself.
- If x is a sequence, it returns `list(`x`)`.
- If x is has attribute "`next`," it takes x to be a generator and returns `list(`x`)`.
- Otherwise, it returns `[`x`]`.

3.3.2 repeatable

A generator can only be used once, whereas iterables such as lists, tuples, sets, and dicts can be iterated over multiple times. The function `repeatable()` converts a generator into a list, but leaves other iterables alone. It coerces `None` to the empty list, but otherwise signals an error if its input is not an iterable. It assumes that any object with a `next` attribute is a generator, and any object with an `__iter__` attribute is an iterable.

3.3.3 concat

The function `concat()` takes a single argument, a list of lists, and returns their concatenation. For example:

```
>>> concat(line.split() for line in open(ex.text1))
['This', 'is', 'a', 'test.', 'It', 'is', 'only', 'a', 'test.']
```

This returns a list containing all (space-separated) tokens from all lines of the file. (Of course, a simpler way to do it would be to use the `read` method of the file to turn it into a string, and split that.)

A related function is `itertools.chain()`, which takes multiple iterables as input (multiple arguments), and produces a generator as output.

3.3.4 unique

The function `unique()` takes a list as input and produces a list with all duplicates removed. The list does not need to be sorted, nor do duplicates need to be adjacent to each other. The algorithm is naive (quadratic), so it is only appropriate for short lists.

```
>>> unique([4, 2, 4, 1, 2])
[4, 2, 1]
```

3.3.5 cross_product

The function `cross_product()` takes a single argument, a list of lists, and produces the cross product of those lists as output.

```
>>> cross_product([['a', 'b'], [1, 2], [42]])
[('a', 1, 42), ('a', 2, 42), ('b', 1, 42), ('b', 2, 42)]
```

3.3.6 Sorted lists

The functions `intersect`, `union`, and `difference` expect sorted lists as input. Their behavior is unpredictable if they are given unsorted lists.

```
>>> x = [1,3,5,6,7]
>>> y = [2,3,4,7,8]
>>> intersect(x,y)
[3, 7]
>>> union(x,y)
[1, 2, 3, 4, 5, 6, 7, 8]
>>> difference(x,y)
[1, 5, 6]
>>> difference(y,x)
[2, 4, 8]
```

3.3.7 Queue

A `Queue` is a first-in first-out queue. The method `write()` inserts an element at the tail of the queue, and `read()` removes and returns the element at the head of the queue.

It is implemented as a buffer with head and tail pointers. Initially the buffer is empty. If the tail is at the end of the buffer, new elements are appended to the buffer and the buffer grows. When the queue is empty, the head and tail are reset to 0.

Space in the buffer before the head is "wasted" space. If the wasted space exceeds a threshold (`maxwaste`), the contents of the queue are relocated so that the head is 0. One can specify `maxwaste` when creating the queue; by default it is 10. Setting `maxwaste` to `None` prevents the contents from being relocated (though the head and tail will still be reset to 0 if the queue becomes empty).

The elements in the queue can be accessed and set by index.

3.4 Generators

The following functions are provided that relate to generators: `chain()`, `nth()`, `head()`, `tail()`, `more()`, `product()`, `count()`, and `counts()`.

For the purpose of illustration, let us define a little generator:

```
>>> def pots ():
...     for i in range(11):
...         yield 2**i
...
>>> type(pots())
<class 'generator'>
>>> list(pots())
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

3.4.1 `chain`

The function `chain()` is imported from `itertools`. It is the generator equivalent of `concat`.

3.4.2 `nth`

The function `nth()` returns a particular item from an iterator.

```
>>> nth(pots(), 2)
4
```

Remember that an iterable is consumed as one iterates through it. In the example just given, we created a new generator by calling `pots()`. If we use its value, though, we need to be careful:

```
>>> iter = pots()
>>> nth(iter, 2)
4
>>> list(iter)
[8, 16, 32, 64, 128, 256, 512, 1024]
```

Note that `nth()` consumed the first three items.

Incidentally, one can achieve the same functionality this way, though more awkwardly:

```
>>> from itertools import islice
>>> next(islice(pots(), 2, None))
4
```

One use of `nth` is to jump to problematic cases in a large iteration. An idiom for finding such cases in the first place is the following:

```
for i, x in enumerate(myiteration):
    if isproblematic(x):
        return i
```

3.4.3 head, tail

The functions `head()` and `tail()` are also provided for inspecting parts of a large iterable.

```
>>> head(pots())
[1, 2, 4, 8, 16]
>>> tail(pots())
[64, 128, 256, 512, 1024]
```

An optional argument specifies how many items one would like to have:

```
>>> head(pots(), 3)
[1, 2, 4]
>>> tail(pots(), 3)
[256, 512, 1024]
```

A more general function is `islice`, from the standard `itertools` module:

```
>>> list(islice(pots(), 2, 5))
[4, 8, 16]
```

3.4.4 more

The function `more()` calls `print` on each item in turn, pausing after a pageful of items has been displayed. Hitting return causes another page to be displayed, and hitting 'q' then enter causes `more()` to quit.

One can adjust the pagesize by setting `more.pagesize`. For example:

```
>>> more.pagesize = 4
>>> more(pots())
1
2
4
8
q
```

3.4.5 product

The function `product()` is analogous to `sum()`. It takes an iterable containing numbers, and returns the product of the numbers.

3.4.6 count

The function `count()` is analogous to `len()`, except that it works for generators as well as lists and other iterables.

```
>>> count(pots())
11
```

Note that `count` is unrelated to `itertools.count()`. The latter returns an infinite iterator that generates the natural numbers.

3.4.7 counts

The function `counts()` creates a table of counts of occurrences.

```
>>> tab = counts('abracadabra')
>>> sorted(tab.items())
[('a', 5), ('b', 2), ('c', 1), ('d', 1), ('r', 2)]
```

3.5 System

The functions described in this section are likely to go away. The standard function `os.system()` is generally more convenient.

3.5.1 call

The function `call()` makes a synchronous system call. It signals an error if the system call returns an error. It returns no value.

```
>>> call('ls', '..')
```

3.5.2 launch

The function `launch()` makes an asynchronous system call. It returns 0 on success and an error code on failure.

3.5.3 run_main

The function `run_main` takes a function *main* and passes `sys.argv` to it.

There are a number of ambiguities in the usual conventions. For example, in an invocation like `foo -t 10`, it is impossible to know whether `-t` is a boolean flag and `10` is a positional argument, or if `-t` has value `10` and there are no positional arguments. Again, in `foo -bt` it is impossible to tell whether the flag is `-bt` or two flags `-b` and `-t` have been combined.

The conventions of `run_main` are as follows. Flags are never combined; one cannot abbreviate `-b -t` as `-bt`. Flag arguments are always attached to the flag with an `=` as connector. For example, `-bt=abc` represents flag `-bt` with value `'abc'`.

The contents of `sys.argv` is parsed into a list of positional arguments `args` and a flag-argument dict `kwargs`. Then the function f is called as f`(*args, *kwargs)`. Flags must precede positional arguments. The special token `--` terminates the flags; it makes it possible to supply positional arguments that begin with `'-'`. A flag token always begins with `'-'`. The portion from character 1 up to the first occurrence of `'='` is the keyword, and the remainder (following the `'='`) is the value. If there is no `'='` then the value is `'1'`. That is, the token `-t` is exactly equivalent to `-t=1`. It is an error if the same flag is provided multiple times.

All arguments are strings. One can always define a helper function that parses the arguments and hands them off to the real function, e.g.:

```
def main (x, trace=False):
    foo(float(x), bool(trace))
```

The function `run_main()` catches exceptions. If the main function returns a value, the value is printed and `sys.exit(0)` is called. If the main function throws an exception, it is caught and printed to stderr, and `sys.exit(1)` is called.

The special flag `-?` causes a usage message to be printed. It lists the positional arguments and flags, and prints the doc string (if any). (Any other flags or arguments are ignored if `-?` is present.)

The special flag `-!` causes a stack trace to be printed if an exception is thrown, instead of the usual brief error printing.

3.5.4 CommandLine

An example of code using `CommandLine`:

```
args = CommandLine('[-m foo] fn*')
foo = None
while args.has_option():
    key = args.option()
    if key == '-m': foo = args.next()
    else: args.usage()
fns = args
```

After the options have been consumed, *args* behaves like a list of the remaining arguments.

3.5.5 Timing

One can create a timer:

```
>>> timer = Timer()
```

Every time one calls `str()` on it, one obtains a printed version of the elapsed time since it was created.

```
>>> print timer
0:00:03.316634
```

The function that `Timer` uses for printing is separately available. It takes two numbers representing start time and end time in seconds.

```
>>> elapsed_time_str(10, 135)
'0:02:05.0000'
```

3.5.6 Progress indicator

To create a progress indicator:

```
>>> progress = Progress(10)
```

The value n is the total number of "work units" that will be necessary. To cause a progress message to be printed, increment the indicator:

```
>>> progress += 1
Progress: 10.00% Time remaining: 24.097824
```

3.5.7 XTerm escapes

The functions `red()` and `green()` set the foreground color for their argument:

```
>>> print red('hi'), green('bye')
```

The function `repln()` causes its argument to replace the contents of the current line. (It does a carriage return and line kill.)

```
>>> print 'hi there',
>>> print repln('bye')
```

Alternatively:

```
>>> print repln(),
>>> print 'bye'
```

Chapter 4

Shell commands: `seal.sh`

The module `seal.sh` contains functions that provide an approximation to a shell.

```
>>> from seal.sh import *
```

4.1 Environment variables: `echo`, `setenv`

```
>>> setenv('FOO', 'Hello world')
>>> echo('$FOO')
Hello world
>>> os.environ.get('FOO')
'Hello world'
```

4.2 File system

4.2.1 Creating directories: `mkdir`, `mkdirp`

`mkdir` is imported from `os`. `mkdirp` is equivalent to `mkdir -p`.

```
>>> mkdirp('/tmp/foo')
```

The function `need_parent` can be used to assure that the parent directory exists for a filename. For example,

```
>>> need_parent('/tmp/foo/myfile')
```

is the same as `mkdirp /tmp/foo`.

4.2.2 Creating files: `touch`, `echo`, `cat`

One can use `touch` to create a new file.

```
>>> cd('/tmp/foo')
>>> touch('bar')
```

The function `echo` takes an optional second argument which is a filename. By default, the file is appended to, leaving any previous contents intact.

```
>>> echo('hi', 'greet')
>>> echo('lo', 'greet')
>>> cat('greet')
hi
lo
```

One can prefix the filename with "`>`" or "`>>`" to explicitly specify overwriting versus appending.

```
>>> echo('boo', '>bar')
>>> cat('bar')
boo
```

The function `cat` behaves similarly. With a single argument, it prints to stdout, and given multiple arguments, it takes the last as the output file name. By default, the output file is created, but one may specify appending by prefixing the filename with "`>>`."

```
>>> cat('greet', 'bar', 'baz')
>>> cat('baz')
hi
lo
boo
>>> cat('bar', 'bar', '>1')
boo
boo
```

4.2.3 Examining files: `more`, `od`, `wc`

The function `cat` can also be used, of course. The other functions (`more`, `od`, `wc`) simply use `os.system()` to call the Unix executables. The type of datum for `od` can be specified using the `type` keyword. Possible values are: `a` (named characters); `c`; *is* where *i* is one of `d`, `o`, `u`, `x` and *s* (optional) is one of `C`, `S`, `I`, `L`, or a number of bytes; or `f` followed optionally by `F`, `D`, or `L`.

```
>>> od('bar')
0000000    b   o   o  \n
0000004
>>> od('bar', 'xC')
0000000    62  6f  6f  0a
0000004
>>> echo('this is a test', 'text')
>>> echo('it is only a test', 'text')
```

```
>>> wc('text')
       2       9      33 text
```

4.2.4 Testing and filename manipulation

The following functions are imported from `os.path`: `isfile`, `isdir`, `islink`, `isabs`, `exists`, `basename`, `dirname`.

4.2.5 Navigation: `pwd`, `cd`, `ls`

```
>>> cd('/tmp/foo')
>>> ls()
bar     baz     greet   text
```

On my machine, `/tmp` is a symbolic link to `/private/tmp`.

```
>>> pwd()
'/private/tmp/foo'
```

The function `lsl` does a long listing.

```
>>> lsl()
total 16
-rw-r--r--  1 spa  wheel  4 Oct 14 17:35 bar
-rw-r--r--  1 spa  wheel  8 Oct 14 17:41 baz
```

The variations `lsd`, `lsld`, and `lslt` are also available.

4.2.6 Copying: `cp`, `mv`, `ln`

```
>>> cp('bar', 'bar2')
>>> ls()
bar     bar2    baz
>>> mv('bar2', 'bar3')
>>> ls()
bar     bar3    baz
>>> ln('bar3', 'bar4')
>>> cat('bar4')
boo
```

Note that `ln` creates a symbolic link, not a hard link. To create a hard link, use `link`. (Both are imported from `os`. In `os`, `ln` is called `symlink`.)

```
>>> lsl()
total 32
-rw-r--r--  1 spa  wheel  4 Oct 14 17:35 bar
-rw-r--r--  1 spa  wheel  4 Oct 14 19:21 bar3
lrwxr-xr-x  1 spa  wheel  4 Oct 14 19:27 bar4 -> bar3
-rw-r--r--  1 spa  wheel  8 Oct 14 17:41 baz
```

4.2.7 Deletion: rm, rmrf, rmdir

```
>>> rm('bar4')
>>> ls()
bar     bar3    baz
>>> cd('..')
>>> lsd('foo')
foo
>>> rmrf('foo')
>>> lsd('foo')
ls: foo: No such file or directory
```

4.3 Misc: sh, pid, launch

The functions `sh` and `pid` are just synonyms for `os.system` and `os.getpid`. The function `launch` calls the executable `open`, which is Mac-specific.

Chapter 5

Input/Output: `seal.io`

The `seal.io` module contains functionality related to files and directories.

The examples in this chapter assume the following imports:

```
>>> import seal
>>> from seal.io import *
>>> from seal.sh import ls, od
```

5.1 Contents, tee, null, `OutputList`, output string

5.1.1 `contents`

The function `contents()` returns the raw contents of a file.

```
>>> contents(ex.text1)
'This is a test.\nIt is only a test.\n'
```

5.1.2 `tee`

The class `tee` is a file-like object that sends everything that is written to it both to a file and to stdout.

```
>>> f = tee('/tmp/foo')
>>> print('Hello', file=f)
Hello
>>> close(f)
>>> contents('/tmp/foo')
'Hello\n'
```

5.1.3 `null`

The object `null` can be used as a null stream.

```
>>> print('Hello', file=null)
>>>
```

5.1.4 OutputList

An `OutputList` is a specialization of `list` that behaves like an output stream. That is, it implements a `write()` method. Strings not ending in newline constitute partial lines. They are accumulated until a string ending with newline is written, at which point all partial lines to that point are concatenated, and the resulting line is appended to the list. Trailing carriage returns and newlines are deleted.

Here is an example:

```
>>> from seal.io import OutputList
>>> output = OutputList()
>>> print('Hello', [10,20], file=output)
>>> print('Bye', file=output)
>>> output
['Hello [10, 20]', 'Bye']
>>> output[0]
'Hello [10, 20]'
```

Two cautions are in order. (1) Embedded newlines are not detected. (2) If the last thing written to the list did not end in newline, it will not appear in the list. It can, however, be accessed as `output.partial`.

5.1.5 Output string

Python already provides `StringIO`, which is an output stream that produces a string. It has been imported into `seal.io` for convenience:

```
>>> output = StringIO()
>>> print('Hello world.', file=output)
>>> print('Bye.', file=output)
>>> output.getvalue()
'Hello world.\nBye.\n'
>>> output.close()  # releases string buffer
```

5.1.6 Input from string

`StringIO` can also be used to create an input stream:

```
>>> list(StringIO('This is a test\nIt is only a test\n'))
['This is a test\n', 'It is only a test\n']
```

5.2 Filenames

5.2.1 Suffixes

The function `strip_suffix()` takes a filename and strips the suffix, if any. A suffix must begin with dot (`.`) and it cannot contain either dot or slash (`/`).

The function `split_suffix()` takes a filename and returns a pair (f, s) where f is the filename without the suffix (if any), and s is the suffix (including the dot). If there is no suffix, s is the empty string.

5.2.2 `Fn`

The class `Fn` is a specialization of string that is used to represent filenames. There are two advantages. First, in some cases we may wish to treat strings as content, but filenames as locations where content is stored; that is not possible unless there is a type distinction between strings and filenames. Second, it is convenient to be able to use the dot operator to construct pathnames. If a filename denotes an existing directory, the dot operator selects a file or subdirectory within it. Otherwise, the dot operator represents a filename suffix. For example:

```
>>> from seal.config import relpath
>>> dest = Fn(seal.config.Dest)
>>> relpath(dest.examples)
'examples'
>>> relpath(dest.examples.foo.bar)
'examples/foo.bar'
```

(The function `relpath()` converts an absolute pathname into a pathname relative to the Seal destination directory.)

Adding a string to a `Fn` results in a new `Fn`:

```
>>> Fn('foo') + 'bar'
'foobar'
>>> type(_)
<class 'seal.io.Fn'>
```

The division slash is used as directory separator; it calls `os.path.join()`.

```
>>> Fn('foo') / 'bar'
'foo/bar'
```

The class `Fn` also provides a some methods for convenience.

- `exists()` returns true just in case the filename names an existing file or directory.
- `isdir()` returns true just in case the filename names an existing directory.
- `parent()` returns the parent directory. For example:

```
>>> fn = Fn('/tmp/foo/')
>>> fn.parent()
'/tmp'
>>> _.parent()
'/'
>>> _.parent()
>>>
```

- `create()` creates a new file or directory. It takes the filename to name a directory if it ends with slash, or if `dir=True` is provided.

- `list()` returns the output of `stat()`, if this is a regular file, and it returns a list of filenames, if this is a directory. The filenames are of type `Fn`; they are unsorted; and they do not include the `.` or `..` directories.

- `rename()` changes the name of an existing file or directory.

- `copy()` creates a new copy of an existing file or directory. In the case of a directory, it does a recursive copy.

- `delete()` deletes an existing file or directory. In the case of a directory, it signals an error if the directory is not empty.

Note that these method names take precedence over the use of the dot operator as a path extender. For example, to refer to the file "`/tmp/list`," one cannot use `tmp.list`; one must instead use `tmp/'list'`.

5.2.3 Directories `dest`, `ex`, etc.

The variable `ex` names the examples directory `$DEST/examples`. It is a `Fn`. For example:

```
>>> relpath(ex.t1.t)
'examples/t1.t'
```

There are several variables that name directories, as follows:

`bin`	`$DEST/bin`
`data`	`$DEST/data`
`dest`	Fn version of `seal.config.Dest`
`ex`	`$DEST/examples`
`here`	The current working directory.
`home`	The user's home directory.
`root`	The root directory, `/`
`tmp`	`/tmp`

5.2.4 tmpfile

The function `tmpfile()` allocates a temporary filename prefix and arranges for any file whose name begins with that prefix to be deleted (if any such exist) when the *filename object* returned by `tmpfile()` is deleted. The filename object may be explicitly deleted using `del`, or deleted by the garbage collector if there are no remaining references to it, or deleted when the program exists.

The filename is of form:

`/tmp/seal_`*pid*`_`*n*

where *pid* is the pid of the python process and *n* is a sequence number that is advanced each time a temporary filename is allocated.

Here is an example. First, let us assure ourselves that the file does not already exist:

```
>>> ls('/tmp/seal*')
ls: /tmp/seal*: No such file or directory
```

Now we allocate the filename:

```
>>> fn = tmpfile()
```

This does not yet create a file:

```
>>> ls('/tmp/seal*')
ls: /tmp/seal*: No such file or directory
```

Now let us create two files:

```
>>> save_string('hi there\n', fn.a)
>>> save_string('bye now\n', fn.b)
```

We confirm that they were created:

```
>>> ls('/tmp/seal*')
/tmp/seal_30278_1.a    /tmp/seal_30278_1.b
```

Now we delete the filename.

```
>>> del fn
```

This also deletes the files:

```
>>> ls('/tmp/seal*')
ls: /tmp/seal*: No such file or directory
```

The fact that the files are deleted when the filename is deleted means that one must be sure not to lose the last reference to the filename while one is still using the files. However, the fact that `file` objects store the filename used to open them means that it is difficult to lose the last reference prematurely.

```
>>> f = open(tmpfile(), 'w')
>>> f.name
'/tmp/seal_94087_2'
>>> type(f.name)
<class 'seal.io._TmpFn'>
```

5.3 Infiles and outfiles

5.3.1 infile

The function `infile()` returns an input stream.

```
>>> [as_ascii(line) for line in infile(ex.text1.utf8)]
['f{e1} f{e1}{newline}', 'ki{014b} ko{014b}{newline}']
```

Note that U+E1 is *a* with an acute, and U+014B is engma:

```
>>> import unicodedata
>>> unicodedata.name('\u00e1')
'LATIN SMALL LETTER A WITH ACUTE'
>>> unicodedata.name('\u014b')
'LATIN SMALL LETTER ENG'
```

In addition to accepting a string as filename, some cases are treated specially:

- If the argument is `'-'`, then the return value is `sys.stdin`.
- If the argument begins with letters (non-empty, only alphabetic) followed by a colon, it is interpreted as a URL.
- If the argument is an open file whose mode begins with `'r'`, or a `StringIO` instance, or an object with a `readline()` method, it is passed through.

Note that `ex` and its extensions, such as `ex.text1`, are of type `Fn`, which is a subclass of `str`.

To provide a string as contents, rather than filename, wrap it in `StringIO`:

```
>>> list(infile(StringIO('This is a test.\nOnly a test.\n')))
['This is a test.\n', 'Only a test.\n']
```

5.3.2 outfile

The function `outfile()` returns an output file.

```
>>> fn = tmpfile()
>>> f = outfile(fn)
>>> print('Hello', file=f)
>>> close(f)
>>> contents(fn)
'Hello\n'
```

Regarding the argument to `outfile()`, there are again some cases that are treated specially:

- The filename `Fn('-')` represents `sys.stdout`.
- If the argument is omitted or is `None`, output is accumulated as a string, which can be retrieved using `getvalue()`.

```
>>> f = outfile()
>>> f.write('hi there\n')
9
>>> f.write('bye\n')
4
>>> f.getvalue()
'hi there\nbye\n'
```

5.3.3 close

Caution: code that calls `outfile()` should *not* call the file's `close()` method. If the user passes "`--`" as input, closing the file will close stdout. Instead use the function `close()`, which closes the file unless it is, or embeds, stdout.

```
>>> close(f)
```

5.4 Load and save functions

5.4.1 General

There is a series of paired "load" and "save" functions for different kinds of contents. They build on unicode input and output streams, and inherit the same conventions regarding their filename arguments.

Where it makes sense, there is also an "iter" function corresponding to each "load" function. The "iter" function returns a generator, and the "load" function returns a list. However, there is no "iter" function corresponding to `load_string()` or `load_dict()`.

Close unicode. The definitions of the "save" functions all have a similar outline:

```
def save_x (x, filename=None):
    f = outfile(filename)
    ...
    return close(f)
```

The function `close_unicode()` will close the file *unless* it is `sys.stdout`. If the file was created with no filename, `close_unicode()` gets the string contents before closing the file, and its return value is the string contents. Otherwise, the return value is `None`.

5.4.2 Strings

Load string. The function `load_string()` returns the entire contents of a file as a unicode string.

```
>>> load_string(ex.text1)
'This is a test.\nIt is only a test.\n'
```

Save string. The companion function `save_string()` does the opposite:

```
>>> fn = tmpfile()
>>> save_string('f\u00e1\n', fn)
```

5.4.3 Lines

Load lines. The function `load_lines()` returns the lines of a file, *without* the trailing newline characters.

```
>>> load_lines(ex.text1)
['This is a test.', 'It is only a test.']
```

There is also a function `iter_lines()` which returns a generator instead of a list.

Save lines. The function `save_lines()` takes an iterator over strings. Each becomes a line of the file. Newline characters are added.

```
>>> fn = tmpfile()
>>> save_lines(['foo', 'f\u00e1'], fn)
```

One can then confirm the contents:

```
>>> [as_ascii(line) for line in infile(fn)]
['foo{newline}', 'f{e1}{newline}']
```

5.4.4 Records

A **record** is a list (more generally, a sequence) of strings representing field values. On disk, each record is a line and field values are separated by tabs. A file containing such records is a **tabular file**.

Load records. The function `load_records()` takes a filename and returns a list of records, representing the contents of the file.

```
>>> load_records(ex.tab1.tab)
[['foo', '42'], ['bar', '15']]
```

Optionally, one can specify the field separator by providing the keyword argument `separator`. The default separator is tab. A value of `None` means that any amount of whitespace constitutes a separator, and leading and trailing whitespace are ignored.

Iter records. There is also a function `iter_records()` that returns a generator instead of a list. It takes the same `separator` argument as `load_records()` does. In addition to the method `next()`, which all generators support, the `iter_records()` generator also supports the method `error()`, which takes an an error message and signals an error, indicating the filename and line number of the most recently read record.

Save records. The function `save_records()` takes an iterator over records and writes them to a file.

```
>>> recs = [('1', 'hi'), ('2', 'lo'), ('3', 'bye')]
>>> fn = tmpfile()
>>> save_records(recs, fn)
>>> load_records(fn)
[['1', 'hi'], ['2', 'lo'], ['3', 'bye']]
```

One can optionally specify the `separator`.

5.4.5 Dict

A dict is represented on disk as a tabular file with two columns: key and value.

Load dict. The function `load_dict()` reads a dict from a tabular file. If there are duplicate keys in the file, only the last copy has any effect: earlier values get overwritten.

```
>>> d = load_dict(ex.tab1.tab)
>>> sorted(d)
['bar', 'foo']
>>> d['foo']
'42'
```

Save dict. The function `save_dict()` takes a dict and writes it to a file. Keys and values must all be strings.

5.4.6 Nested dict

A nested dict is specified with dotted keys and values. One or more whitespace characters serve as separator between key and value. For example, the following is the contents of `ex.nivre.exp`:

```
command seal.dp.nivre
dataset spa.orig
features nivre-2007
nulls True
split.feature fpos.input.0
split.cpt.s 0
split.cpt.t 1
split.cpt.d 2
split.cpt.g 0.2
split.cpt.c 0.5
split.cpt.r 0
split.cpt.e 1.0
```

The function `load_nested_dict()` creates a dict in which the keys are `'command'`, `'dataset'`, `'features'`, `'nulls'`, and `'split'`. The value for `'split'` is a subdict with keys `'feature'` and `'cpt'`, and within the subdict, the value for `'cpt'` is a sub-subdict.

5.4.7 Paragraphs

A paragraph is a maximal block of lines not containing an empty line.

Load paragraphs. The function `load_paragraphs()` reads a file and returns a list of paragraphs.

```
>>> load_paragraphs(ex.par1.txt)
['This is\na test.\n', 'It is only\na test.\n']
```

Save paragraphs. The function `save_paragraphs()` takes an iterator over paragraphs and writes each to the named file. An empty line is written as a separator before each paragraph except the first.

5.4.8 Blocks

A block is a contiguous sequence of non-empty lines. Separators between blocks consist of one or more empty lines. A block is represented as a list of lines; carriage return and newline are stripped from the lines.

Load blocks. The function `iter_blocks()` reads a file and generates a sequence of blocks. The function `load_blocks()` converts the generator to a list.

```
>>> load_blocks(ex.par1.txt)
[['This is', 'a test.'], ['It is only', 'a test.']]
```

Save blocks. The function `save_blocks()` takes an iterator over blocks (lists of lists of strings) and writes each to the named file. An empty line is written as separator between each pair of blocks.

5.4.9 Record blocks

A record block is a contiguous sequence of non-empty records. One or more empty records (i.e., empty lines) separate record blocks. A record block is represented as a list of lists, each record being a list of fields (strings).

5.5 Tokens

Files that contain something comparable to code—for example, grammar files or files containing predicate-calculus expressions—are treated as sequences of tokens.

5.5.1 Load, Iterate, Tokenize

Load tokens. The function `load_tokens` interprets a file (or string) as a list of tokens. The token definition is kept intentionally simple: quoted strings are recognized, the delimiters `()[]{}` are recognized as special characters, unquoted space separates tokens, and `#` begins a comment.

```
>>> print(load_string(ex.tok1), end='')
12 + foo(bar=42.0, baz="hi there")
>>> load_tokens(ex.tok1)
['12', '+', 'foo', '(', 'bar=42.0,', 'baz=', 'hi there', ')']
```

The class `Token` is a specialization of `str`. It has an additional attribute `type` whose value is `'word'`, `'eof'`, or one of the six delimiter characters `()[]{}`. No token whose type is `'eof'` is ever returned by the tokenizer, but it is used as an end-of-file sentinel. Functions that test for types can also use the pseudo-type `'any'` which matches anything except `'eof'`.

Quoted strings are returned as independent tokens, but they are not distinguished in type from unquoted words: both quoted and unquoted strings have the type `'word'`. One can tell the difference, however, by examining the attribute `.quotes`, whose value is either `"'"` or `'"'`, for a quoted string, and `None`, for an unquoted string. Backslash is an escape character inside of a quoted string, but nowhere else.

In addition to tokens, the file may contain whitespace and comments, which are discarded. Whitespace is anything that is deemed to be whitespace by `isspace()`. Newlines are not treated specially. Comments begin with `#` and continue to the end of the line.

Iter tokens. The function `iter_tokens()` returns a tokenizer, which implements the standard `next()` method, but also provides finer-grained control. First, one can peek at the next token using the `token()` method.

```
>>> toks = iter_tokens(ex.tok1)
>>> toks.token()
'12'
```

A token has the attributes `type`, `line`, and `offset`. The offset is counted from the beginning of the line.

```
>>> tok = toks.token()
>>> tok.type
'word'
>>> tok.line
1
>>> tok.offset
0
```

At the end of file, `toks.token()` will exist, but its type will be `'eof'`.

Tokenize. The function `tokenize(s)` simply converts its input to a pseudo-file (using `String.IO`) and calls `iter_tokens()`.

Error and warning. Tokens support the method `error()`, which takes an error message and raises an exception in which line and offset are included in the message. There is also a method `warning()` that prints a warning instead of raising an exception.

5.5.2 Additional methods

Has next. The method `has_next()` can be used to test the type of the next token, without consuming it.

```
>>> toks.has_next('word')
True
>>> toks.has_next('eof')
False
```

Calling `has_next()` with no argument is equivalent to calling it with the argument `'any'`.

```
>>> toks.has_next('any')
True
>>> toks.has_next()
True
```

The `has_next()` method can also be used to test for a particular token string, by providing the keyword `string`. For example:

```
>>> toks.has_next(string='12')
True
```

For a special-character token, the type and string are identical.

```
>>> next(toks)
'12'
>>> next(toks)
'+'
>>> next(toks)
'foo'
>>> toks.token()
'('
>>> toks.token().type
'('
>>> toks.has_next('(')
True
```

Boolean value. The boolean value of the iterator is `True` if there are any tokens remaining, and `False` if it is at EOF.

```
>>> bool(toks)
True
>>> notoks = iter_tokens(StringIO())
>>> bool(notoks)
False
```

Accept. The method `accept()` tests whether the next token has a given type; or, with the keyword `string`, it tests for the identity of the next token. If the next token satisfies the specification, it is consumed from the stream and returned. If not, `accept()` returns `None`. For example,

```
>>> toks.accept('word')
>>> toks.accept('(')
'('
```

Require. The method `require()` is like `accept()`, except that it signals an error if the specification is not satisfied.

```
>>> toks.token()
'bar=42.0,'
>>> toks.require(')')
Traceback (most recent call last):
    ...
Exception: [.../examples/tok1 line 1 char 9] Expecting ')'
>>> toks.require('word')
'bar=42.0,'
>>> toks.token()
'baz='
>>> toks.require(string='baz=')
'baz='
```

Note that `require()` returns `None` if eof is required:

```
>>> notoks.require('eof')
>>>
```

5.5.3 Syntax

We distinguish between the "hard" special characters `'"#` and the "soft" special characters `()[]{}`. The choice of hard special characters cannot be modified, but one can choose a different set of soft special characters.

One can also choose to have newlines returned as tokens. Only newlines at the end of non-empty lines are returned as tokens. A line consisting solely of a comment is considered empty.

These choices are encapsulated as a `Syntax` object. For example:

```
>>> syn = Syntax(special='()[]{}.,:=', eol=True)
>>> out = load_tokens(ex.tok1, syntax=syn)
>>> out[4:10]
['bar', '=', '42', '.', '0', ',']
```

If `special` is omitted, one gets the default soft special characters `()[]{}`. The parameter `eol` defaults to `False`.

The variable `DefaultSyntax` contains the default syntax: the soft special characters are `()[]{}` and newlines are never returned as tokens.

One can change syntax while scanning. The scanner returned by `iter_tokens()` has methods `push_syntax()` and `pop_syntax()`. They may affect the value of methods like `has_next()` or `token()` that look ahead in the input: the lookahead token is rescanned after a change in syntax.

5.5.4 Writing tokens

There is no `save_tokens()` function. The token stream is generally only an intermediate step in building a structured object such as a grammar. The convention used with grammars and trees is to define a "loader" that can be used to scan a structured file, and to write an object to a file in a scanable form. The loader generally has paired `scan` and `unscan` methods for each type of expression in the format.

One piece of functionality is provided here as a convenience for unscan methods. Syntax instances have a method `scanable_string()` that produces a version of a string that can be written to a file, and will produce the original string when scanned in by `iter_tokens()`, assuming that the same syntax is in use. Specifically, `scanable_string()` returns a quoted version of the string if it contains a space or special character, and returns the string unchanged otherwise.

```
>>> syn.scanable_string('foo')
'foo'
>>> syn.scanable_string('foo:bar')
"'foo:bar'"
```

The function `scanable_string` uses the default syntax.

```
>>> fn = tmpfile()
>>> out = outfile(fn)
>>> out.write(scanable_string('hi'))
2
>>> out.write(' ')
1
>>> out.write(scanable_string('x + y'))
7
>>> out.write(' ')
1
```

```
>>> out.write(scanable_string('oh \u306e!'))
7
>>> out.write('\n')
1
>>> out.close()
>>> print(contents(fn), end='')
hi 'x + y' 'oh \u306e!'
>>> load_tokens(fn)
['hi', 'x + y', 'oh \u306e!']
```

Note: when writing non-word tokens, one should write them as they are. The `scanable_string()` method converts its input to something that scans in as a *word* token.

5.6 Formatting

5.6.1 Indenter

The `Indenter` class provides a Unicode output file that does automatic indentation. There is a prevailing indentation level, and indentation spaces are automatically inserted after each newline that is written to the formatter. The level of indentation is increased using `begin_indent()` and decreased using `end_indent()`. It is initially zero. A formatter may be turned off, in which case writing commands are accepted but generate no output. The formatter is initially on.

```
>>> out = Indenter()
>>> out.write('hi there\n')
>>> out.begin_indent()
>>> out.write('foo\n')
>>> out.begin_indent()
>>> out.write('bar\n')
>>> out.write('baz\n')
>>> out.end_indent()
>>> out.end_indent()
>>> out.write('blip\n')
>>> print(out.getvalue(), end='')
hi there
   foo
      bar
      baz
blip
```

5.6.2 Tabular

The function `tabular()` takes a table, represented as an iterator over rows (lists), and produces a string representation with aligned columns. It converts

the table to a list (infinite generators will not work!) and sets the width of each column to the maximum width of the string representation of any object in the column.

```
>>> table = [['hi there', 42],
...          ['foo', 15],
...          ['elephants', 20]]
>>> print(tabular(table))
hi there  42
foo       15
elephants 20
```

5.7 Wget

The function `wget()` is a shorthand for `urllib.urlretrieve()`.

Chapter 6

XML files: `seal.xml`

This chapter documents the module `seal.xml`. The examples assume that one has done:

```
>>> from seal.xml import *
>>> from seal.io import ex, contents
>>> from seal.tree import nodes, subtree, subtrees, terminal_string
```

The standard Python library provides XML parsing, but the facilities it provides generally signal an error when encountering ill-formed XML. Unfortunately, a great deal of XML on the web is ill-formed, and aborting is not a very graceful way of dealing with it. The XML parser in `seal.xml` is designed to be very robust and lightweight.

Although it might seem that a chapter on XML belongs in the "Python extensions" part of the documentation, it is placed here because it builds on `seal.tree`.

6.1 XML tags

6.1.1 Iter and load tags

The XML parser described in the previous section calls `iter_xml_tags()` to convert the file to a stream containing a mix of XML tags and character data. The function `iter_xml_tags()` is actually the constructor for a class. It has no methods beyond the standard iterator methods. For example:

```
>>> for elt in iter_xml_tags(ex.xml1): print(repr(elt))
...
<Tag start html [] 0>
'\n'
<Tag start body [('foo', 'hi&bye'), ('bar', '16')] 7>
'\nA "little" '
<Tag start b [] 59>
'example'
<Tag end b [] 69>
'.\n'
<Tag end body [] 75>
'\n'
<Tag end html [] 83>
'\n'
```

The elements are either of type `Tag` or of type `str`.

The XML standard requires quotes around attribute values, but the tag scanner does not insist on them. Entity references are expanded in character data as well as in attribute values.

The function `load_xml_tags()` converts the iterator into a list:

```
>>> tags = load_xml_tags(ex.xml1)
>>> tags[0]
```

```
<Tag start html [] 0>
>>> len(tags)
12
```

6.1.2 Tags

A `Tag` instance has the following attributes.

`type`	One of `"start"`, `"end"`, or `"empty"`.
`label`	The label (category) of the tag.
`ftrs`	A list of pairs (*att*, *value*).
`cpos`	The character position in the plain text file.
`line_number`	The line number in the original XML file.

For example:

```
>>> tag = tags[2]
>>> tag
<Tag start body [('foo', 'hi&bye'), ('bar', '16')] 7>
>>> tag.type
'start'
>>> tag.label
'body'
>>> tag.ftrs
[('foo', 'hi&bye'), ('bar', '16')]
>>> tag.cpos
7
>>> tag.line_number
2
```

Note that the features are represented as a list of pairs. If multiple features have the same key, all will be present.

6.1.3 Entities

The tag iterator calls `decode_xml_entities()` to convert XML entities like "`&`" to characters ("`&`").

The known codes are listed in `EntityTable`. For example:

```
>>> EntityTable['amp']
'&'
```

6.2 XML trees

6.2.1 Load XML

The main function is `load_xml()`. It reads an XML file and converts it to a tree. For example, consider the file `ex.xml1`:

```
>>> print(contents(ex.xml1), end='')
<html>
<body foo="hi&amp;bye" bar=16>
A &quot;little&quot; <b>example</b>.
</body>
</html>
```

We read it as a tree:

```
>>> xml1 = load_xml(ex.xml1)
>>> print(xml1)
0    (html
1       ('#CDATA' '\n')
2       (body
3          ('#CDATA' '\nA "little" ')
4          (b
5             ('#CDATA' example))
6          ('#CDATA' '.\n'))
7       ('#CDATA' '\n'))
```

6.2.2 Examining the tree

The functions from `seal.tree` can be useful with XML trees. For example, one can pick out subtrees using the function `subtrees()`. It takes a second argument which is either a category or a predicate.

```
>>> subtrees(xml1, 'b')
[<Tree b ...>]
>>> subtrees(xml1, lambda x: x.cat == '#CDATA' and x.word != '\n')
[<Tree #CDATA '\nA "little" '>, <Tree #CDATA example>, <Tree #CDATA '.\n'>]
```

The function `subtree()` is just like `subtrees()`, except that it returns a tree; it signals an error if the specified tree does not exist or is not unique.

```
>>> body = subtree(xml1, 'body')
```

The function `terminal_string()` returns the string contents of a node.

```
>>> terminal_string(body)
'\nA "little"  example .\n'
```

The `subtrees()` function will not recurse inside any node that it returns. For example:

```
>>> subtrees(xml1, lambda x: not x.word)
[<Tree html  ...>]
```

To retrieve all nodes matching a given criterion, use the function `nodes()` and list comprehension:

```
>>> [n for n in nodes(xml1) if not n.word]
[<Tree html ...>, <Tree body ...>, <Tree b ...>]
```

It is worth noting that the `iter_xml_trees()` function handles ill-formed XML gracefully. For example:

```
>>> print(contents(ex.bad.html), end='')
<html>
<body>
<p>
This is an example with lots
of missing end tags.
<table>
<tr><th>Name <th>Rank <th>SerialNo</tr>
<tr><td>Smith <td>Corporal <td>1234567</tr>
<tr><td>Jones <td>Private <td>7654321</tr>
<tr><td>Howard <td>Major General <td>0000001</tr>
</table>
</body>
```

The missing end tags are inserted automatically:

```
>>> bad = load_xml(ex.bad.html)
>>> rows = subtrees(bad, 'tr')
>>> for row in rows: print(terminal_string(row))
...
Name  Rank  SerialNo
Smith  Corporal  1234567
Jones  Private  7654321
Howard  Major General  0000001
```

End tags are paired with the nearest matching start tag. If there is no start tag with the same label, the end tag is silently ignored.

Within the region spanned by a matching tag pair, there may be unmatched start tags. They are dealt with in a right-to-left pass, as follows. First, every category has a "precedence" assigned to it. (The precedence is only meaningful for HTML categories; it is a fixed constant for any non-HTML categories.) If the precedence is zero (for categories `br` and `img`), the unmatched start tag is converted to an empty tag. Otherwise, the "span" of the unmatched start tag is grown to cover as much material as possible, until it reaches the end of the region defined by the encompassing explicit tag pair, or until it encounters a node of equal or higher precedence. In other words, no element created from an unmatched start tag will contain a child whose precedence is equal to or greater than its own.

To get the value for an attribute, use the function `getvalue()`.

```
>>> getvalue(body, 'foo')
'hi&bye'
```

6.2.3 Tidy

When there are unmatched start tags, `iter_xml_trees()` calls the function `tidy()` to decide where the end tags should go. It uses a table encoding intuitive operator precedences for HTML tags.

Part II

Math and Machine Learning

Chapter 7

Math

7.1 Probability: `seal.prob`

This chapter documents the module `seal.prob`. The examples assume that one has done:

```
>>> from seal.prob import *
```

7.1.1 Functions

The module provides a collection of generally useful functions. For the purposes of these functions, a **vector** is any object that contains numbers and implements `__iter__()`, `__len__()`, and `__getitem__()`. A **distribution** is a vector whose elements are non-negative and sum to one.

Base-two log. The function `lg()` returns the base-two logarithm of a number.

```
>>> lg(2)
1.0
>>> lg(1024)
10.0
```

Taking `lg(0)` signals an out of domain error. (Although the Python documentation warns that some implementations may return a value representing infinity, instead.)

Entropy. The function `entropy()` returns the entropy of a distribution. Specifically, `entropy(p)` computes:

$$-\sum_{x \in \mathsf{p}} x \lg x$$

For example:

```
>>> entropy([.5, .5])
1.0
>>> entropy([.25, .25, .5])
1.5
```

Zero values are ignored—they do not cause an error even if `lg(0)` signals an error.

```
>>> entropy([.5, .5, 0])
1.0
```

Dot product. The function `dotprod()` returns the dot product of two vectors. It signals an error if the vectors differ in length.

```
>>> dotprod([.2, .5, .3], [0, 1, 1])
0.8
```

Cross entropy. The function `cross_entropy()` takes two distributions, `p` and `q`, and computes:

$$-\sum_i \mathtt{p}[i] \cdot \lg \mathtt{q}[i]$$

If any element of `q` is 0, `cross_entropy()` may signal an error, *unless* the corresponding element of `p` is also zero.

```
>>> cross_entropy([.5, .5, 0], [.5, .25, .25])
1.5
>>> cross_entropy([.5, .5, 0], [.5, .5, 0])
1.0
```

Divergence. The function `divergence()` returns the divergence of distributions `p` and `q`. The divergence is simply the cross entropy minus the entropy of `p`.

```
>>> divergence([.5, .5, 0], [.5, .25, .25])
0.5
>>> divergence([.5, .5, 0], [.5, .5, 0])
0.0
```

F-measure. The function `f_measure()` returns the F-measure given precision and recall.

```
>>> f_measure(.5, .5)
0.5
>>> f_measure(1, .5)
0.6666666666666666
>>> f_measure(1, 0)
0.0
```

7.1.2 Dist

A distribution (class `Dist`) is essentially a dict whose keys are n-tuples. For example, consider the following:

```
>>> d = Dist([('the', 'big', 'cat', 1),
...           ('the', 'big', 'dog', 3),
...           ('the', 'fat', 'cat', 2),
...           ('the', 'fat', 'dog', 1),
...           ( 'a' , 'big', 'cat', 1),
...           ( 'a' , 'fat', 'cat', 2)])
...
```

Each tuple in the initializer is called an *item.* In this case, each item is a 4-tuple consisting of three strings and a number. Each item consists of a *key* and a *value.* The keys are also called *events.* The last element is always the value, and the elements excluding the last constitute the key. In our example, the keys are 3-tuples of strings, and the values are integers. The resulting Dist is said to have dimensionality 3. That is, the dimensionality of a Dist is the dimensionality of its keys.

Unlike with a regular dict, with a Dist one may use partial keys. For example, one may do:

```
>>> sd = d['the','big']
```

The result is a dist of smaller dimensionality, called a subdist. In this case, the subdist `sd` represents the mapping:

```
'cat' => 1
'dog' => 3
```

For example:

```
>>> sd['dog']
3
```

Incidentally, Dists also differ from dicts with respect to the treatment of missing keys. Accessing a nonexistent key yields a value of 0 instead of an error:

```
>>> sd['antelope']
0
```

The values in a dist may be anything, but there are two particularly common cases: the values are integers representing counts (as in our example), or they are floats representing probabilities. The process of *normalization* converts the former to the latter.

In simple normalization, all the values in the entire Dist are added together, and each value is then divided by the total. In the case of our example, the result would be the following mapping:

```
('the', 'big', 'cat') => .1
('the', 'big', 'dog') => .3
('the', 'fat', 'cat') => .2
('the', 'fat', 'dog') => .1
( 'a' , 'big', 'cat') => .1
( 'a' , 'fat', 'cat') => .2
```

Alternatively, one can divide each of the keys into two pieces: a *condition* and an *outcome.* That is, when one normalizes, one can specify a *conditionalization dimension* that is less than the key dimension. If we specify that conditions are 2-tuples, the result of normalization is as follows:

```
('the', 'big'): 'cat' => .25
                'dog' => .75
('the', 'fat'): 'cat' => .666667
                'dog' => .333333
( 'a' , 'big'): 'cat' => 1.00
( 'a' , 'fat'): 'cat' => 1.00
```

To summarize, an item consists of a key and a value, and a key consists of a condition and an outcome. An unconditionalized Dist is one in which the condition dimension is 0, and keys are the same as outcomes.

7.1.3 Estimators

The classes described here are not currently implemented. I need to determine whether they exist in an older form of Seal.

A **Counts** object is constructed from a sample. It contains or can compute the following information. The variable x ranges over outcome types, the variable s ranges over subsamples, and $\bar{s}$ represents the complement of subsample s.

$N(x)$	tokens of type x
$N = \sum_x N(x)$	total number of tokens
$T(r) = \sum_x [\![N(x) = r]\!]$	types with count r
$T = \sum_x 1$	total number of types
$N_s(x)$	tokens of type x in subsample s
$T_s(r) = \sum_x [\![N_a(x) = r]\!]$	types with count r in subsample s
$N_s(r) = \sum_x [\![N_{\bar{s}}(x) = r]\!] N_s(x)$	tokens of types with other-sample count r

Estimators are classes with a single method **call** that returns a **Distribution**.

The **RelativeFrequencyEstimator** is constructed from a **Counts** object. It returns the distribution

$$\hat{p}(x) = \frac{N(x)}{N}$$

The **LidstoneDistribution** with parameter λ returns

$$\hat{p}(x) = \frac{N(x) + \lambda}{N + T\lambda}$$

Special cases are the Laplace estimator ($\lambda = 1$) and the Expected Likelihood Estimator (ELE), also known as the Jeffrey-Perks law ($\lambda = 1/2$).

The **DeletedEstimator** divides its sample into two subsamples.

7.2 Matrices: seal.mat

The module `seal.mat` provides a minimal implementation of matrices.

7.3 Clustering: seal.cluster

```
>>> from seal.cluster import *
```

7.3.1 UTM

The class `UTM` represents an upper triangular matrix. Cells in a triangular matrix are identified by a pair of indices, but the order of the indices does not matter. The rows/columns of the matrix are identified not only by index but by name. One provides a list of names to create the matrix.

```
>>> utm = UTM(names=['foo', 'bar', 'baz'])
>>> print(utm)
foo bar          0
foo baz          0
bar baz          0
```

Otherwise, one sets and accesses cells as one would in a regular matrix.

```
>>> utm['foo','baz'] = 10
>>> utm['baz','bar'] = 20
>>> utm['bar','foo'] = 6
>>> print(utm)
foo bar          6
foo baz          10
bar baz          20
```

One may alternatively use numeric indices.

```
>>> utm[1,2] = 12
>>> print(utm)
foo bar          6
foo baz          10
bar baz          12
>>> utm[2,1]
12
```

The `len()` of the matrix is the number of rows/columns.

```
>>> len(utm)
3
```

7.3.2

Chapter 8

Machine learning: `seal.ml`

8.1 Learner API

A **learning algorithm** is an object that contains the following functions. It may be a module or an instance (in which case the following are bound methods).

`train(`*p*`,`*o*`)` Should take a `seal.ml.Problem` instance *p* and an optional output stream *o*. Should create and save a model, writing files whose pathnames begin with *p*.`modelfn`. Only files needed for `load_model()` should use the prefix *p*.`modelfn`. Other working files, including in particular gold and predicted labels on test data, should use the prefix *p*.`workfn`. Apart from writing files, the function should have no side effects. There is no return value.

`accuracy(`*w*`,`*m*`,`*o*`)` Takes a working filename *w* and (optionally) a model filename *m* and an output stream *o*. The latter two should be specifiable as keyword arguments `modelfn=`*m* and `output=`*o*. Returns a triple: proportion correct, number correct, number of test instances.

`load_model(`*m*`)` Takes a model filename prefix and returns the model. This should work even if the working files are all deleted. A model should behave as a function that takes an individual instance and returns a predicted label.

A `Problem` has the following members: `train`, `test`, `options`, `modelfn`, `workfn`. The constructor takes those names as keyword arguments, or else they may be set after the problem is created. *Train* and *test* should be iterables of learning instances. *Options* is a dict. *Modelfn* and *workfn* are pathnames.

```
>>> from seal.ml import Problem
>>> p = Problem(modelfn='/tmp/model', workfn='/tmp/work')
>>> p.options = {'foo': 'bar'}
>>> p.train = ...
>>> p.test = ...
```

A `Problem` also provides an `init()` method, for convenience. It creates *p*.`modelfn` and *p*.`workfn` as directories, and writes the file `ModelFN` in the *p*.`workfn` directory. After calling `init()`, one can get the contents of the modelfn from the workfn using the function `load_modelfn()`.

```
>>> p.modelfn
'/tmp/model'
>>> p.init()
>>> from seal.ml import load_modelfn
>>> load_modelfn('/tmp/work')
'/tmp/model'
```

A `Problem` also has a method `clean()` which deletes the working files, model files, or both. If the filenames are to be understood as prefixes, add the keyword `prefix=True`.

```
>>> p.clean('work')               # deletes /tmp/work recursively
>>> p.clean('work', prefix=True)  # deletes /tmp/work* recursively
>>> p.clean('model')
>>> p.clean('all')                # deletes both
```

8.2 Instances: seal.ml.instance

Most instance functions are contained in `seal.ml.sym` and `seal.ml.num`. The generic module `seal.ml.instance` contains only the function `get()`, which returns the value for an attribute, and works for both symbolic and numeric instances.

```
1 >>> get(inst, att)
```

8.3 Symbolic

8.3.1 Instances

For classifier training datasets, we wish to keep the format as simple as possible. A data matrix is a list of instances. An instance is a list of features or attribute-value pairs.

To have sufficient flexibility, we will need to deal with at least a few different instance formats. In all cases, the first element is the label, and the remaining elements are features.

- **Symbolic instance**: the features are strings.
- **AV instance**: the features are pairs of attributes (strings) and values (strings).
- **Dense numeric instance**: the features are numbers representing attribute values, where attributes are positional.
- **Sparse numeric instance**: the features are pairs of attribute IDs (integers) and value (numbers).

There is a separate class for each instance type. They are all subclasses of `list`.

8.3.2 Symbolic instances

The functions described here allow features to be either strings or AV pairs. Actually, the save function will allow anything that the `unicode()` function accepts, though the load function only returns instances containing strings or pairs.

```
>>> from seal import ex
>>> from seal.ml import sym, instance
>>> insts = sym.load_instances(ex.class2.train)
>>> print(insts[0])
- len:S wid:S
>>> sym.save_instances(insts[:3], '-')
-       len=S   wid=S
-       len=S   wid=M
-       len=S   wid=M
```

One can use `ml.instance.get()` on the instance. It returns the value for an attribute, or `None` if the attribute is not found. If the instance contains features rather than attribute-value pairs, `get()` returns `True` if the feature is found, and `None` otherwise.

```
>>> instance.get(insts[0], 'wid')
'S'
```

8.3.3 Stats

The function `statistics()` compiles statistics for a dataset. It returns two dicts, one containing counts for labels, and one containing counts for features.

```
>>> (ls,fs) = sym.statistics(insts)
>>> sorted(ls)
['+', '-']
>>> ls['-']
8
>>> ls['+']
6
>>> sorted(fs)[:3]
[('len', 'L'), ('len', 'M'), ('len', 'S')]
>>> fs['wid', 'S']
5
```

The function `print_stats()` prints out the stats for human reading. Labels and features are sorted from most-frequent to least-frequent.

```
>>> sym.print_stats(insts)
Labels:
       8 -
       6 +

Features:
       7 len S
       5 wid L
       5 wid S
       4 len L
       4 wid M
       3 len M
```

8.4 Numeric

8.4.1 Coder

A `Coder` converts symbolic instances to numeric instances. A coder can be created from a list of instances:

```
>>> from seal.ml import num
>>> coder = num.Coder(insts)
```

It enumerates the labels and features that it encounters in the data from which it is created. (The data itself is discarded.) Then it maps symbol to numeric values:

```
>>> coder.label('+')
2
>>> coder.feature(('wid', 'M'))
3
```

The coder can be applied as a function to a single instance:

```
>>> testinsts = sym.load_instances(ex.class2.test)
>>> coder(testinsts[0])
<Instance 1 3:1 4:1>
```

It may be applied to a list of instances using its `encode()` method:

```
>>> coder.encode(testinsts)
[<Instance 1 3:1 4:1>]
```

There is also an iterator version: `iter_encode()`.

The coder can be saved to a file using its `save()` method.

```
>>> coder.save('tmp.coder')
```

8.4.2 Decoder

A `Decoder` inverts a coder. It can be created from a coder or from a filename of a file in which a coder was saved.

```
>>> decoder = num.Decoder(coder)
>>> decoder = num.Decoder(filename=ex.coder1)
>>> decoder.label(1)
'-'
>>> decoder.feature(4)
('len', 'M')
```

It can be called as a function on a numeric instance, or the methods `decode()` or `iter_decode()` can be applied to a list of instances.

8.5 Libsvm

8.5.1 Train

```
>>> from seal.ml import Problem, libsvm
>>> p = Problem(modelfn='/tmp/model', workfn='/tmp/work')
>>> p.train = sym.load_instances(ex.class2.train)
>>> p.test = sym.load_instances(ex.class2.test)
>>> libsvm.train(p)
```

The following files are written in `/tmp`:

```
model.coder  work.log   work.train       work.test
model.model  work.pred  work.train.asc   work.test.asc
                        work.train.prov  work.test.prov
```

The `.coder` file contains the numeric encoding for the labels and features, and the `.model` file is produced by the executable `svm-train`. What `svm-train` writes to stdout is saved in the `.log` file. The `.train` and `.test` files contain the training and testing instances in libsvm format. Ascii versions are found in the `.asc` files, and provenance information is written in the `.prov` files. Finally, the executable `svm-predict` produces the `.pred` file; it contains the model's predictions on the test instances.

One can load and print the ascii instances as follows:

```
>>> sym.print_instances(sym.load_instances('/tmp/work.train.asc')[:3])
- len=S wid=S
- len=S wid=M
- len=S wid=M
>>> sym.print_instances(sym.load_instances('/tmp/work.test.asc'))
- len=M wid=M
```

8.5.2 Coder and decoder

The convenience function `coder()` takes a name and loads a coder from the file *name*`.coder`.

```
>>> coder = libsvm.coder('/tmp/model')
>>> einst = coder(p.train[0])
>>> einst
<Instance 1 1:1 2:1>
```

The convenience function `decoder()` takes a name and loads a decoder from *name*`.coder`.

```
>>> decoder = libsvm.decoder('/tmp/model')
>>> decoder(einst)
<Instance - len:S wid:S>
```

8.5.3 Accuracy

The function `accuracy()` compares test labels and predicted labels, and returns three values: the proportion of correct predictions, the number of correct predictions, and the total number of predictions. (The first number is the ratio of the second and the third.)

```
>>> libsvm.accuracy('/tmp/work')
(1.0, 1, 1)
```

In accordance with the learner API (8.1), `accuracy()` also accepts a second argument (the model filename), but ignores it.

8.5.4 Predictor

Also in accordance with the learner API, one can create a classifier using `load_model()`:

```
>>> f = libsvm.load_model('/tmp/model')
```

The files `model.model` and `model.coder` must exist. To avoid an error if they do not exist, you can test for them:

```
>>> libsvm.model_exists('/tmp/model')
True
```

The loaded predictor behaves like a function. It takes an (unencoded) instance and returns a predicted label.

```
>>> inst = p.test[0]
>>> f(inst)
'-'
```

8.5.5 Description of libsvm format

Libsvm expects numeric instances in sparse format. Fields in an instance are separated by single spaces. The first field contains the label, which is an integer in the case of classification. Subsequent fields consist of attribute-id and value joined by a colon. Attributes are numbered beginning at 1 and must be specified in increasing order. Here is an example:

```
+1 1:0.708333 2:1 3:1 6:-1
-1 1:0.583333 2:-1 3:0.333333 4:-0.603774 5:1 6:-1
+1 1:0.166667 2:1 3:-0.333333 4:-0.433962
```

Skipped attributes have value 0.

8.6 Split learner: seal.ml.split

A split learner divides up the space of instances based on the value of one of the features (the **split feature**), training a separate sublearner for each split value.

8.6.1 Train

The sublearner may be a module or instance. It must conform to the learner API (8.1).

```
>>> from seal import ex, io
>>> from seal.ml import Problem, split, libsvm, sym
>>> lrn = split.Learner(libsvm)
```

We create the learning problem.

```
>>> p = Problem(modelfn='/tmp/model', workfn='/tmp/work')
>>> p.options = {'feature':'col'}
>>> p.train = sym.load_instances(ex.class3.train)
>>> p.test = sym.load_instances(ex.class3.test)
```

The instances in `ex.class3` have a feature `col` with two different values (`R` and `G`). The `G` instances are duplicates of the `R` instances, except that all the labels are reversed. The option `'feature':'col'` indicates that the `col` feature should be used to split the instances into groups.

In the options, one can also specify `'na_value'` and `'na_cutoff'`. The `na_value` will be used for instances that have no value for the split feature. If `na_cutoff` is provided, and a given split value v is attested fewer than `na_cutoff` times, then the value v is lumped together with the `na_value`.

Finally, one may include `'cpt'` among the options. Its value should be a subdict that will be passed as options to the sublearner.

Call the trainer:

```
>>> lrn.train(p, output=io.null)
```

Note that the output stream `None` represents stdout. `io.null` is the null output stream, which produces no output. We can see that a number of files are created, however:

```
>>> import os
>>> sorted(os.listdir('/tmp/model'))
['Params', 'Values', '_G.coder', '_G.model', '_R.coder', '_R.model']
>>> len(os.listdir('/tmp/work'))
16
```

8.6.2 Accuracy

We can compute the accuracy on the test data:

```
>>> lrn.accuracy('/tmp/work', '/tmp/model', output=io.null)
(1.0, 2, 2)
```

8.6.3 Classify

We can load a predictor from the model directory. We first delete the work directory to show that it is not used.

```
>>> p.clean('work')
>>> f = lrn.load_model('/tmp/model')
```

The model behaves as a function that takes an instance (or feature vector) and predicts its label.

```
>>> f(p.test[0])
'_'
```

8.7 Experiments: ml.experiment

One runs an experiment like this:

```
$ python -m seal.ml.experiment myfile.exp
```

One may run an experiment in Python like this:

```
>>> from seal.ml import experiment
>>> experiment.run('myfile.exp')
```

The `run()` function is a wrapper that dispatches to the actual experiment-running function, called the *command.* The command is specified in the experiment file, by naming the module in which it resides. The command is the function called `run_experiment` in that module.

An experiment file is simply a specification for a dict, possibly with nested dicts as values. The file consists of keys and values, separated by a single space character. If a key contains a dot (period), the value is a dict that is constructed from the suffix of the key. For example, consider:

```
foo 10
bar.a hi
bar.b bye
```

This is represented internally as:

```
{'foo': '10',
 'bar': {'a': 'hi',
         'b': 'bye'}}
```

Subdicts of arbitrary depth are permitted.

To be a valid experiment file, one of the keys must be `'command'`, and the key `'experiment'` must *not* be used. The value for `'command'` is the name of the module containing the relevant `run_experiment` function, e.g., `seal.dp.nivre`.

The key `'experiment'` will be added to the dict. Its value will be a subdict with three keys. In the subdict, the value of `'name'` will be the name of the experiment file, excluding an optional filename suffix `.exp`. The value of `'filename'` will be the absolute pathname of the experiment file. The value of `'dir'` will be the absolute pathname of the directory in which the experiment file resides.

The `run_experiment` function will receive two arguments: the dict and the working-directory pathname. In general, it should treat the experiment-file directory as read-only, and write any working files into the working directory.

Part III

Languages

Chapter 9

Languages

9.1 Languages: `seal.data.langdb`

The database in `seal.data.langdb` is compiled by merging data from the *Ethnologue* and from the Library of Congress's official ISO 639-2 database. It uses the `iso-639-2` and `iso-639-3` packages.

The database is called `languages`:

```
1  >>> from seal.data.langdb import languages
```

The information in `languages` exactly reflects the published databases, with the following exceptions:

- In the published databases, retired codes had no entry for scope or type, with the exception of one retired code, which had scope-type of `IL` (living language). For the sake of uniformity, I have assigned all retired codes scope `'R'` and type `'R'`.
- In the published databases, the `names` field was filled only if the language had multiple names, in which case `names` included the reference name. For the sake of uniformity, `names` now always includes the reference name, and may be a singleton list containing only the reference name. Inverted names have been treated similarly.

9.1.1 Language codes

The standard three-letter language codes are ISO 639-3 codes. There are several other code sets in the ISO 639 family.

- ISO 639-1: These are the standard two-letter language codes. Only 184 languages have a 639-1 code.
- ISO 639-2: These were created for librarians. 418 languages have a 639-2 code. 20 languages have two different 639-2 codes: a "bibliographic"

code (639-2/B) and a "terminological" code (639-2/T). The Library of Congress is the registration authority.

- ISO 639-2/B: The bibliographic version of 639-2 codes. These do not always agree with 639-3.
- ISO 639-2/T: The terminological version of 639-2 codes. These constitute a subset of 639-3.
- ISO 639-3: The standard three-letter language codes. SIL is the registration authority. These extend the 639-2/T codes to 8121 languages.
- ISO 639-5: An extension to 639-2 to cover language groupings. The Library of Congress is the registration authority.

9.1.2 Access by code

The database can be accessed by ISO-639-3 code to get a language:

```
>>> print(languages['spa'])
Code:    spa
Code2B:  spa
Code2T:  spa
Code1:   es
Type:    Living
Scope:   Language
RefName: Spanish
Name:    Spanish
```

The four codes listed are 639-3, 639-2/B, 639-2/T, and 639-1, in that order.

9.1.3 Languages

Although one accesses `languages` as a table, one iterates over it as a list of languages.

```
>>> len(languages)
8121
>>> sum(1 for lang in languages if lang.code2b != lang.code2t)
20
```

A language instance has the following members:

`code`	The 639-3 language code (a string).
`code2b`	The 639-2/B language code, or `None`.
`code2t`	The 639-2/T language code, or `None`.
`code1`	The 639-1 language code, or `None`.

`scope`	The value is `'I'` for individual language, `'M'` for macrolanguage, `'S'` for special code, and `'R'` for retired codes. The special codes are used when one needs a code for something that is not actually a language. They are `'mis'` for an uncoded language, `'mul'` when the thing to be coded contains multiple different languages, `'und'` when the language is undetermined, and `'zxx'` when the thing to be coded does not actually have linguistic content.
`type`	The value is `'A'` for an ancient language, `'C'` for a constructed language, `'E'` for an extinct language, `'H'` for an historical language, `'L'` for a living language, `'S'` for a special code, and `'R'` for retired codes.
`name`	The reference name for the language.
`names`	All names for the language, including the reference name.
`inv_names`	Inverted names (like `'English, Old'`).
`comment`	Comments.
`parent`	The macrolanguage that this language belongs to, if any.
`members`	The member languages, if this is a macrolanguage.
`retirement`	`None` unless this is a retired code. If this is a retired code, the value is an object with the following members: `code` repeats the language code, `name` repeats the name, `reason` is the retirement reason, `date` is the retirement date (a string), `replacement` is the new code this one was replaced with (if any), and `split` is an English string indicating which codes this one was split into (if any). The retirement reasons are: `'C'` for a code change, `'D'` for deletion of a duplicate code, `'M'` for the merger of multiple codes into a new code, `'S'` for the splitting of one code into multiple codes, and `'N'` for deleting of a code that represents a non-existent language. There is a value for `replacement` for the `'C'`, `'D'`, and `'M'` cases, and a a value for `split` for the `'S'` case.

9.1.4 Access by name

One can access languages by name. Since names are sometimes ambiguous, this returns a list:

```
>>> languages.named('spanish')
[<Living Language spa 'Spanish'>]
>>> languages.named('pao')
[<Living Language blk "Pa'o Karen">, <Living Language ppa 'Pao'>]
```

Note that the key need not be the reference name: “Pa’O” is one of the alternate names for language `blk`:

```
>>> languages['blk'].names
["Pa'O", "Pa'o Karen"]
>>> languages['blk'].name
"Pa'o Karen"
```

The search methods, including not only `named()` but also the methods `find()` and `search()` discussed below, normalize both the language names and the search key, as follows:

- Letters are normalized to lower case. I.e., search is case-insensitive:

```
>>> languages.named('SPANISH')
[<Living Language spa 'Spanish'>]
```

- Anything in parentheses is ignored.

```
>>> languages.named('wali')
[<Living Language wll 'Wali (Sudan)'>, <Living Language wlx 'Wali (Ghana)'>]
```

- Hyphens are treated like spaces:

```
>>> languages.named('karkar yuri')
[<Living Language yuj 'Karkar-Yuri'>]
>>> languages.named('old-english')
[<Historical Language ang 'Old English (ca. 450-1100)'>]
```

- Accents are removed:

```
>>> languages.named('yari')
[<Living Language yri 'Yar\xed'>]
```

  Here the Unicode character U+ED represents *í* (letter *i* with an acute accent).

- Everything that is not a letter is ignored in comparison:

```
>>> languages.named('p!ao?')
[<Living Language blk "Pa'o Karen">, <Living Language ppa 'Pao'>]
```

9.1.5 Access by name part

The method `named()` does not find a language if one provides only part of the name:

```
>>> languages.named('chin')
>>> languages.named('matu chin')
[<Living Language hlt 'Matu Chin'>]
```

To find a language if one knows only part of the name, used the method `find()`:

```
>>> len(languages.find('chin'))
33
```

9.1.6 Access by character sequence

The method `find()` looks for complete words in the name. (Remember that hyphen is treated as a word separator.) To find a language given only a part of a word, use `search()`:

```
>>> languages.search('ruman')
[<Living Language rup 'Macedo-Romanian'>]
```

Chapter 10

Lexica

10.1 Panlex

10.1.1 Basic usage

The source file:

`archive/2013/main/semdep/panlex/panlex.py`

In the directory `/cl/data/panlex/lex`, this writes the files `spa-eng.txt` and `spa-eng-sources.txt`:

```
1  >>> from panlex import Bilex
2  >>> b = Bilex('spa','eng')
3  >>> b.create()
```

10.1.2 Structure

We usually think of a lexical entry as having a form something like the following:

bank$_1$, *n.* 1. A financial institution. 2. Any storage facility. *v.* 1. To store.
bank$_2$, *n.* 1. The shore of a river.

Multiple entries may share the same headword, in which case the headword represents two separate homonyms. Each entry consists of numbered definitions, grouped by part of speech.

Panlex takes the numbered definition to be primary, and calls it a **denotation**. In the above example, there are four denotations. A denotation combines a headword (called an **expression**), a part of speech, and a definition.

A bilingual dictionary can be viewed as consisting of paired denotations, one in each language. A denotation For this purpose, Panlex introduces the more abstract notion of a **meaning**. Each denotation corresponds to a unique meaning, but it is possible for a single meaning to link multiple denotations.

Accordingly, a dictionary (which Panlex calls an **approver**) consists of a list of meanings. A meaning may have a definition (though it need not). There are one or more denotations that correspond to the meaning. The denotation itself does not have a language, but it points to an expression (the headword) which does.

The database consists of triples of form subject-relation-object. Subject and object are instances, and the relation is a constant. The types of triples are listed in Table 10.1.

10.1.3 Utility functions

The function `attribute_entries()` iterates over the records for a given subject type or a given subject-relation pair. For example:

```
>>> i = attribute_entries('expression', 'label')
>>> i.next()
(('expression', 'label', 'string'), '3990756' u'!')
```

The entries are of form (t, v_1, v_2), where t is of form (t_1, r, t_2).

Collect variety languages. The function `collect_variety_languages()` iterates over the variety-language records, and constructs a table indexed by variety ID (an int), whose value is the variety's language. E.g.:

```
>>> vlangs = collect_variety_languages()
>>> vlangs[187]
'eng'
```

Collect approvers. The function `collect_approvers()` returns a table indexed by approver ID, in which the values are lists of form [lang, variety, quality, title].

Extracting bilexicons. A bilexicon is represented in Python by the class `Bilex`:

```
>>> b = Bilex('spa','eng')
```

Create raw. The first step is to create the raw bilexicon:

```
>>> b.create_raw()
```

This takes about 25 minutes to run. The output (in this example) is the file `spa-eng-raw.txt` in the directory `/cl/data/panlex/lex`.

The `create_raw()` method starts by loading the variety-language table, which maps varieties to their languages.

Then it goes through the expression-variety records, creating a table of expressions. The keys are expressions (ints) and the values are lists of form [variety,

denotation	`type`	the constant `'Denotation'`
denotation	`expression`	the expression representing the containing entry
denotation	`wordClass`	the part of speech
denotation	`meaning`	shared across entries and across languages
expression	`type`	always the constant `'Expression'`
expression	`label`	a string value, either ASCII or XML string
expression	`languageVariety`	that the word belongs to
expression	`degradedText`	reduced label, only lowercase letters and digits
wordclass	`type`	always the constant 'WordClass'
wordclass	`label`	an ASCII string
wordclass	`sameAs`	the value is a resource, whatever that is
meaning	`type`	always the constant 'Meaning'
meaning	`identifier`	an ASCII string, represents an identifier in the original source
meaning	`definition`	a definition object
meaning	`approver`	the original source
definition	`type`	always the constant 'Definition'
definition	`label`	the string
definition	`languageVariety`	that the string is written in
variety	`type`	always the constant 'LanguageVariety'
variety	`label`	its name (ASCII)
variety	`varietyOf`	the language
language	`type`	may be 'Language', 'PanlexLanguage', or 'LexvoLanguage'
language	`iso639-3`	the iso639-3 code
language	`iso639-1`	the iso639-1 code
iso639-3	`type`	always the constant 'Iso639-3'
iso639-3	`sameAs`	a lexvo iso639-3 code
iso639-1	`type`	always the constant 'Iso639-1'
iso639-1	`sameAs`	a lexvo iso639-1 code
approver	`type`	always the constant 'Approver'
approver	`variety`	the language variety that the source documents
approver	`registrationDate`	a date
approver	`label`	the name of the source (ASCII)
approver	`creator`	ASCII or XML string
approver	`quality`	an integer
approver	`isbn`	ASCII
approver	`license`	a license object
approver	`year`	of publication (integer)
approver	`publisher`	ASCII
approver	`title`	ASCII or XML string
approver	`homepage`	a URL
license	`type`	always the constant 'License'
license	`label`	ASCII name

Table 10.1: Types of triples

label, degraded text]. An entry is created only for expressions whose variety's language is one of the two languages of interest. Label and degraded text are initially set to the empty string.

Next it goes through the expression-label and expression-degraded-text records, filling in the other fields of the expression entries.

Next it creates a denotations table. It goes through the denotation-expression records. If the expression has an entry in the expressions table, then a new entry is created in the denotations table. The key is the denotation (an int), and the value is a list of form [expression, part of speech, meaning]. Initially only the expression is set. Part of speech is initialized to the empty string and meaning is initialized to 0.

Next it goes through the denotation-pos records and the denotation-meaning records, filling in the remaining fields in the denotation entries.

By that point, memory is pretty much full. Output is written to *lang1*`-`*lang2*`-raw.txt`. We pass through the denotations table. Each denotation entry contains an expression ID, we use it to fetch the expression entry. The expression entry contains a variety ID; we use it to look up the language. Each denotation generates one line of output, of form:

m lang v expr degraded pos d e

The single letters represent integer IDs: meaning (m), variety (v), denotation (d), expression (e). The denotation and expression IDs are included only for debugging purposes.

Sort raw. The method `sort_raw()` calls Unix `sort` to sort the raw file by meaning, language, variety, and label. The output is written to *lang1*`-`*lang2*`-m1.txt`. It takes a couple minutes to run.

Create m2. The method `create_m2()` adds approvers, and also filters out monolingual meanings. (I tried adding approvers when creating the raw file, but Python runs out of memory.)

```
1 >>> b.create_m2()
```

The method scans through the `m1.txt` file, collecting a table of meanings. For each block of meanings, note is kept of whether both languages are seen. If so, an entry is created in the meanings table, and otherwise no entry is created. The meanings table is indexed by meaning ID, and the value is the approver ID (initialized to 0).

After creating the meanings table, the method passes through the meaning-approver records and sets the values (approvers) for the meanings.

Next it calls `collect_approvers()` to get the quality information for each approver.

Finally, it passes a second time through the `m1.txt` file. Each time it encounters a new meaning, it looks in the meanings table to see whether it should be kept or not. If the meaning is a keeper, the quality of the approver is looked

up in the approvers table. Each line from `m1.txt` that is to be kept is copied to `m2.txt`, and two new fields are added at the end: approver ID and quality. Hence the lines in `m2.txt` are of form:

m lang v expr degraded pos d e a q

where "a" is approver and "q" is quality (both are ints).

Create sources. The method `create_sources()` extracts detailed information about each of the approvers. It writes the file *lang1*`-`*lang2*`-sources.txt`. The line format is:

a rel value

where "a" is the approver ID. The relations (attributes) are: `lang`, `variety`, `regdate`, `label`, `creator`, `isbn`, `lic_id`, `license`, `year`, `publ`, `title`, and `url`. An empty line is inserted before each block of records sharing a common value for "a."

By word. The method `by_word()` creates a file containing lines of form

word-lang1 quality word-lang2

The method `sort_by_word()` then sorts that file.

It turns out that the quality scores for the approvers are not very informative about whether the entries are actually good. For example, the top quality source (quality 7) for the Spanish word "a" includes meanings "crazy," "missionary," and "physical"—completely bogus. A much better gauge appears to be the number of sources in which the translation occurs.

10.2 Census: `seal.data.census`

The module `seal.data.census` contains the U.S. census name information. The basic function is `get()`.

```
>>> from seal.data import census
>>> harry = census.get('harry')
>>> harry
<Name HARRY mr=70 lr=2812>
```

The argument to `get()` is case-insensitive. If the argument is not found in the database, the return value is `None`.

A name object encapsulates three entries:

```
>>> harry.male
<Entry HARRY 0.251000 52.991000 70>
>>> harry.female
<Entry None 0.000000 0.000000 0>
>>> harry.last
<Entry HARRY 0.004000 56.240000 2812>
```

Note that "Harry" never appears as a female name. A unique zero entry is used in all such cases.

An entry contains four pieces of information:

```
>>> harry.male.string
'HARRY'
>>> harry.male.freq
0.251
>>> harry.male.cumfreq
52.991
>>> harry.male.rank
70
```

The frequency and cumulative frequency are in percent: about a quarter of a percent of male census entries (in the sample on which the tables were based) have first name "Harry."

A name object also has a method `maleness()`, which returns the conditional probability that the name is male, given that it is a first name. That is, if m is the frequency in the male entry, and f is the frequency in the female entry, maleness is $m/(m + f)$. (If the name never occurs as a first name, maleness defaults to 0.5.)

```
>>> jordan = census.get('jordan')
>>> jordan.maleness()
0.8235294117647058
```

One can iterate over all names by calling the function `names()`.

Chapter 11

Universal Corpus

11.1 Corpus: `seal.uc.corpus`

This section documents the module `seal.uc.corpus`. The examples assume that one has done:

```
>>> from seal.uc.corpus import *
>>> from seal.io import ex
```

11.1.1 Document preparation pipeline

The corpus is intended to support collection and preparation of documents, and not just the finished product. Logically, one begins by searching for relevant documents, collecting bibliographic information into a document "card catalog."

Some of the documents are downloaded. The original format depends on the source, but one format of interest consists of page images accompanied by plain text produced with OCR software. These are paginated documents.

Documents of particular interest are grammars, dictionaries, and texts with translations. In such documents, individual pages often contain two or more languages. We would like to extract bitexts where possible. To train and evaluate bitext extraction, we annotate some pages manually. This produces bitext-annotated pages.

When we have identified monolingual texts—whether complete documents, individual pages, or merely snippets—we can do further processing that includes segmentation (dividing the text into sentences or comparable units), tokenization, and wordlist construction. In some cases, texts, segmentations, tokenizations, or wordlists are alignable, defining parallel texts, parallel segmentations, etc. These digested items constitute the **kernel**; they represent the "finished product."

11.1.2 Item store and corpus

The general file organization is by the type of processing that is being done. We wish to make it as easy as possible to add new processing, providing new "views" on old documents.

One creates a new empty corpus using `create_corpus()`. It takes the pathname of the corpus; it is an error if a file with that pathname already exists. For example:

```
>>> create_corpus('/tmp/corpus')
```

There is no return value. One can delete the corpus using `delete_corpus()`:

```
>>> delete_corpus('/tmp/corpus')
```

One can also create a new corpus by copying an old one. The new corpus has no connections to the old one. In particular, modifications to the new corpus will not affect the old corpus or vice versa.

```
>>> copy_corpus(ex.uc, '/tmp/corpus')
```

It is an error if the old corpus does not exist, or if the new filename does exist.

One opens an existing corpus by instantiating the `Corpus` class. The constructor takes a single argument: the pathname of the corpus directory. There is a sample corpus in `ex.uc`.

```
>>> corpus = Corpus('/tmp/corpus')
```

The corpus is a collection of `ItemStore` objects. The class `ItemStore` is a specialization of `DataTable`. The main difference is this: a data table is represented by a pair of files (`.tab` and `.hdr`), whereas an item store is a directory containing those two files along with files representing the contents of individual items.

The `ItemStore` class itself differs from `DataTable` only in that it has an additional attribute, `dirname`, which contains the pathanme of the store's directory. The data table resides in the files `index.tab` and `index.hdr` in that directory.

The corpus has attributes for each item store that it contains:

- `users`: representing annotators
- `langs`: the languages
- `cards`: a card catalog of documents. There is no implication that a document listed here appears anywhere else in the corpus. The card catalog includes bibliographic records for many items of interest that have not (yet) been fetched.
- `pages`: OCR output for scanned page images.

For example:

```
>>> len(corpus.cards)
119
```

One can find items using the `DataTable` search methods. For example, one can directly access items by ID:

```
>>> print(corpus.cards['111'])
docid        111
name
author       R. F. Fortune
title        Arapesh
pub_date     1942
pub_country
pubdom
source       Digital General Collection
source_id    ACR7567.0019.001
url          http://name.umdl.umich.edu/ACR7567.0019.001
media
scan         Full Text
ocr
script       Latin-based
langs
content_lang ape
gloss_lang   eng
local_url
notes
```

Languages have attributes `langid` and `name`:

```
>>> print(corpus.langs['ape'])
langid ape
name   Arapesh
```

Users have attributes `userid` and `rights`. (The attribute `rights` is not currently used.)

```
>>> print(corpus.users['abney'])
userid abney
rights
```

For indexed attributes, one can use the method `where()`.

```
>>> len(corpus.pages.where('docid', '111'))
252
```

To determine which attributes are indexed, use `indexed_fields()`:

```
>>> corpus.pages.indexed_fields()
('itemid', 'docid')
```

More general than `where()` is the method `items()`. It uses indices for efficiency, when they are available.

```
>>> corpus.pages.items(docid='111', number='23')
[<Page 111 23>]
```

To get the list of attested values for an attribute, use `values()`.

```
>>> corpus.pages.values('number')[:5]
['1', '2', '3', '4', '5']
>>> len(corpus.pages.values('number'))
412
```

11.1.3 Item

The more substantial difference between a data table and item store lies in the items themselves. They belong to the class `Item`, which is a specialization of `Record`. The main additional functionality that an `Item` provides is the ability to get and set its contents.

The contents of an item resides in a file in the item-store directory. The method `filename()` specifies its location. By defult, `filename()` is *dir*/*id,* where *dir* is the item-store directory and *id* is the item ID. However, specializations of `Item` may override the `filename()` method. A common convention is to group items into directories corresponding to their value for an indexed attribute. For example, pages are grouped by docid:

```
>>> p = corpus.pages.item(docid='111', number='23')
>>> p.filename()
'/tmp/corpus/pages/111/23'
```

Incidentally, for convenience in defining the `filename()` method, items also provide a method `dirname()` that returns the pathname of the item store directory.

The method `contents()` returns the contents of the file, appropriately parsed. The default implementation signals an error, and is appropriate for items that do not actually have contents. More commonly, specializations of `Item` override the `contents()` method to specify how the file is to be parsed. For example, `Page` just calls `load_string()`.

```
>>> p.contents()[:10]
'\nFORTUNE, '
```

Other specializations of `Item` may return more elaborate objects.

The method `set_contents()` takes an object of the sort that is returned by `contents()`, and saves it back into the file. Subsequent calls to `contents()` will return the revised contents.

When an item is deleted, the contents file is automatically deleted as well.

11.1.4 Connective IDs

Connections across items are created using various kinds of IDs. Document IDs (docids) connect items that represent different levels of processing of the same text. For example, in the previous section we examined page 23 of document 111. We can use the docid to get the corresponding bibliographic record from `cards`.

```
>>> card = corpus.cards.item(docid='111')
>>> card['author']
'R. F. Fortune'
>>> card['title']
'Arapesh'
```

Translation IDs (transids) identify items that are translations of each other. Items that have the same type and the same transid are **aligned**, meaning that their internal components line up one-to-one. For example, a segmented bitext is represented by a pair of segmented texts sharing a transid, and each segment of one is a translation of the corresponding segment of the other. Annotator IDs connect items annotated by the same person. Additional connective IDs may be introduced in the future.

11.1.5 Kernel

The kernel consists of items of class `KernelItem`. The basic item is a **text**. A document may contain multiple texts: a document may contain multiple languages, but a text is required to be monolingual.

A **parallel text** is a set of texts that are mutual translations of one another. An example is the Bible. To identify a unique text, it is not always sufficient to pair a `textid` with a `langid`. For example, there are multiple English translations of the Bible. Adding a `docid` does create a unique identifier. For example, the different English translations of the Bible correspond to different docids. (Note that the pairing of textid and docid is also insufficient: an interlinear Greek New Testament is a multilingual document contain both a Greek and an English version of the text.)

A **segmentation** is a list of sentences. The underlying text is identified by the langid, textid, and docid of the segmentation. It is possible to have multiple segmentations of the same text. They are distinguished by having differing values for `segid`. All items with a shared value for segid are segment-aligned translations of each other.

A **tokenization** is a list of tokenized sentences, each of which is a list of tokens. The underlying text is identified by the item's textid, langid, and docid, and the underlying segmentation is identified by adding the item's segid.

A **wordlist** is a list of senses, each sense being represented as a list of words. A parallel wordlist is uniquely identified by textid. All wordlists with a shared value for textid are sense-aligned translations of one another. The addition of langid and docid identifies a unique monolingual wordlist.

In sum, a kernel item has the following properties:

- `itemid`: a unique identifier for the item.
- `type`: one of `txt` (text), `snt` (segmentation), `spn` (segmentation spans), `tok` (tokenization), `lex` (wordlist).
- `langid`: represents the language of the item; external key for the `langs` item store.
- `textid`: represents a translationally-equivalent set of items. Items of type `txt` that share a value for `transid` are translations of one another. Wordlists that share a value for `textid` are sense-aligned translations of one other. For segmentations and tokenizations, the `textid` is used only as part of the triple (`textid`, `langid`, `docid`) used to identify the underlying text.
- `docid`: represents the bibliographic record for the item; it is an external key for the `cards` item store. It is an error to have multiple texts or multiple wordlists that share the same values for `textid`, `langid`, and `docid`.
- `segid`: represents a segmented text that may be rendered in multiple languages. Segmentations that share the same value for `segid` are segment-aligned translations of each other.

We give some examples drawn from the sample corpus. First, let us list the kernel items:

```
>>> print(corpus.kernel)
1 txt deu 117 bible
2 snt deu 117 bible bible.s
3 snt deu 117 bible bible.v
4 txt eng 118 bible
5 snt eng 118 bible bible.s
6 tok eng 118 bible bible.s
7 snt eng 118 bible bible.v
8 txt eng 119 bible
9 snt eng 119 bible bible.s
10 snt eng 119 bible bible.v
11 lex deu 120 swadesh
12 lex eng 120 swadesh
13 txt ape 114 genesis
14 txt sat 115 genesis
15 txt eng 116 genesis
16 spn eng 118 bible bible.s
```

Text. Item 4 is an example of a text:

```
>>> txt = corpus.kernel['4']
>>> print(txt)
itemid 4
type   txt
langid eng
docid  118
textid bible
segid
```

Note that document 118 is the King James Bible:

```
>>> corpus.cards['118']['title']
'The Holy Bible (King James Version)'
```

The contents of a text is a plain string:

```
>>> txt.contents()[:25]
'The First Book of Moses, '
```

Parallel text. The set of texts with the same textid constitute a parallel text.
(They are mutual translations.) For example:

```
>>> corpus.kernel.items(type='txt', textid='bible')
[<KernelItem 1 deu txt>, <KernelItem 4 eng txt>, <KernelItem 8 eng txt>]
>>> [item['langid'] for item in _]
['deu', 'eng', 'eng']
```

Segmentation. The contents of a segmentation is a list of strings.

```
>>> i5  = corpus.kernel['5']
>>> segs = i5.contents()
>>> segs[0]
'In the beginning, God created the heaven and the earth.'
>>> len(segs)
6
```

The original text is identified by the `textid`, `langid`, and `docid`:

```
>>> i5['textid']
'bible'
>>> i5['langid']
'eng'
>>> i5['docid']
'118'
>>> corpus.kernel.item(type='txt', textid='bible', langid='eng', docid='118')
<KernelItem 4 eng txt>
```

In this case, `type='txt', docid='118'` would suffice to identify a unique item.
But, in general, a document may contain multiple texts and multiple languages.

Spans. One can additionally associate segments with spans in the original text.

```
>>> spans = corpus.kernel['16'].contents()
>>> len(spans)
6
>>> spans[0]
(44, 99)
```

Aligned segments. Segmentations that share the same `segid` constitute segment-aligned translations.

```
>>> corpus.kernel.items(type='snt', segid='bible.s')
[<KernelItem 2 deu snt>, <KernelItem 5 eng snt>, <KernelItem 9 eng snt>]
```

Tokenization. The contents of a tokenization is a list of lists.

```
>>> i6 = corpus.kernel['6']
>>> tok = i6.contents()
>>> len(tok)
6
>>> tok[0][:5]
[<In>, <the>, <beginning,>, <God>, <created>]
```

Wordlist. The contents of a wordlist is a list of lists.

```
>>> i11 = corpus.kernel['11']
>>> wl = i11.contents()
>>> len(wl)
10
>>> wl[0]
['ich']
>>> wl[1]
['Du', 'Sie']
```

Wordlists that share a common `textid` are aligned.

```
>>> corpus.kernel.items(type='lex', textid='swadesh')
[<KernelItem 11 deu lex>, <KernelItem 12 eng lex>]
```

Part IV

Trees and Treebanks

Chapter 12

Trees: `seal.tree`

This chapter documents the module `seal.tree`. The examples assume that one has done:

```
>>> from seal.tree import *
>>> from seal.io import contents
```

12.1 Node attributes

The nodes of a tree are represented by instances of the class `Tree`. There is no separate node class: a node and the tree rooted at the node are both represented by a `Tree` instance.

A `Tree` instance has eight attributes: `word`, `children`, `nld`, `role`, `cat`, `sem`, `id`, and `parent`.

12.1.1 Basic node types

We wish to accommodate the nodes that occur in three kinds of trees: unheaded phrase-structure trees, headed phrase-structure trees, and dependency trees. The attributes `word`, `children`, and `role` are used to distinguish five basic node types as summarized in the following table. The head of a node is defined to be a child whose role is `'head'`.

			word	
	children	**head**	**N**	**Y**
leaf	**N**		empty leaf	leaf word
interior	**Y**	**Y**	headed phrase	governor
		N	unheaded phrase	

For governors and leaf nodes, there is no distinction between headed and unheaded. For the former, a child with role `'head'` has no special status, and for the latter, there are no children.

Interior and leaf. The `children` attribute distinguishes between *interior* nodes and *leaf* nodes. The former have children; the latter do not. Governors and phrasal nodes are interior nodes.

Terminal and nonterminal. The interior-leaf distinction is *not* the same as the terminal-nonterminal distinction. The latter is a property of categories, as determined by a grammar. Terminal categories are not allowed to appear on the lefthand side of a rewrite rule, whereas nonterminal categories that are not useless appear on the lefthand side of at least one rewrite rule. It is possible to have leaf nodes with nonterminal categories: such nodes are *null expansions.* A tree generated by a constituent-structure grammar cannot have interior nodes labeled with terminal categories, but dependency grammars make no terminal-nonterminal distinction, and permit trees in which interior nodes are labeled with parts of speech, which would be terminal categories in a constituent-structure grammar.

Leaf words and empty leaves. The `word` attribute distinguishes between leaf words and empty leaves. The former have a non-null value for `word`, and the latter do not. Note that the expression *leaf word* is not redundant in a dependency tree: leaf words contrast with *governors,* which are interior nodes with a value for `word`. However, in a constituency tree, only leaf nodes have values for `word`, so in that context we can refer to leaf words simply as *words.*

Depending on the kind of category it has, an empty leaf may represent either a null terminal (like an empty complementizer) or a null expansion (corresponding to a rewrite rule with nothing on the right-hand side). We are careful not to refer to null terminals as "words," reserving the term *word* for a node with a non-null value for `word`.

Governor versus phrase. The `word` attribute also distinguishes between governors and phrases. Both are interior nodes; the former has a non-null value for `word`, while the latter does not. That is, a *governor* is a node that has non-null values for both `children` and `word`. Governors are used in dependency trees; their children are called *dependents.* By contrast, a *phrase* or *phrasal node* has children but no word. Phrasal nodes are used in constituency trees.

Heads. We further subdivide phrasal nodes according to whether they have heads or not. The head of a phrasal node is defined to be a child whose `role` is "`head`." A phrasal node with a head is a headed phrase, and a phrasal node without a head is an unheaded phrase. Governors and leaves are by definition headless.

12.1.2 Other attributes: `nld`, `parent`, `cat`, `role`, `id`, `sem`

Number of left dependents. The `nld` attribute is relevant only for governors. It indicates the number of left dependents. In the terminal string, the

governor is ordered after its left dependents and before its right dependents. If a phrasal node or leaf has a value for `nld`, the value is ignored.

Parent. The `parent` attribute is set by the function `set_parents()`. It permits one to navigate not only down a tree, but also back up again.

Cat. The `cat` attribute represents the syntactic category of the tree. The category may be anything, though strings and `Category` instances are the commonest choices.

Role. The links in a dependency tree are often labeled. The link label indicates the relationship between governor and dependent, such as "subject" or "object." The same relationship can be useful in constituent trees as indicating the **role** of a child relative to its parent (or the head of its parent).

As already mentioned, the role "`head`" has a special status if the parent is a phrasal node.

ID. Nodes are sometimes assigned identifiers, such as the indices used to encode movement relations or control.

Sem. The value for `sem` is the semantic translation of the node.

12.1.3 Example

Here is an example of constructing a tree manually, by constructing individual nodes. The first two arguments to the constructor are the category and a list of children. A word may be specified using the keyword "`word`."

```
>>> det = Tree('Det', word='the')
>>> n = Tree('N', word='dog')
>>> np = Tree('NP', [det, n])
```

Here are examples for the three main attributes.

```
>>> np.cat
'NP'
>>> np.children
[<Tree Det the>, <Tree N dog>]
>>> np.word
>>> det.word
'the'
```

One can set `role` and `id`:

```
>>> det.role = 'spec'
>>> n.role = 'head'
>>> np.id = 1
```

The `nld` attribute is only relevant for dependency trees: see below under "Dependents."

One can print out the tree rooted at a node using a `print` statement:

```
>>> print(np)
0    (NP &1
1       (Det:spec the)
2       (N:head dog))
```

Notice that nodes are numbered. One can access them directly by number:

```
>>> np[2]
<Tree N dog>
```

This is particularly useful for large trees.

12.1.4 Copy

The method `copy()` makes a shallow copy of a node. If the original node has children, a fresh copy of the child list is made, but the child nodes themselves are not copied.

```
>>> y = np.copy()
>>> y is np
False
>>> y.children is np.children
False
>>> y.children == np.children
True
>>> y.children[0] is np.children[0]
True
```

One can modify any of the attributes `cat`, `children`, `word`, `nld`, `role`, or `id` when making the copy.

```
>>> z = np.copy(children=[])
>>> print(z)
0    (NP &1)
```

12.2 Node functions

The `Tree` class has few methods. Instead, there is a large collection of functions that are intended to work with any object (though not all of them are fully general yet).

12.2.1 Accessors

Instead of using the attributes directly, it is best to use the accessor functions `getword()`, `getchildren()`, `getnld()`, `getrole()`, `getcat()`, `getsem()`, `getid()`, and `getparent()`. These functions can be applied to arbitrary objects, not just `Tree` instances. If called on something that lacks the attribute in question, they return `None`. There is one exception: if a string is passed to `getword()`, it returns the string itself. (Hence a string behaves like a leaf node that has a value for `word` but has no category.)

Some examples:

```
>>> getcat(np)
'NP'
>>> getcat('hi')
>>> getword(det)
'the'
>>> getword('hi')
'hi'
```

12.2.2 Predicates

Basic predicates. The following functions are available to test for properties of a node: `is_interior()`, `is_leaf()`, `is_governor()`, `is_phrase()`, `is_headed_phrase()`, `is_unheaded_phrase()`, `is_leaf_word()`, `is_empty_leaf()`, They have all been previously discussed. Some examples:

```
>>> is_interior('hi')
False
>>> is_leaf(det)
True
>>> is_leaf('hi')
True
>>> is_headed_phrase(np)
True
```

Is empty. The function `is_empty()` tests whether a node is empty or not. This is technically not a property of the node itself, but of the tree rooted at the node: a node is empty just in case neither it nor any of its descendants has a value for `word`.

```
>>> is_empty(Tree())
True
>>> is_empty(Tree('NP', [Tree('N')]))
True
>>> is_empty(Tree('NP', [Tree('N', word='dog')]))
False
```

Is unary. The function `is_unary()` returns true just in case the node has exactly one child.

```
>>> is_unary(np)
False
>>> is_unary(det)
False
>>> is_unary(Tree('NP', [Tree('N', word='rice')]))
True
```

Node type. The function `nodetype()` returns one of the following: `'leaf'`, `'governor'`, `'unheaded phrase'`, or `'headed phrase'`.

```
>>> nodetype(np)
'headed phrase'
>>> nodetype(det)
'leaf'
>>> nodetype('hi')
'leaf'
```

12.2.3 Structural access

Head child. The function `head_child()` returns the child whose role is "`head`," if one exists. (If there is more than one, it returns only the first.)

```
>>> head_child(np)
<Tree N dog>
```

Head index. The function `head_index()` returns the head child's index in the `children` list. It returns -1 if there is no head child. Children are numbered from 0.

```
>>> head_index(np)
1
>>> head_index('hi')
-1
```

Child index. The function `child_index()` takes two arguments, parent and child, and returns the index of the child in the parent's `children` list. It returns -1 if the child is not found.

```
>>> child_index(np, det)
0
>>> child_index(np, 'foo')
-1
```

Dependents. If the node has a value for `nld`, the function `left_dependents()` returns all children up to, but not including, `nld`. The function `right_dependents()` returns all remaining children. If the node has no value for `nld`, but it does have a head child, then `left_dependents()` returns all children preceding the head child, and `right_dependents()` returns all children following the head child. If the node has neither `nld` nor a head child, both functions signal an error.

```
>>> left_dependents(np)
[<Tree Det the>]
>>> right_dependents(np)
[]
>>> sbj = Tree('N', word='dogs')
>>> v = Tree('V', word='chase')
>>> obj = Tree('N', word='cats')
>>> v.children = [sbj, obj]
>>> v.nld = 1
>>> left_dependents(v)
[<Tree N dogs>]
>>> right_dependents(v)
[<Tree N cats>]
```

Expansion. If a node has children, the function `expansion()` returns a tuple consisting of the node's category followed by the categories of its children. Some of the categories may be `None`. If the node has no children, the return value is `None`.

```
>>> expansion(np)
('NP', 'Det', 'N')
```

12.2.4 Destructive

The function `delete_child()` takes a node and a child index, and deletes the child at that index. The value for `nld` is adjusted, if necessary. There is no return value.

```
>>> delete_child(v,0)
>>> left_dependents(v)
[]
>>> right_dependents(v)
[<Tree N cats>]
```

12.3 Trees

12.3.1 Tree types

The type of a tree is defined by the type of interior nodes it contains.

- A tree is an unheaded phrase-structure tree if all its interior nodes are unheaded phrasal nodes.
- A tree is a headed phrase-structure tree if all its interior nodes are headed phrasal nodes.
- A tree is a dependency tree if all its interior nodes are governors.

A hybrid tree is one that satisfies none of these three definitions.

All three types of tree contain identical leaf nodes. They differ only in their interior nodes. Technically, a tree containing no interior nodes (i.e., consisting of a single terminal node) satisfies all three definitions.

The following functions test tree types:

```
>>> is_headed_tree(np)
True
>>> is_unheaded_tree(np)
False
>>> is_dependency_tree(v)
True
```

The function `treetype()` returns the tree type: `'headed phrase'`, `'unheaded phrase'`, or `'governor'` (the lattermost for a dependency tree). It returns `'leaf'` if the tree consists of a single leaf node, and `None` if the tree is hybrid.

```
>>> treetype(np)
'headed phrase'
>>> treetype(v)
'governor'
>>> treetype(det)
'leaf'
```

12.3.2 Load and parse

Iter trees. The function `iter_trees()` reads trees in a lisp-like format from a file or string. Like all the load/save functions (Chapter 5), `iter_trees()` takes its argument to name a file if it is a `Fn`, and to provide the contents, if it is a string. Here is an example:

```
>>> ts = iter_trees(ex.tree2)
>>> next(ts)
<Tree S ...>
>>> print(_)
0    (S
1       (NP:subj &1
           foo
2          (Det the)
3          (N dog))
```

```
4       (VP:head &2
5          (V chased)
6          (NP:dobj
7             (Det the)
8             (N cat))))
```

Load and parse. The function `load_trees()` simply returns

```
list(iter_trees(fn))
```

The function `parse_trees()` also dispatches to `iter_trees()`, but it wraps its arguments in a `Contents` instance (from `seal.io`) so that the argument is interpreted as a string representing a tree, rather than a filename. The function `parse_tree()` returns a single tree instead of a list of trees; it signals an error if its argument does not parse as a single tree.

```
>>> foo = parse_tree('''
...     (NP:subj&1 foo
...         (Det the)
...         (N:head dog))
... ''')
>>> print(foo)
0    (NP:subj &1
        foo
1       (Det the)
2       (N:head dog))
```

12.3.3 Print and save

Pretty-print string. The function `tree_string()` takes a tree and returns a pretty-printed string.

```
>>> tree_string(foo)
'0    (NP:subj &1\n        foo\n1       (Det the)\n2       (N:head dog))'
>>> print(_)
0    (NP:subj &1
        foo
1       (Det the)
2       (N:head dog))
```

The `Tree` method `__str__()` simply dispatches to `tree_string()`.

One can suppress the node numbers by specifying `numerate=False`.

```
>>> print(tree_string(foo, numerate=False))
(NP:subj &1
   foo
   (Det the)
   (N:head dog))
```

The string-valued attributes (`word`, `cat`, `role`, `id`) are formatted without surrounding quotes unless they contain whitespace or one of the following special characters: left or right parenthesis, left or right square bracket, colon, or single or double quote. For example:

```
>>> print(Tree('Multi\x20N', id='a/b', word='hors d\'oeuvre'))
0    ('Multi N' "hors d'oeuvre" &a/b)
```

Save trees. The function `save_trees()` takes an iterator over trees and a filename, and prints the trees to the named file.

```
>>> fn = tmpfile()
>>> save_trees([foo], fn)
>>> print(load_string(fn), end='')
(NP:subj &1
   foo
   (Det the)
   (N:head dog))
```

As with all of the "save" functions, the filename is option; if omitted, it collects and returns its output as a string.

```
>>> s = save_trees([foo])
>>> print(s, end='')
(NP:subj &1
   foo
   (Det the)
   (N:head dog))
```

12.3.4 Tabular tree files

There is also a tabular format for representing trees in a file. An example is provided by the file `ex.t1`, whose contents are:

```
>>> print(contents(ex.t1), end='')
[       S
[       NP
+       Det     the
+       N       cat
]
[       VP
+       V       chased
[       NP
+       Det     the
+       N       dog
]
]
]
```

A record may contain up to six fields:

1	Record type	Left bracket for the beginning of a nonterminal node, right bracket for the end of a nonterminal node, and plus for a terminal node.
2	Category	Syntactic category.
3	Word	It may not contain a tab or newline, but any other character (including space) is allowed.
4	Role	A symbol representing the relation between the node and its parent or governor.
5	Head	A numeric index, identifying either a particular child, or a position among the children.
6	ID	A numeric index for the node.

None of the fields is obligatory. Additional fields beyond these six are also permitted, but ignored.

The function `iter_tabular_trees()` can be used to read a file in tabular tree format. It is a generator over trees:

```
>>> t1 = next(iter_tabular_trees(ex.t1))
>>> print(t1)
0    (S
1       (NP
2          (Det the)
3          (N cat))
4       (VP
5          (V chased)
6          (NP
7             (Det the)
8             (N dog))))
```

The function `load_tabular_trees()` converts the generator into a list.

Conversely, the function `save_tabular_trees()` takes a tree iterator and a filename, and saves the trees in tabular format.

```
>>> fn = tmpfile()
>>> save_tabular_trees([foo], fn)
>>> for line in open(fn): print(line, end='')
...
[        NP        foo        subj        0        1
+        Det        the
+        N        dog        head
]
```

12.3.5 Drawing

The function `draw_tree()` draws a tree. It requires the package "graphviz" to be installed. It takes a second argument, which is the filename to write. If the filename is omitted, a temp file is written and displayed to the screen.

12.4 Tree iterations

If one iterates directly over a tree, one iterates over its nodes in preorder.

12.4.1 Preorder and text order walks

A walk is an iteration over the nodes of a tree. Two different walks are defined: `preorder()` and `textorder()`. In a preorder walk, one visits a node before any of its children. For phrase-structure trees, text order is identical to preorder, but for dependency trees, they differ. More precisely, in a text-order walk, any node that has a value for `nld` is visited after visiting its left dependents, but before visiting its right dependents.

To illustrate, we create two trees. The first is a headed phrasal tree:

```
>>> h = parse_tree('''(S (NP (Det the) (N:head dog))
...                      (VP:head (V:head barked))
...                      (Adv loudly))''')
```

The second is a dependency tree:

```
>>> d = parse_tree('(V (N (Det the) dog) barked (Adv loudly))')
```

We can confirm that the preorder and text order walks for the phrasal tree are the same:

```
>>> for node in preorder(h): print(repr(node))
...
<Tree S ...>
<Tree NP ...>
<Tree Det the>
<Tree N dog>
<Tree VP ...>
<Tree V barked>
<Tree Adv loudly>
>>> for node in textorder(h): print(repr(node))
...
<Tree S ...>
<Tree NP ...>
<Tree Det the>
<Tree N dog>
<Tree VP ...>
<Tree V barked>
<Tree Adv loudly>
```

But they differ for the dependency tree:

```
>>> for node in preorder(d): print(repr(node))
...
<Tree V barked ...>
<Tree N dog ...>
<Tree Det the>
<Tree Adv loudly>
>>> for node in textorder(d): print(repr(node))
...
<Tree Det the>
<Tree N dog ...>
<Tree V barked ...>
<Tree Adv loudly>
```

12.4.2 Nodes and edges

The `__iter__()` method of a tree, and the function `iter_nodes()`, are both synonyms for `preorder()`. The function `nodes()` turns the generator into a list.

An *edge* is a pair (p, c) where p is a parent node and c is one of its children. The function `iter_edges()` returns an iteration over the edges in a tree. The function `edges()` turns the iteration into a list.

12.4.3 Subtrees

Subtrees, iter subtrees. The function `iter_subtrees()` returns an iteration over subtrees that satisfy a given predicate. The function `subtrees()` turns the iteration into a list.

The difference between these functions and simply filtering the output of `nodes()` is that `iter_subtrees()` terminates the recursion whenever it finds a node matching the predicate.

```
>>> subtrees(h, lambda x: is_leaf(head_child(x)))
[<Tree NP ...>, <Tree VP ...>]
```

If the second argument is a string, it is taken to be the desired node category.

```
>>> subtrees(h, 'Adv')
[<Tree Adv loudly>]
```

Subtree. The function `subtree()` takes the list produced by `subtrees()` and returns its member, if there is exactly one. It signals an error if the list is not a singleton list.

12.4.4 Paths and leaves

Paths. The function `paths()` returns the list of paths through the tree. A path is represented by a string in which node categories are separted by "/." The categories in the path are ordered from root to leaf.

```
>>> paths(h)
['S/NP/Det', 'S/NP/N', 'S/VP/V', 'S/Adv']
```

Leaves. The function `leaves()` returns the list of leaf nodes in a tree. The leaves are listed in preorder.

Words. The function `words()` differs from `leaves()` in two ways: it only includes leaves that have a value for `word`, and it uses a text-order walk.

```
>>> words(h)
['the', 'dog', 'barked', 'loudly']
```

Tagged words. The function `tagged_words()` is like `words()`, except that it produces a list of pairs of form (*word, cat*).

```
>>> tagged_words(h)
[('the', 'Det'), ('dog', 'N'), ('barked', 'V'), ('loudly', 'Adv')]
```

Terminal string. The function `terminal_string()` takes the output of `words()` and turns it into a string. The words are separated by spaces.

```
>>> terminal_string(h)
'the dog barked loudly'
```

12.4.5 Predicates

Is e-free. The function `is_efree_tree()` returns true just in case the tree contains no empty nodes.

Is unary-free. The function `is_unaryfree_tree()` returns true just in case there are no unary-branching nodes in the tree.

12.4.6 Copy tree

The function `copy_tree()` does a deep copy of a tree. Unlike the node method `copy()`, `copy_tree()` does recurse through the whole tree, making copies of all nodes.

12.4.7 Transformations

The operations described in this section, as well as the transformations described in the chapters on head-marking and stemma conversion, are destructive. To protect a tree, make a copy before applying destructive operations.

12.4.8 Delete nodes

The function `delete_nodes()` deletes all nodes with a given category. (However, it never deletes the root node.)

Eliminate epsilons. The function `eliminate_epsilons()` eliminates all empty nodes from a tree. If the tree was initially headed, any heads that are empty will get deleted.

```
>>> e = parse_tree('''
...   (S
...     (NP (N ))
...     (VP
...       (VBZ )
...       (RB surely)
...       (NP Fido)))
... ''')
>>> eliminate_epsilons(e)
>>> print(e)
0    (S
1       (VP
2          (RB surely)
3          (NP Fido)))
```

Set parents, get root. The function `set_parents()` destructively adds a `parent` attribute to every node in the tree, pointing back to the node's parent.

After parents have been set in a tree, one can use the function `getroot()` to go from any node to the root node. It follows parent links up the tree to the root node.

12.5 Tree builder

A `TreeBuilder` is a stack-like data structure for constructing a tree. Here is an example of use:

```
>>> tb = TreeBuilder()
>>> tb.start('NP')
<Tree NP>
>>> tb = TreeBuilder()
>>> tb.start('S')
<Tree S>
>>> tb.start('NP', role='subj')
<Tree NP>
>>> tb.leaf('Det', 'the')
<Tree Det the>
>>> tb.leaf('N', 'dog')
```

```
<Tree N dog>
>>> tb.end()
<Tree NP ...>
>>> tb.start('VP', role='head')
<Tree VP>
>>> tb.leaf('V', 'barks', role='head')
<Tree V barks>
>>> tb.end()
<Tree VP ...>
>>> tb.end()
<Tree S ...>
>>> tb.tree()
<Tree S ...>
>>> print(_)
0    (S
1       (NP:subj
2          (Det the)
3          (N dog))
4       (VP:head
5          (V:head barks)))
```

The methods for building a phrasal node are `start()` and `end()`. Both return the node.

To build a dependency node, one also uses the method `middle()` to mark the position at which the governor occurs. For example:

```
>>> tb.start('V', word='chase')
<Tree V chase>
>>> tb.leaf('N', 'dogs')
<Tree N dogs>
>>> tb.middle()
>>> tb.leaf('N', 'cats')
<Tree N cats>
>>> tb.end()
<Tree V chase ...>
>>> tb.tree()
<Tree V chase ...>
>>> print(_)
0    (V
1       (N dogs)
        chase
2       (N cats))
```

The builder allows one to construct multiple trees; it saves them on a list until one calls either `tree()` or `trees()`. The latter returns the list of trees constructed. The former returns a single tree, and signals an error if there is not exactly one tree on the list. Both methods signal an error if there is an

incomplete tree in progress. Both methods restore the builder to its empty state.

12.6 Summary

The following tables summarize the attributes and methods of `Tree`, as well as the functions in `seal.tree`.

`cat`	Syntactic category.
`children`	A list of children, possibly `[]` or `None`.
`id`	An identifier (can be anything), or `None`.
`nld`	The number of left dependents (for a governor node).
`role`	Role with respect to parent or governor.
`word`	The word associated with this node.
`sem`	The semantic translation.
`parent`	Only if `add_parents()` has been called.

Table 12.1: Attributes of `Tree`

`Tree(...)`	Constructor: cat, children, word.
`print(t)`	Pretty-print
`for n in t`	Iteration: preorder
`t[i]`	Direct access of nodes
`t.copy()`	Shallow copy

Table 12.2: Attributes and methods of `Tree`

Accessors
getcat(t)
getword(t)
head_child(t)
head_index(t)
child_index(t,c)
left_dependents(t)
right_dependents(t)
expansion(t)
getroot(t)
preorder(t)
textorder(t)
iter_nodes(t)
nodes(t)
iter_subtrees(t,P)
subtrees(t,P)
paths(t)
leaves(t)
words(t)
tagged_words(t)
terminal_string(t)
tree_string(t)
print_tree(t)
copy_tree(t)

Destructive
delete_child(t,i)
delete_nodes(t,cat)
eliminate_epsilons(t)
set_parents(t)

Predicates
nodetype(t)
is_interior(t)
is_leaf(t)
is_governor(t)
is_phrase(t)
is_headed_phrase(t)
is_unheaded_phrase(t)
is_leaf_word(t)
is_empty_leaf(t)
is_empty(t)
is_unary(t)
treetype(t)
is_headed_tree(t)
is_unheaded_tree(t)
is_dependency_tree(t)
is_efree_tree(t)
is_unaryfree_tree(t)

Files
iter_trees(s)
load_trees(s)
parse_tree(s)
save_trees(ts,fn)
iter_tabular_trees(fn)
load_tabular_trees(fn)
save_tabular_trees(ts,fn)
draw_tree(t)

Table 12.3: Functions in seal.tree

Chapter 13

Head marking: `seal.head`

This chapter documents the module `seal.head`. The examples assume you have done:

```
>>> from seal.head import *
>>> from seal.tree import parse_tree, copy_tree
```

13.1 Head rules

13.1.1 Mark heads

The function `mark_heads()` destructively converts an unheaded tree into a headed tree. Note that it is called for side effect; it does not return a value.

```
>>> u = parse_tree('''(S (NP (DT the) (NN dog))
...                       (VP (VB chases)
...                           (NP (DT a) (NN cat))))''')
>>> h = copy_tree(u)
>>> mark_heads(h)
>>> print(h)
0    (S
1       (NP
2          (DT the)
3          (NN:head dog))
4       (VP:head
5          (VB:head chases)
6          (NP
7             (DT a)
8             (NN:head cat))))
```

Nodes that already have heads are left unchanged: `mark_heads()` only determines heads for headless nodes.

Head-marking uses a set of head rules. The function `mark_heads()` will accept a rule set as second argument. By default, it uses `DefaultHeadRules`. These represent a modified version of the rules in Michael Collins's thesis. The original rules are also available as `CollinsMagermanRules`.

13.1.2 Find head

The main work is done by the function `find_head()`. It takes a node and indicates which of its children should be head. The return value is the index of the predicted head node.

```
>>> print(u)
0    (S
1       (NP
2          (DT the)
3          (NN dog))
4       (VP
5          (VB chases)
6          (NP
7             (DT a)
8             (NN cat))))
>>> find_head(u[0])
1
>>> find_head(u[4])
0
```

For debugging, one can turn tracing on:

```
>>> find_head(u[4], trace=True)
Rule <Rule VP 0>, found VB: h=0
0
```

One can print the head rules:

```
>>> print(DefaultHeadRules)
Non-head cats: `` CO CC , . '' :
NAC
    0 L: NN NNS NNP NNPS NP NAC EX $ CD QP PRP VBG JJ JJS JJR ADJP FW
SBAR
    0 L: WHNP WHPP WHADVP WHADJP IN DT S SQ SINV SBAR FRAG
...
NP
    0 R: NN NNP NNPS NNS NX POS JJR
    1 L: NP
    2 R: $ ADJP PRN
    3 R: CD
    4 R: JJ JJS RB QP
```

13.1.3 The Magerman-Collins head rules

As mentioned above, the head-marking rules are an adaptation of the Magerman-Collins rules, as given in Collins's dissertation [1], pp. 238 ff. Collins adapts them from Magerman [1331]. The rules from Collins's dissertation are listed in table 13.1. Their use is best explained with an example. Suppose the parent node has category `CONJP`. The rule for `CONJP` has direction "`R`" and child categories `CC RB IN`. One looks first for the rightmost child with category `CC`. If none is found, look for the rightmost child with category `RB`. If none is found, look for the rightmost child with category `IN`. As a default, take the rightmost child as head.

If the parent category is `NP`, the following rules are used. The first one that matches determines the head.

- The rightmost child that is a terminal node, if its category is `POS`,
- The rightmost child that is one of: `NN NNP NNPS NNS NX POS JJR`,
- The leftmost child that is `NP`,
- The rightmost child that is one of: `$ ADJP PRN`,
- The rightmost child that is `CD`,
- The rightmost child that is one of: `JJ JJS RB QP`,
- The rightmost child that is a terminal node.

These rules may fail if the parent category is not listed, or if an NP contains no terminal node. In either of those cases, Collins specifies no action, but one presumably takes the rightmost child as head.

Finally, there is an adjustment rule that applies in the case of coordination.

- If the head is immediately preceded by `CC`, and there is another child before the `CC`, then that other child becomes head.

13.2 Decoordination

Head-marking is problematic in coordination structures, because, in the usual view, all the coordinands have an equal claim to being head. If desired, one can break the symmetry in coordination structures, before head marking, by calling the function `decoordinate()`.

The function `decoordinate()` replaces all coordinate structures with single-headed structures, in which the first coordinand is left in place, but other coordinands are wrapped in a new adjunct node with category "`CO`" and role "`co`." The replacement is destructive. Here is an example:

ADJP	L	NNS QP NN $ ADVP JJ VBN VBG ADJP JJR NP JJS DT FW RBR RBS SBAR RB
ADVP	R	RB RBR RBS FW ADVP TO CD JJR JJ IN NP JJS NN
CONJP	R	CC RB IN
FRAG	R	
INTJ	L	
LST	R	LS :
NAC	L	NN NNS NNP NNPS NP NAC EX $ CD QP PRP VBG JJ JJS JJR ADJP FW
PP	R	IN TO VBG VBN RP FW
PRN	L	
PRT	R	RP
QP	L	$ IN NNS NN JJ RB DT CD NCD QP JJR JJS
RRC	R	VP NP ADVP ADJP PP
S	L	TO IN VP S SBAR ADJP UCP NP
SBAR	L	WHNP WHPP WHADVP WHADJP IN DT S SQ SINV SBAR FRAG
SBARQ	L	SQ S SINV SBARQ FRAG
SINV	L	VBZ VBD VBP VB MD VP S SINV ADJP NP
SQ	L	VBZ VBD VBP VB MD VP SQ
UCP	R	
VP	L	TO VBD VBN MD VBZ VB VBG VBP VP ADJP NN NNS NP
WHADJP	L	CC WRB JJ ADJP
WHADVP	R	CC WRB
WHNP	L	WDT WP WP$ WHADJP WHPP WHNP
WHPP	R	IN TO FW

Table 13.1: The Collins-Magerman head-marking rules.

```
>>> t = parse_tree('''(NP (N trains)
...                       (',' ',') (N planes)
...                       (',' ',') (CC and) (N autos))''')
>>> decoordinate(t)
>>> print(t)
0    (NP
1       (N trains)
2       (CO:co
3          (, ,)
4          (N planes)
5          (, ,)
6          (CC and)
7          (N:head autos)))
```

Chapter 14

Dependency conversion: `seal.dep`

This chapter documents the module `seal.dep`. The examples assume you have done:

```
>>> from seal.dep import *
>>> from seal.tree import parse_tree
```

14.1 Dependency conversion

The toplevel function is `convert()`. It takes optional arguments giving the type of `input` and the type of `output`. By default, `input` is `'tree'` and `output` is `'efstemma'`.

```
>>> t = parse_tree('(S (NP (Pron this))'
...                '   (VP (VBZ is)'
...                '       (NP (DT a) (NN test))))')
...
>>> print(convert(t))
0 *root* None None None None
1 this   Pron None None 2
2 is     VBZ  None root 0
3 a      DT   None None 4
4 test   NN   None None 2
```

To see how heads are assigned, one can specify `'headed'` output:

```
>>> print(convert(t, output='headed'))
0   (S
1      (NP
2         (Pron:head this))
```

```
3      (VP:head
4         (VBZ:head is)
5         (NP
6            (DT a)
7            (NN:head test))))
```

Or if one prefers a dependency tree to a stemma:

```
>>> print(convert(t, output='dep'))
0   (VBZ:root
1      (Pron this)
          is
2      (NN
3         (DT a)
             test))
```

The legal types input and output types are:

- `'tree'` for an unheaded constituency tree,
- `'headed'` for a headed constituency tree,
- `'dep'` for a dependency tree,
- `'stemma'` for a `Sentence` possibly containing empty words,
- `'efstemma'` for an ϵ-free stemma.

These reflect the steps of the conversion: `mark_heads()` converts an unheaded tree to a headed tree, `dependency_tree()` converts a headed tree to a dependency tree, `stemma()` converts a dependency tree to a stemma, and `eliminate_epsilons()` eliminates empty words.

All steps except the first are non-destructive. If given an unheaded tree as input, `convert()` makes a copy before calling `mark_heads()`, unless the keyword argument `destructive=True` is provided.

The keyword arguments `projections` and `reductions` may optionally be provided; they are passed directly to `dependency_tree()`.

14.2 Dependency tree

14.2.1 Usage

The central function provided by `seal.dep` is `dependency_tree()`, which converts a headed phrase-structure tree to a dependency tree. (It signals an error if it encounters a headless node.)

```
>>> h = parse_tree('''
...     (S (NP:subj (Det the) (N:head dog))
...        (VP:head (V:head chased)
```

```
...                  (NP:obj (Det a) (N:head cat)))
...         (Adv:mod quickly))
... ''')
>>> d = dependency_tree(h)
>>> print(d)
0    (V:root
1       (N:subj
2          (Det the)
           dog)
        chased
3       (N:obj
4          (Det a)
           cat)
5       (Adv:mod quickly))
```

The function `dependency_tree()` takes two keyword arguments: `projections` and `reductions`. They are passed directly to the `tree()` method of `Projection`, which is discussed below.

It should be noted that the dependency tree may contain empty nodes. The conversion treats all terminal nodes alike, whether they have a string or `None` as their value for `.word`.

14.2.2 Projections

The `dependency_tree()` function works by converting the tree first to its *projections,* where a projection is defined as a list of nodes, each being the head of the previous. There is one projection for each leaf node. For example, in the tree h above, "the" has projection (Det), "dog" has projection (NP, N), "chased" has projection (S, VP, V), "a" has projection (Det), "cat" has projection (NP, N), and "quickly" has projection (Adv).

The left dependents of a projection are defined to be the concatenation of left dependents of the nodes it contains, from outermost to innermost. The right dependents are defined to be the concatenation of the right dependents of the nodes, from innermost to outermost. For example, the only left dependent of (S, NP, V) is the subject NP, and its right dependents are the object NP and the adverb.

The class `Projection` represents a projection. One creates a projection from a headed tree:

```
>>> p = Projection(h)
```

This actually creates projections recursively for the entire tree.

Nodes. The value of attribute `nodes` is the list of nodes that make up the projection:

```
>>> p.nodes
[<Tree S ...>, <Tree VP ...>, <Tree V chased>]
```

Ldeps, rdeps. The attributes `ldeps` and `rdeps` contain the left and right dependents, converted to projections:

```
>>> p.ldeps
[<Projection NP N dog>]
>>> p.rdeps
[<Projection NP N cat>, <Projection Adv quickly>]
```

Lr, parent, headsib. Each non-root projection has values for `lr`, `parent`, and `headsib`, representing the configuration in which the root node occurs in the original tree. This configuration is called the "reduction" represented by attaching the root of projection to its parent. For example, the projection for the subject NP occurs as a left dependent in S, with head child VP. Accordingly:

```
>>> sp = p.ldeps[0]
>>> sp.lr
'L'
>>> sp.parent
<Tree S ...>
>>> sp.headsib
<Tree VP ...>
```

(For the root projection, all three attributes have the value `None`.)

Tree. The method `tree()` converts a projection into a dependency tree. By default, the category of a projection is taken to be the part of speech of the head node (that is, `nodes[-1]`.cat), and the role is the role (if any) of the root node (that is, `nodes[0].role`).

There are two boolean keyword arguments that can be used to select alternative definitions of category and role. If `projections` is true, then the category is the concatenation of all categories in the projection. For example:

```
>>> print(p.tree(projections=True))
0    (S_VP_V:root
1       (NP_N:subj
2          (Det the)
           dog)
        chased
3       (NP_N:obj
4          (Det a)
           cat)
5       (Adv:mod quickly))
```

If `reductions` is true, then the role is represented by a `Reduction` object, which prints out as the concatenation of `lr`, `nodes[0].cat`, `parent.cat`, and `headsib.cat`. For example:

```
>>> print(p.tree(reductions=True))
0    (V:root
1       (N:'L_NP:subj_S_VP'
2          (Det:L_Det_NP_N the)
           dog)
        chased
3       (N:'R_NP:obj_VP_V'
4          (Det:L_Det_NP_N a)
           cat)
5       (Adv:'R_Adv:mod_S_VP' quickly))
```

One can specify both `projections` and `reductions`, if desired.

14.2.3 Reduction

The class `Reduction` represents the configuration, in the original headed phrase structure tree, in which a dependent occurs. It has four attributes:

- `lr` may be "`L`," for a dependent that precedes its head sibling, or "`R`," for one that follows, or "`root`," for the root node.
- `dep` is the category of the dependent.
- `parent` is the category of the parent node.
- `head` is the category of the head sibling.

14.3 Stemmas and governor arrays

14.3.1 `Word` and `Sentence`

A dependency stemma is represented by a `Sentence` instance, which contains `Word` instances representing the individual words of the sentence. A `Sentence` may itself have an `index()`, which is intended to represent its position in a collection of sentences such as a treebank. Otherwise, a `Sentence` is simply a list of `Word` instances. The word at position 0 is a pseudo-word representing the root.

To create a sentence with a known number of words, use `make_sentence()`:

```
>>> s = make_sentence(4, index='test')
>>> s[1].form = 'This'
>>> s[2].form = 'is'
>>> s[3].form = 'a'
>>> s[4].form = 'test'
>>> print(s)
0 *root* None None None None
1 This   None None None 0
2 is     None None None 0
```

```
10    3 a      None None None 0
11    4 test   None None None 0
```

The methods of `Sentence` are as follows:

s`.index()`	returns the index of the sentence.
s`.providence()`	returns the index as a string, or `None`.
`len(`*s*`)`	includes the root pseudo-word.
`iter(`*s*`)`	iterates over all words, including the root pseudo-word.
s`[`*i*`]`	returns the *i*-th word; the root pseudo-word is at 0.
s`.words()`	returns a list of word forms (strings), excluding the root pseudo-word.
s`.nwords()`	excludes the root pseudo-word.
`cmp(`*s*`,`*t*`)`	compare to other sentence *t*. Sentences are compared by comparing words from left to right until a difference is found. The root pseudo-words are assumed identical, and are not included in the comparison.
s`.append(`*w*`)`	adds *w* (not a copy) to the list of words.
s`.form(`*i*`)`	returns the form of the *i*-th word.
s`.cat(`*i*`)`	returns the category of the *i*-th word.
s`.lemma(`*i*`)`	returns the lemma of the *i*-th word.
s`.morph(`*i*`)`	returns the morph of the *i*-th word.
s`.govr(`*i*`)`	returns the governor of the *i*-th word.
s`.role(`*i*`)`	returns the role of the *i*-th word.
s`.column(`*c*`)`	returns the column named *c*, which should be one of `'form'`, `'cat'`, `'lemma'`, `'morph'`, `'govr'`, or `'role'`. The column is a list of values, one for each word. It includes the root pseudo-word.

The members of `Word` are as follows:

w`.index`	the position of the word in the sentence; the root pseudo-word has index 0.
w`.form`	the printed form of the word.
w`.cat`	the part of speech. In sentences read from a CoNLL-format file, the cat is a pair (*cpos, fpos*).

w`.lemma`	the lemma, i.e., the key to use for lexical access.
w`.morph`	morphological information.
w`.govr`	the index of the governor.
w`.role`	the role with respect to the governor.

The methods of `Word` are:

`cmp(`w_1`,`w_2`)`	Comparison is done by comparing attribute values in the order `form`, `cat`, `lemma`, `morph`, `govr`, `role`. Note that `index` is omitted: words at different positions in the sentence may be equal.
`tagged_string()`	returns "*form_cat*".

14.3.2 Conversion to `Sentence` (stemma)

A stemma is a list of `Word` objects, one for each word in the sentence. The `Word` class represents a word as the dependent in a dependency link. The function `stemma()` converts a dependency tree into a stemma. For example:

```
>>> s = stemma(d)
>>> print(s, end='')
0 *root*   None None None None
1 the      Det  None None 2
2 dog      N    None subj 3
3 chased   V    None root 0
4 a        Det  None None 5
5 cat      N    None obj  3
6 quickly  Adv  None mod  3
```

The columns are: index, word, part of speech, lemma, role, and governor. The value for governor is the index of the governor, not the governor itself.

One can access a stemma like a list:

```
>>> s[2]
<Word 2 dog/N:subj govr=3>
>>> s[2].role
'subj'
>>> s[2].govr
3
>>> s[3]
<Word 3 chased/V:root govr=0>
```

The length of the stemma is the number of words in the sentence plus one for the root:

```
>>> len(s)
7
```

The element at index 0 is a pseudo-word representing the root of the sentence.

```
>>> s[0]
<Word 0 *root*>
```

The method `words()` returns a list of word forms (strings) excluding the root pseudo-word.

```
>>> s.words()
['the', 'dog', 'chased', 'a', 'cat', 'quickly']
```

14.3.3 Governor array

A very compact representation of a dependency tree is the *governor array*. This is simply a list of numbers representing, for each word, the index of the governor of that word.

```
>>> governor_array(d)
[2, 3, 0, 5, 3, 3]
```

The argument to `governor_array()` may be either a stemma or something that can be converted to a stemma using the function `stemma()`.

14.3.4 DepLists

A `DepLists` object behaves as a list of lists. It is indexed by word index i, and returns the list of indices of words dependent on i. For example, in our example `Sentence s`, word 3 (*chased*) has dependents 2 (*dog*), 5 (*cat*), and 6 (*quickly*).

```
>>> deps = DepLists(s)
>>> deps[3]
[2, 5, 6]
>>> len(deps)
7
```

The `DepLists` object prints out readably:

```
>>> print(deps)
[0] *root*
        root: [3] chased
[1] the
[2] dog
        None: [1] the
[3] chased
        subj: [2] dog
        obj: [5] cat
        mod: [6] quickly
[4] a
[5] cat
        None: [4] a
[6] quickly
```

It contains a pointer to the original sentence, which can be used for access to the identity of the dependents, etc.

```
>>> deps.sentence[2].form
'dog'
```

14.3.5 Adding lemmata

The Sentence method `add_lemmata()` sets the `lemma` attribute for each word (except the root). It is destructive. It only works for English.

14.3.6 Eliminating epsilons

The Sentence method `eliminate_epsilons()` eliminates empty words (those whose form is `None`). It is possible for empty words to have dependents. Suppose word w has governor g, which is empty. The new governor of w is defined to be its lowest non-empty ancestor, where *ancestor* means the transitive closure of *governor-of.*

```
>>> h = parse_tree('''
...   (VP (V:head thought)
...       (CP (C:head)
...           (S
...              (NP:subj (Name:head John))
...              (VP:head (V:head left)))))
... ''')
>>> s = stemma(dependency_tree(h))
>>> print(s)
0 *root*  None None None None
1 thought V    None root 0
2 None    C    None None 1
3 John    Name None subj 4
4 left    V    None None 2
>>> print(s.eliminate_epsilons())
0 *root*  None None None None
1 thought V    None root 0
2 John    Name None subj 3
3 left    V    None None 1
```

14.4 CoNLL Format

14.4.1 Raw format

To get the raw contents of a file in CoNLL dependency format, use `seal.io.iter_record_blocks()`.

```
>>> from seal import io, ex
>>> sent = next(io.iter_record_blocks(ex.depsent1))
```

```
>>> sent[0]
['1', 'This', 'this', '_', 'pron', '_', '2', 'subj', '_', '_']
```

The fields are: index, form, lemma, cpos, fpos, morph, head, rel, phead, prel. The fields cpos, phead, and prel are considered "extra" information: they are optional, whereas fpos, head, and rel are obligatory. (Head and rel are obligatory, but need not be projective; phead and rel are optional, but must be projective.) Missing fields are represented with a single underscore character.

14.4.2 Iter, load, and save sentences

The function `iter_sentences()` reads a CoNLL-format file as a sequence of `seal.dep.Sentence` instances. It takes a filename as input, with an optional "`#proj`" or "`#std`" suffix.

The mapping between the raw fields and the Sentence attributes is done as follows. For each word, if both cpos and fpos are present, then the cat is the pair (cpos, fpos). If only one is present, it becomes the cat. If the filename ends in `#proj`, the phead and prel are used; otherwise, the head and rel are used. (The suffix "`#std`" selects head and rel, but that is also the default.)

```
>>> s = next(conll_sents(ex.depsent1))
>>> print(s[1])
<Word 1 This/pron:subj (this) govr=2>
>>> s[1].cat
'pron'
```

The function `load_sentences()` returns a list rather than an iteration. The function `save_sentences()` takes a list of sentences and a filename as input.

```
>>> save_sentences([s], '/tmp/sents')
>>> sents = load_sentences('/tmp/sents')
>>> print(sents[0])
0 *root* None None None    None
1 This   pron this subj    2
2 is     vb   be   mv      0
3 a      dt   a    det     4
4 test   n    test prednom 2
```

14.4.3 Universal postag mapping

Das and Petrov (2011) [3145] introduced a set of universal part-of-speech tags that were subsequently used in the McDonald et al. delexicalized parsers. Petrov, Das & McDonald [3300] describe a set of tag tables, which are installed in

`/cl/data/conll/2006/universal-pos-tags`

The function `load_umap()` loads a tag map from a file, returning a dict. (If given a relative pathname, it expands it relative to the `universal-pos-tags` directory.)

```
>>> map = load_umap('da-ddt.map')
>>> map['VA']
'VERB'
```

The function `apply_umap()` takes a map and a sentence in which the word `cat` values are (`cpos, fpos`) pairs, and it changes the `cat` values to be *map*[*fpos*].

The function `umapped_sents()` takes a filename and a map, and generates a sequence of sentences in which the map has been applied to the parts of speech. It takes an optional flag `projective=True` whose meaning is the same as for `conll_sents()`.

```
>>> from seal.io import data
>>> fn = data/'conll/2006/danish/ddt/train/danish_ddt_train.conll'
>>> s = next(umapped_sents(fn, map))
>>> s[1].form
'Samme'
>>> s[1].cat
'ADJ'
```

Chapter 15

Treebanks

15.1 Dependency Treebanks: `seal.data.dep`

15.1.1 Accessing datasets

A dataset has a **language** and a **version**. Languages are specified as ISO 639-3 codes. There are currently four different versions, as follows. The original CoNLL treebanks from the 2006 shared task have version `orig`. Datasets converted to the Das-Petrov universal tagset (DPU) have version `umap`. The Universal Dependency Treebank (UDT) with standard encoding has version `uni`. The Universal Dependency Treebank with content-head encoding (`ch`). The Penn Treebank (PTB) converted to dependencies using my adaptation of the Magerman-Collins (MC) rules has version `dep`. The same converted to the Das-Petrov tagset has version `umap`. Table 15.1 lists the currently available datasets.

The **name** of a dataset is language-dot-version, for example `dan.orig`. The function `dataset()` gives access to a dataset by name:

```
>>> from seal.data import dep
>>> dep.dataset('dan.orig')
<Dataset dan.orig>
```

The function `datasets()` gives access to sets of datasets. Language or version may be specified:

```
>>> dep.datasets(lang='dan')
[<Dataset dan.orig>, <Dataset dan.umap>]
>>> len(dep.datasets(version='orig'))
18
>>> len(dep.datasets())
52
```

Name	Lg	Ver	Description
arb.orig	arb	orig	CoNLL-2006 Arabic
arb.umap	arb	umap	CoNLL-2006 + DPU, Arabic
bul.orig	bul	orig	CoNLL-2006 Bulgarian
bul.umap	bul	umap	CoNLL-2006 + DPU, Bulgarian
ces.orig	ces	orig	CoNLL-2006 Czech
ces.umap	ces	umap	CoNLL-2006 + DPU, Czech
dan.orig	dan	orig	CoNLL-2006 Danish
dan.umap	dan	umap	CoNLL-2006 + DPU, Danish
deu.ch	deu	ch	UDT, content-head, German
deu.orig	deu	orig	CoNLL-2006 German
deu.umap	deu	umap	CoNLL-2006 + DPU, German
deu.uni	deu	uni	UDT, German
eng.dep	eng	dep	Penn Treebank, MC heads
eng.umap	eng	umap	Penn Treebank, MC heads + DPU
fin.ch	fin	ch	UDT, content-head, Finnish
fra.ch	fra	ch	UDT, content-head, French
fra.uni	fra	uni	UDT, French
ind.uni	ind	uni	UDT, Indonesian
ita.uni	ita	uni	UDT, Italian
jpn.uni	jpn	uni	UDT, Japanese
kor.uni	kor	uni	UDT, Korean
nld.orig	nld	orig	CoNLL-2006 Dutch
nld.umap	nld	umap	CoNLL-2006 + DPU, Dutch
por.orig	por	orig	CoNLL-2006 Portuguese
por.umap	por	umap	CoNLL-2006 + DPU, Portuguese
por.uni	por	uni	UDT, Portuguese
slv.orig	slv	orig	CoNLL-2006 Slovenian
slv.umap	slv	umap	CoNLL-2006 + DPU, Slovenian
spa.ch	spa	ch	UDT, content-head, Spanish
spa.orig	spa	orig	CoNLL-2006 Spanish
spa.umap	spa	umap	CoNLL-2006 + DPU, Spanish
spa.uni	spa	uni	UDT, Spanish
swe.ch	swe	ch	UDT, content-head, Swedish
swe.orig	swe	orig	CoNLL-2006 Swedish
swe.umap	swe	umap	CoNLL-2006 + DPU, Swedish
swe.uni	swe	uni	UDT, Swedish
tur.orig	tur	orig	CoNLL-2006 Turkish
tur.umap	tur	umap	CoNLL-2006 + DPU, Turkish

Table 15.1: The available datasets. DPU = Das-Petrov Universal tagset. UDT = Universal Dependency Treebank.

15.1.2 Dataset instances

The class `Dataset` represents a treebank. There are two specializations, `UMappedDataset` and `FilterDataset`. Each dataset has a name, a description, a language represented as an ISO 639-3 code, and a version.

```
>>> ds = dep.dataset('dan.orig')
>>> ds.name
'dan.orig'
>>> ds.desc
'Danish, CoNLL-2006'
>>> ds.lang
'dan'
>>> ds.version
'orig'
```

Simple datasets also have a training file pathname, a test file pathname, and (sometimes) a dev file pathname. (To be precise, datasets in the `uni` and `ch` collections have a dev file pathname, but `orig` datasets do not.) The pathnames are also available for umapped datasets, but the files contain the original (unmapped) trees. Filter datasets do not have pathnames.

```
>>> from seal.config import relpath
>>> relpath(ds.train)
'data/conll/2006/danish/ddt/train/danish_ddt_train.conll'
>>> relpath(ds.test)
'data/conll/2006/danish/ddt/test/danish_ddt_test.conll'
>>> ds.dev
>>>
```

15.1.3 Sentences

`sents()`

A dataset instance has a `sents()` method that generates sentences for a specified **section** of the treebank. All treebanks have `'train'` and `'test'` sections. In addition, `uni` and `ch` datasets have a `'dev'` section, and the English datasets have `'dev_train'`, `'dev_test'`, and `'reserve_test'` sections.

```
>>> sents = list(ds.sents('train'))
>>> len(sents[0])
14
```

A convenience function called `sents()` is also available to retrieve the sentences for a particular segment of a dataset directly:

```
>>> sents = list(dep.sents('dan.orig', 'train'))
```

Sentences

A sentence can be viewed as a list of records. Word 0 is always the root pseudo-word. "Real" words start at position 1. The length of the sentence includes the root, so the last valid index is the length minus one.

```
>>> s = sents[0]
>>> s[0]
<Word 0 *root*>
>>> s[1]
<Word 1 Samme/A.AN:ROOT (/degree=pos...) govr=0>
>>> s[13]
<Word 13 ./X.XP:pnct govr=1>
```

Each record has ten fields: `i`, `form`, `lemma`, `cpos`, `fpos`, `morph`, `govr`, `role`, `pgovr`, and `prole`. The field `cpos` represents the coarse part of speech, and `fpos` represents the fine part of speech. The fields `pgovr` and `prole` represent the word's governor and role in the projective stemma. They may not be available. The fields `govr` and `role` are always available, but they are not guaranteed to be projective.

All fields except `i`, `govr`, and `pgovr` are string-valued. If not available, their value is the empty string. The values for `i`, `govr`, and `pgovr` are integers. If they are not available, their value is `None`. The fields `i` and `govr` are always available, except that word 0 has no govr.

The values for `govr` and `pgovr` can be used used as an index into the sentence, with the value 0 representing the root.

One can get just a list of word forms (strings) using the method `words()`. This provides suitable input for a standard parser. The root pseudo-word is not included. The method `nwords()` returns the number of words excluding the root.

```
>>> ws = s.words()
>>> ws[:3]
['Samme', 'cifre', ',']
>>> len(ws)
13
>>> s.nwords()
13
```

Column-major view

A sentence provides separate methods for each of the word attributes, indexed by the word number, with 0 being the root pseudo-word.

```
>>> s.form(0)
'*root*'
>>> s.form(1)
'Samme'
```

```
>>> s.form(13)
'.'
```

The attributes are as listed above: `form`, `lemma`, `cpos`, `fpos`, `morph`, `govr`, `role`, `pgovr`, and `prole`.

```
>>> s.form(2)
'cifre'
>>> s.lemma(2)
''
>>> s.cat(2)
('N', 'NC')
>>> s.morph(2)
'gender=neuter|number=plur|case=unmarked|def=indef'
>>> s.govr(2)
1
>>> s.role(2)
'nobj'
```

Word forms need not be ascii.

```
>>> from seal.misc import as_ascii
>>> as_ascii(s.form(12))
'v{e6}rtsnation'
```

Without `as_ascii`, the form would print as "værtsnation."

One can fetch a column as a tuple using the method `column()`.

```
>>> g = s.column('govr')
>>> g[:5]
(None, 0, 1, 1, 7)
```

Creating a sentence

If desired, one can create a Sentence as follows.

```
>>> from seal.dep import Sentence, Word
>>> s = Sentence()
>>> s.append(Word(1, 'This', ('PRON', 'PRON'), 'this', '', 2, 'subj'))
>>> s.append(Word(2, 'is', ('VB', 'VB'), 'be', '', 0, 'mv'))
>>> s.append(Word(3, 'a', ('DT', 'DT'), 'a', '', 4, 'det'))
>>> s.append(Word(4, 'test', ('N', 'N'), 'test', '', 2, 'prednom'))
```

The numbers must be sequential from 1; they provide a quality check.

15.1.4 Dependency files

Loading

On disk, the training and test files are in CoNLL dependency format. The `sents()` method uses `seal.dep.conll_sents()` to read them:

```
>>> from seal.dep import conll_sents
>>> f = conll_sents(ds.train)
>>> s = next(f)
>>> len(s)
14
```

Format

The file `seal.ex.depsent1` provides an example:

```
1       This    this    pron    pron    _       2       subj    2       subj
2       is      is      vb      vb      _       0       mv      0       mv
3       a       a       dt      dt      _       4       det     4       det
4       test    test    n       n       _       2       prednom 2       prednom

```

Each sentence is (obligatorily) terminated by an empty line. Fields are separated by single tab characters. There are ten fields: *id, form, lemma, cpos, fpos, morph, govr, role, pgovr, prole.*

15.1.5 Universal Pos Tags

The `'umap'` versions of the treebanks are mapped from the `'orig'` versions using the tag tables of Petrov, Das & McDonald [3300]. They are instances of `UMappedDataset`, which uses `UMappedDepFile`. See §14.4.3.

```
>>> ds = dep.dataset('dan.umap')
>>> s = next(ds.sents('train'))
>>> s[1].form
'Samme'
>>> s[1].cat
'ADJ'
```

Chapter 16

Dependency Parser: `seal.dp`

16.1 Pseudo-projective parsing: seal.dp.nnproj

Pseudo-projective parsing involves a transformation applied to a set of CoNLL sentences, and an inverse transformation applied to the output of the parser.

The transformation converts the stemma to a projective stemma. Nonprojectivity arises exactly when we have two crossing arcs, instead of proper nesting.

This section assumes:

```
>>> from seal.dp.nnproj import *
>>> from seal import ex
>>> from seal.dep import conll_sents
>>> from seal.data import dep
```

16.1.1 Toplevel

The function `print_stats()` runs the projectivizer and reverter on a list of sentences, and reports the results. For example:

```
>>> print_stats(dep.sents('dan.orig', 'test'))
Projective:       280 / 322 (86.956522%)
Not projective:   42 / 322 (13.043478%)

Not projective:
  Revertible:     39 / 42 (92.857143%)
  Not revertible: 3 / 42 (7.142857%)

Revertible:
   1 lifts:    33
   2 lifts:     3
   3 lifts:     1
   4 lifts:     2

Not revertible:
   1 lifts:     1
   2 lifts:     2
```

16.1.2 Nivre & Nilsson's algorithm

An arc (g, d) is defined to be nonprojective just in case it crosses another arc (g', d') and g' dominates g. Find the shortest nonprojective arc (g, d), breaking ties in favor of leftmost arcs. Lift (g, d) by replacing it with (h, d), where h is the governor of g. Continue until there are no nonprojective arcs.

16.1.3 Functions

Let us use the following sentence as a running example.

```
>>> s = next(conll_sents(ex.depsent2))
>>> print(s)
0 *root* None    None None None
1 a      pos.pos a    A    2
2 b      pos.pos b    B    4
3 c      pos.pos c    C    2
4 d      pos.pos d    D    0
5 e      pos.pos e    E    7
6 f      pos.pos f    F    3
7 g      pos.pos g    G    0
8 h      pos.pos h    H    7
>>> govrs = s.column('govr')
```

The following functions provided by `seal.depparse` apply to governor lists.

Dominates determines whether a given word dominates another. Domination is reflexive and transitive.

```
>>> dominates(4, 1, govrs)
True
>>> dominates(4, 5, govrs)
False
```

Is nonproj determines whether an arc is nonprojective or not. An arc (g, d) is defined to be nonprojective just in case any word between g and d (exclusive) has a governor that is outside the range (g, d).

```
>>> is_nonproj((3,6), govrs)
True
```

Has nonproj arcs returns `True` if there are any nonprojective arcs in the sentence.

```
>>> has_nonproj_arcs(govrs)
True
```

Nonproj arcs returns an iterator over the nonprojective arcs in the sentence.

```
>>> list(nonproj_arcs(govrs))
[(7, 5), (3, 6)]
```

Next nonproj arc returns the nonprojective arc with the smallest span. It breaks ties in favor of the leftmost arc.

```
>>> next_nonproj_arc(govrs)
(7, 5)
```

16.1.4 Projectivizer functions

Projectivize takes either a sentence or iterator over sentences, and returns the same type of object.

```
>>> ps = projectivize(s)
>>> print(ps)
0 *root* None    None None None
1 a      pos.pos a    A    2
2 b      pos.pos b    B    4
3 c      pos.pos c    C    2
4 d      pos.pos d    D    0
5 e      pos.pos e    G|E  0
6 f      pos.pos f    C|F  0
7 g      pos.pos g    G    0
8 h      pos.pos h    H    7
```

The return value is a copy; the original sentence is not modified. The projectivizer only modifies non-projective arcs, so if the original sentence is already projective, the new sentence is identical to the old.

Revert takes a projectivized sentence, or iterator over sentences, and attempts to reconstruct the original.

```
>>> rs = revert(ps)
>>> rs == s
True
```

Stats takes a sentence or an iterator over sentences. For a single sentence, it projectives and then attempts to revert the sentence. It then returns a pair (*rev, nlifts*) where *rev* is either `'revertible'` or `'not-revertible'` and *nlifts* is the number of lifts performed during projectivization. (Zero lifts means that the original was already projective.) For example:

```
>>> stats(s)
('revertible', 4)
```

For a list of sentences, `stats()` returns a table mapping the stats produced for single sentences to the list of indices of sentences that have those stats. (Note that it uses *sent*`.index()`, not the actual position of the sentence in the input list.)

```
>>> sents = dep.sents('dan.orig', 'test')
>>> tab = stats(sents)
>>> for (k,v) in sorted(tab.items()):
...     print(k, len(v))
...
('not-revertible', 1) 1
```

```
('not-revertible', 2) 2
('revertible', 0) 280
('revertible', 1) 33
('revertible', 2) 3
('revertible', 3) 1
('revertible', 4) 2
>>> tab['not-revertible', 2]
[131, 198]
```

16.1.5 Projectivizer implementation

A `Projectivizer` implements the Nivre & Nilsson algorithm.

```
>>> p = Projectivizer()
```

It implements the following methods.

Set sent sets `p.orig` to a given sentence. It initializes `p.govrs` and `p.roles` to be copies of the corresponding columns of the sentence. It initializes `p.lifted` to be a list containing `False` for each word in the sentence. And it initializes `p.nlifts` to 0.

```
>>> p.set_sent(s)
>>> print(p)
(2, 1) (4, 2) (2, 3) (0, 4) (7, 5) (3, 6) (0, 7) (7, 8)
```

Note that printing a projectivizer lists the arcs represented by its `govrs`.

Lift takes an arc (g, d) as input. It changes the governor of d to be the governor of g.

```
>>> p.lift((7,5))
>>> print(p)
(2, 1) (4, 2) (2, 3) (0, 4) (0, 5) (3, 6) (0, 7) (7, 8)
```

The governor of 7 is 0, so the arc $(7, 5)$ has been replaced with $(0, 5)$.

Run repeatedly chooses the next arc and lifts it, until there are no more nonprojective arcs. It returns the resulting list of governors.

```
>>> p.run()
>>> print(p)
(2, 1) (4, 2) (2, 3) (0, 4) (0, 5) (0, 6) (0, 7) (7, 8)
```

Sentence returns an updated CoNLL sentence.

```
>>> print(p.sentence())
0 *root* None    None None None
1 a      pos.pos a    A    2
2 b      pos.pos b    B    4
3 c      pos.pos c    C    2
4 d      pos.pos d    D    0
5 e      pos.pos e    G|E  0
6 f      pos.pos f    C|F  0
7 g      pos.pos g    G    0
8 h      pos.pos h    H    7
```

Calling a projectivizer as a function calls `set_sent()` and `run()`, and then calls `sentence()` to generate a projectivized sentence. However, if the input sentence already contains p-governors, it immediately returns the input sentence. To be precise, it returns a triple (s, p, n) where s is the projectivized sentence, p is `True` if the projectivized sentence is in fact the original sentence, and n is the number of lifts performed, or `None` if the output is the original sentence.

16.1.6 Reverter implementation

The function `set_sent()` initializes the reverter with a new sentence.

```
>>> r = Reverter()
>>> r.set_sent(ps)
>>> print(r)
0 None None
1 2    A
2 4    B
3 2    C
4 0    D
5 0    G|E
6 0    C|F
7 0    G
8 7    H
```

The method `find_govr()` is given arguments *root* and *role*, and does a breadth-first search starting at *root* to find a word whose role is *role.*

```
>>> r.find_govr(0, 'G')
7
```

The method `lower()` is given a word d as input. It calls `find_govr()` to find a new governor for d, and reattaches d to the new governor.

```
>>> r.lower(5)
>>> print(r)
0 None None
1 2    A
2 4    B
3 2    C
4 0    D
5 7    E
6 0    C|F
7 0    G
8 7    H
```

The method `run()` goes through the sentence from left to right. It calls `lower()` on each word whose role contains a vertical bar.

```
>>> r.run()
>>> print(r)
0 None None
1 2    A
2 4    B
3 2    C
4 0    D
5 7    E
6 3    F
7 0    G
8 7    H
```

The method `sentence()` returns a sentence whose govrs and roles are taken from the current state of the reverter. Pgovrs and proles are empty.

```
>>> rs = r.sentence()
>>> rs == s
True
```

Calling the reverter as a function does `set_sent()` and `run()`, and returns `sentence()`.

16.2 Parser: seal.dp.parser

The dependency parser described here is based on Nivre (2007). See detailed discussion in [p138]. The examples in this section assume you have done:

```
>>> from seal import ex
>>> from seal.dp.parser import *
>>> from seal.dep import conll_sents
>>> from seal.data import dep
```

16.2.1 Configurations

A `Configuration` contains a stack and an input pointer. One initializes a configuration either from a tokenized sentence (i.e., a simple list of strings) or from a `seal.dep.Sentence` instance, in which case the `words()` method is called to get a list of strings. The stack is initialized to contain just a root node.

The attribute `words` contains the sentence as list of strings, with the pseudo-word `'*root*'` as the 0-th word. The attribute `sent` contains the `Sentence` (if any).

```
>>> c0 = Configuration(['this', 'is', 'a', 'test'])
>>> c0.words
['*root*', 'this', 'is', 'a', 'test']
>>> c0.sent
>>>
```

The member `pointer` indicates the earliest word that is yet to be processed. Its value is initially 1.

```
>>> c0.pointer
1
```

The method `input()` indexes words relative to the pointer. The word at the pointer is number 0. The return value is a word index, or `None` if the given index is invalid.

```
>>> c0.input(0)
1
>>> c0.input(-1)
>>> c0.input(4)
>>>
```

The stack contains word indices. It is contained in the member `_stack`, but it is accessed through the method `stack()`. The bottom of the stack is conceptually to the left (earlier words) and the top is to the right (later words). The top of the stack is position 0. Invalid positions are defined to contain `None`.

```
>>> c0.stack(0)
0
>>> c0.stack(1)
>>>
```

The first few parsing actions are typically to shift words onto the stack, with the result that the stack simply contains the first few nonnegative integers. But after some attachments are performed, the stack will no longer have such a simple relationship to the sentence. For example:

```
>>> c1 = c0.shift()
>>> c2 = c1.attach_right('subj')
>>> tmp = c2.shift()
>>> tmp._stack
[0, 2]
```

There is one more data structure, in the member `_nodes`. It contains attachment information resulting from parsing actions. There are four actions: shifting a word from the input onto the stack, attaching the next input word leftwards (to the word on top of the stack), attaching the top word on the stack rightwards (to the first input word), and popping the stack.

The member `_nodes` contains one `Node` for each word in the sentence (with 0 being the root node). A `Node` has the following members:

`index`	its position in the sentence, with the root at 0.
`govr`	the index of its governor.
`role`	its role with respect to its governor.
`lc`	the index of its leftmost left child.
`rc`	the index of its rightmost right child.
`ls`	the index of its preceding sibling, if it is a right child.
`rs`	the index of its following sibling, if it is a left child.

All except `index` may have the value `None`.

16.2.2 Elementary features

The following methods are used to compute feature values. They all take a word index w as input, and they are forgiving in the sense that they simply return `None` if w is `None`, or if the requested feature does not exist. The return values are either strings or word indices, or `None`.

`word(`w`)`	the word form (string).
`lemma(`w`)`	the lemma (string).
`cpos(`w`)`	the coarse part of speech. If the input is a CoNLL sentence, this is `cat[0]`, and otherwise it is `cat`.
`fpos(`w`)`	the fine part of speech. If the input is a CoNLL sentence, this is `cat[1]`, and otherwise it is `cat`.

`morph(`w`)` the morphological information (string).

`true_govr(`w`)` the governor recorded in the original `Sentence`. Signals an error if the configuration was not initialized from a `Sentence`.

`true_role(`w`)` the role recorded in the original `Sentence`.

`govr(`w`)` the governor, if the word has been attached.

`role(`w`)` the role, if the word has been attached.

`lc(`w`)` the leftmost child, if this word has any left children.

`rc(`w`)` the rightmost child, if this word has any right children.

`ls(`w`)` the left sibling, if this node is a right child and there are preceding right children.

`rs(`w`)` the right sibling, if this node is a left child and there are any following left children.

`is_complete(`w`)` indicates whether a given word has acquired all of its true dependents. To be precise, it returns `False` if any of the unattached words in lookahead have the given word as true governor.

Continuing with our previous example:

```
>>> c2.word(0)
'*root*'
>>> c2.word(None)
>>>
>>> c2.govr(1)
2
>>> c2.role(1)
'subj'
>>> c2.lc(2)
1
```

To illustrate the "supervised" methods, let us create a configuration from a CoNLL sentence.

```
>>> sent = next(conll_sents(ex.depsent2))
>>> print(sent)
0 *root* None    None None None
1 a      pos.pos a    A    2
2 b      pos.pos b    B    4
3 c      pos.pos c    C    2
4 d      pos.pos d    D    0
5 e      pos.pos e    E    7
```

```
6 f      pos.pos f    F    3
7 g      pos.pos g    G    0
8 h      pos.pos h    H    7
>>> cc = Configuration(sent)
```

We shift the first word onto the stack and attach it rightwards, leaving just the root on the stack and "b" as the next word of input:

```
>>> cc = cc.shift()
>>> cc = cc.attach_right('A')
>>> print(cc)
Configuration 0.2:
    stack: 0
    pointer: 2
    tgovr:    2  4  2  0  7  3  0  7
    trole:    A  B  C  D  E  F  G  H
    govr:     2
    role:     A
    cpos:  2  po po po po po po po po
    fpos:  2  po po po po po po po po
    form:  *r a  b  c  d  e  f  g  h
    i:     0  1  2  3  4  5  6  7  8
           *     |-
```

Now attach word 2 to the root (leftwards):

```
>>> cc = cc.attach_left('B')
>>> print(cc)
Configuration 0.3:
    stack: 0 2
    pointer: 3
    tgovr:    2  4  2  0  7  3  0  7
    trole:    A  B  C  D  E  F  G  H
    govr:     2  0
    role:     A  B
    cpos:  2  po po po po po po po po
    fpos:  2  po po po po po po po po
    form:  *r a  b  c  d  e  f  g  h
    i:     0  1  2  3  4  5  6  7  8
           *     *  |-
```

Now word 2 has a governor (albeit the incorrect one), but it is still incomplete because word 3's true governor is 2:

```
>>> cc.govr(2)
0
>>> cc.true_govr(2)
4
```

```
>>> cc.true_govr(3)
2
>>> cc.is_complete(2)
False
```

Attaching word 3 to word 2 completes word 2:

```
>>> cc = cc.attach_left('C')
>>> print(cc)
Configuration 0.4:
    stack: 0 2 3
    pointer: 4
    tgovr:    2  4  2  0  7  3  0  7
    trole:    A  B  C  D  E  F  G  H
    govr:     2  0  2
    role:     A  B  C
    cpos:  2  po po po po po po po po
    fpos:  2  po po po po po po po po
    form:  *r a  b  c  d  e  f  g  h
    i:     0  1  2  3  4  5  6  7  8
           *     *  *  |-
>>> cc.is_complete(2)
True
```

16.2.3 Actions

The actions for an arc-eager stack-based parser are implemented. As briefly mentioned above, there are four actions.

Shift pushes the first input word onto the stack and moves the input pointer one position to the right.

Attach right attaches the word on top of the stack rightwards, to the first input word. The attached word is popped off the stack. An error is signalled if the word on top of the stack already has a governor.

Attach left attaches the first word in the input leftwards, to the word on top of the stack. An error is signalled if the word to be attached already has a governor. The newly attached word is shifted onto the stack, and the input pointer is advanced.

Reduce pops the stack. It is assumed that the word on top of the stack has a governor, but no check is done.

16.2.4 Executing an action

The configuration can be applied as a function to an abbreviated action name: `'al'` (attach left), `'ar'` (attach right), `'sh'` (shift), `'re'` (reduce). An optional second argument provides the label, for the attachment actions.

```
>>> print(c2('al', 'mv'))
Configuration:
    stack: 0 2
    pointer: 3
    govr:    2  0
    role:    su mv
    form: *r th is a  te
    i:    0  1  2  3  4
          *     *  |-
```

16.2.5 Supervised oracle

An oracle function takes a configuration and returns the next action to take. The function `supervised_oracle()` expects a configuration constructed from a labeled sentence, and looks at the true stemma to determine the next action. The configuration must have a value for `conll`.

```
>>> s = next(conll_sents(ex.depsent1))
>>> print(s)
0 *root* None None None    None
1 This   pron this subj    2
2 is     vb   be   mv      0
3 a      dt   a    det     4
4 test   n    test prednom 2
```

Here is an example of using the supervised oracle:

```
>>> c = Configuration(s)
>>> supervised_oracle(c)
('sh', None)
>>> (act, role) = _
>>> c = c(act, role)
>>> print(c.buffer_string())
*r Th | is a te
```

The oracle works as follows. Let L and R be the two words on either side of the pointer.

- If R doesn't exist, stop.
- If R's true governor is L, and R is unattached, then attach-left.
- If L's true governor is R, and L is unattached, then attach-right.

- If L is attached and complete (i.e., no word in the lookahead is governed by L), then reduce.
- Otherwise, shift.

One can perform an entire computation using the function `computation()`. The output is a list of triples (*config, act, role*).

```
>>> comp = computation(s, supervised_oracle)
>>> (cfg, act, role) = comp[2]
>>> print(cfg)
Configuration 0.2:
    stack: 0
    pointer: 2
    tgovr:    2  0  4  2
    trole:    su mv de pr
    govr:     2
    role:     su
    cpos:  2  pr vb dt n
    fpos:  2  pr vb dt n
    form:  *r Th is a  te
    i:     0  1  2  3  4
           *     |-
```

For convenience, there is also a `print_computation()` function:

```
>>> print_computation(comp)
*r | Th is a te
 -> sh None
*r Th | is a te
 -> ar subj
*r | is a te
 -> al mv
*r is | a te
 -> sh None
*r is a | te
 -> ar det
*r is | te
 -> al prednom
*r is te |
 -> stop None
```

16.2.6 Creating a classifier training set

The function `instances()` takes a `Sentence` and a feature function, and produces a sequence of machine-learning instances. It calls `computation()` to get a sequence of configurations with actions. Each step produces a machine-learning

instance. The action is the instance label (the role, if any, is appended to the action), and the instance's features are the result of applying the feature function to the configuration.

```
>>> for inst in instances(s, simple_features):
...     print(inst)
...
sh s2:None s1:*root* la1:This la2:is
ar_subj s2:*root* s1:This la1:is la2:a
al_mv s2:None s1:*root* la1:is la2:a
sh s2:*root* s1:is la1:a la2:test
ar_det s2:is s1:a la1:test la2:None
al_prednom s2:*root* s1:is la1:test la2:None
```

The feature function receives a configuration as input and returns a list of attribute-values pairs. `Simple_features()` is a fairly trivial example.

```
>>> (c,_,_) = comp[2]
>>> print(c)
Configuration 0.2:
    stack: 0
    pointer: 2
    tgovr:    2  0  4  2
    trole:    su mv de pr
    govr:     2
    role:     su
    cpos:  2  pr vb dt n
    fpos:  2  pr vb dt n
    form:  *r Th is a  te
    i:     0  1  2  3  4
           *     |-
>>> simple_features(c)
[('s2', None), ('s1', '*root*'), ('la1', 'is'), ('la2', 'a')]
```

16.3 Features: seal.dp.features

16.3.1 Compile

The main function is `compile()`, which takes a set of feature specifications (a string) and produces a function that maps configurations to instances.

```
>>> from seal.dp.features import *
>>> cfgs = [cfg for (cfg,_,_) in comp]
>>> f = compile('fpos stack 0, fpos input 0')
>>> f(cfgs[0])
[('fpos.input.0', 'pron')]
>>> f(cfgs[1])
[('fpos.stack.0', 'pron'), ('fpos.input.0', 'vb')]
```

By default, features with a null value are suppressed. One can change this behavior by passing `nulls=True` to `compile()`.

```
>>> f = compile('fpos stack 0, fpos input 0', nulls=True)
>>> f(cfgs[0])
[('fpos.stack.0', 'null'), ('fpos.input.0', 'pron')]
```

16.3.2 Format

Feature specifications are built up from accessor functions such as `fpos` and `stack`. The simplest specifications are of the form `'stack 0'` or `'input 2'`, in which the argument is a number. Only the functions `stack` and `input` may be used in this way. All other functions take a subexpression as argument. The available functions are: `form`, `lemma`, `cpos`, `fpos`, `morph`, `govr`, `role`, `lc`, `rc`, `ls`, `rs`. Multiple feature specifications may be separated either by comma or newline.

16.3.3 Load

One can alternatively load feature specifications from a file.

16.3.4 Implementation

The function `load()` simply calls `compile()` on the contents of the file. The function `compile()` first splits the input text into feature specs. Feature specs may be separated either by commas or newlines.

```
>>> from seal.dp.features import specs
>>> sps = specs('form input 0, fpos input 0, role lc input 0')
>>> sps
['form input 0', 'fpos input 0', 'role lc input 0']
```

The specs are then used to create a `FunctionList` object, which in turn uses `_compile1()` to turn each spec into a function.

The function `_compile1()` takes a spec consisting of a sequence of words, like `['role', 'lc', 'input', '0']`. The first word is the *operator*. The operators `stack` and `input` are nonrecursive; they take the next word (which must be the last word) as argument. For example,

```
1   _compile1(['input', '0'])
```

converts the `'0'` to an int and returns the function:

```
1   lambda cfg: cfg.input(0)
```

The other operators are recursive. For example, if the first word is `lc`, the remainder of the spec is passed to `_compile1()` to obtain a function `f`, and the return value is:

```
1   lambda cfg: cfg.lc(f(cfg))
```

The result is always a function that takes a configuration as input and returns a string or `None`.

16.4 Evaluation: seal.dp.eval

The following functions are in the module seal.dp.eval.

```
>>> from seal.dp.eval import *
>>> from seal import ex
>>> from seal.dep import conll_sents
```

16.4.1 evaluate

This is the main function. It takes a parser, a list of sentences with gold pgovrs and proles, and prints out evaluation information. The parser should place its output in the govr and role slots, not pgovr and prole. One may specify excludepunc=False to count punctuation tokens. (They are ignored by default.) One may provide output=*stream* to specify an output stream other than stdout.

```
>>> evaluate(parser, sents)
```

16.4.2 ispunc

The function ispunc() returns True if all the characters in the given string have a Unicode category beginning with "P."

```
>>> ispunc('.')
True
>>> ispunc('Dr.')
False
```

16.4.3 eval_sent

The function eval_sent() evaluates a single sentence. Its arguments are *pred* and *truth.* It considers the govrs and roles of the predicted sentence, but the pgovrs and proles of the true sentence. (A projective dependency parser can produce non-projective output if it ever fails to attach a word, so the output of even a projective dependency parser is stored in the govr/role slots rather than the pgovr/prole slots.)

The outputs are *las, uas, la, n,* where *las* is the number of words that have the correct govr and role, *uas* is the number of words that have the correct govr, *la* is the number of words that have the correct role, and n is the number of words. Nota bene: these are counts, not proportions. Note also that n will be less than the length of the sentence. The length of the sentence includes the root token (position 0), which is never included in n. Also, by default, punctuation tokens are ignored. (One can cause them to be counted by specifying excludepunc=False.)

```
>>> pred = next(conll_sents(ex.depsent3_pred))
>>> gold = next(conll_sents(ex.depsent3_gold))
```

```
>>> eval_sent(pred, gold)
(2, 3, 2, 4)
>>> eval_sent(pred, gold, excludepunc=False)
(3, 4, 3, 5)
```

16.4.4 compare

The function `compare()` prints out a detailed comparison of a predicted and a gold sentence.

```
>>> compare(pred, gold)
1   This G R 2 subj 2 subj
2   is   G R 0 mv   0 mv
3   a        2 pt   4 det
4   test G   2 obj  2 prednom
5 * .        2 obj  2 prednom

LAS: 2 4 0.5
UAS: 3 4 0.75
LA:  2 4 0.5
```

Punctuation tokens are marked with "`*`" in the second column. Tokens marked "`G`" contribute to the UAS score, tokens marked "`R`" contribute to the LA score, and tokens marked "`G R`" contribute to the LAS score.

16.5 Nivre parser: seal.dp.nivre

16.5.1 Experiment

Here is an example of running an experiment:

```
$ cp /cl/examples/nivre-2007.ftrs ./
$ cp /cl/examples/nivre.exp ./
$ python -m seal.ml.experiment nivre.exp work
```

This creates the directory `work` as a subdirectory of the current working directory. All output is written in `work`, except the main summary, which is written to stdout and also saved in the file `nivre.out` as a sister to `nivre.exp`. When this particular experiment has completed, the file `nivre.out` should contain, among other things:

```
...
acc: 0.908801020408 correct= 9975 ntest= 10976
...
LAS:    3912 4991 0.783810859547
UAS:    4102 4991 0.821879382889
LA:     4391 4991 0.879783610499
NSents: 206
```

The file `nivre.exp` is called the **experiment file**. Omitting the `.exp` extension gives the **experiment name**, which in this case is `nivre`. The directory in which the experiment file resides (the current working directory, in this case), is called the **experiment directory**. The directory `work` is called the **working directory.** See §8.7.

Here is an example of an experiment file:

```
command seal.dp.nivre
dataset spa.orig
features nivre-2007
nulls True
split.feature fpos.input.0
split.cpt.s 0
split.cpt.t 1
split.cpt.d 2
split.cpt.g 0.2
split.cpt.c 0.5
split.cpt.r 0
split.cpt.e 1.0
```

The command is `seal.dp.nivre`, which names the module. The function that runs the experiment is `run_experiment` within that module. The steps it goes through are the following.

- Call `save_experiment()` and then `load_experiment()`, to make a persistent copy of the experiment and feature files in the working directory. Loading the experiment also creates the feature function and loads the dataset.
- Call the `train()` function to train an oracle, which is a classifier whose classes are parsing actions. The `train()` function converts training and testing sentences to instances, then uses them to train an oracle.
- Load a `Model` from the working directory.
- Call the model's `accuracy()` method to get the classification accuracy of the oracle on the test instances.
- Call the model's `evaluation()` method to use the oracle to parse the test sentences, and determine the accuracy (LAS, UAS) of the resulting parser.

Output is passed through a `tee` so that it goes both to stdout and to the file *expname*`.out` in the experiment directory.

Dataset. The dataset `spa.orig` is Spanish, original format. To get a list of available datasets:

```
>>> from seal.data import dep
>>> sorted(dep.datasets)
```

See §15.1.1 for details.

Features. The features are `nivre-2007`, which are found in the file `nivre-2007.ftrs` residing in the experiment directory. Here are the contents of the feature file:

```
form input 0
lemma input 0
cpos input 0
fpos input 0
morph input 0
form input 1
fpos input 1
fpos input 2
fpos input 3
role lc input 0
form stack 0
lemma stack 0
cpos stack 0
fpos stack 0
morph stack 0
role stack 0
fpos stack 1
```

```
form govr stack 0
role lc stack 0
role rc stack 0
```

The first line says that the `input[0].form` is one feature. The last line says that `stack[0].rc.role` is one feature. For more details, see §16.3.

Nulls. There are two ways that a feature may be null: either the feature expression (e.g., `input[0].form`) results in an error when evaluated, or it results in a value that is boolean false. If `nulls` is true, then null values are represented as `null`. Otherwise, features with null values are omitted from the instance. See §16.3.

Split. The parser calls `seal.ml.split` to do training and testing. It splits instances into sub-datasets and does SVM training on each sub-dataset separately. The value of `split.feature` is the feature to use to split the dataset: each distinct value of the feature names a separate sub-dataset.

Split.cpt. The split trainer calls a learner on each sub-dataset. Here the learner is hardcoded as `seal.ml.libsvm`. The `split.cpt` settings are parameters of the libsvm learner. See §8.5.

16.5.2 General usage

To train and use a parser, one first requires an experiment file. Assume that `ptb.exp` contains the contents:

```
command seal.dp.nivre
dataset ptb.umap
features delex
nulls True
split.feature fpos.input.0
split.cpt.s 0
split.cpt.t 1
split.cpt.d 2
split.cpt.g 0.2
split.cpt.c 0.5
split.cpt.r 0
split.cpt.e 1.0
```

Then one creates the model directory `ptb.model` by doing:

```
>>> from seal.dp import nivre
>>> nivre.train('ptb)
```

Training also creates the directory `foo.work`. The work directory can be used to evaluate parser accuracy, provided that the training dataset includes a

test portion as well. There are two separate functions for measuring accuracy. Remember that the parser uses an oracle. For a given test sentence, the correct parse translates into a sequence of parsing actions, each taken from a particular configuration. Each configuration corresponds to a learning instance, and the correct action is the true label. The `accuracy()` function reports on the accuracy of the trained oracle on the test instances.

```
1
2
3     ; it gives the
4     proportion of correct predictions that it makes on the testing instances.
5
6     To train:
7     \begin{myverb}
8
9     >>> nivre.train('foo')
```

The file `'foo.exp'` must exist. This writes a lot of files, split by part of speech of INPUT[0]. The list of parts of speech occurring in training is written to `StatsTrainParts` and those in test files are written to `StatsTestParts`. Training is only done where both training and testing files exist.

To compute the accuracy of the predictions on the test files:

```
1     >>> nivre.accuracy()
2     Accuracy: 0.581359329446 correct= 6381 ntest= 10976
3     Fa acc= 0.333333333333 correct= 1 ntest= 3
4     Fc acc= 0.639606396064 correct= 520 ntest= 813
5     Fd acc= 0.576923076923 correct= 15 ntest= 26
6     ...
```

16.5.3 Options

The `train()` function takes the following options:

- `features`: the filename of a set of feature specifications.
- `split_ftr`: the attribute to use for splitting up the training data.

Chapter 17

MST Parser: `seal.mst`

```
>>> from seal.mst import mst
>>> mst('spa')
```

The function `mst()` accepts the following optional arguments:

- `fmt`: one of `'orig'`, `'uni'`, `'ch'`. Default is `'orig'`.
- `outfn`: redirect stdout to this file.
- `space`: the amount of memory to allocate, in GB. Defaults to `'5'`.

The language name and format are passed to `seal.data.dep.datasets` to retrieve the dataset for parsing. Its `train` and `test` files are used.

Part V

Preprocessing and Finite-State Models

Chapter 18

Preprocessing

18.1 Orthography: `seal.orth`

A **transcript** is a representation that is intended to be neutral between speech and text. For text, it represents the result of tokenization and normalization. For example, "18" and "eighteen" are both normalized to "eighteen." In general, the normal forms are written out.

18.1.1 Transcriber

The `Transcriber` class converts a text to a transcript. The text is represented as a single string. The `Transcriber` instance is an iterator over normalized tokens.

18.1.2 Abbreviations

The transcriber uses a table of abbreviations:

`/cl/data/seal/abbreviations`

18.2 Tokenizer: `seal.tok`

The module `seal.tok` contains a tokenizer for Latin scripts.

18.2.1 Usage

The main function is `tokenized`, which takes a text (string) and returns a list of tokens.

```
1  >>> from seal.tok import tokenized
2  >>> t = tokenized('"Hi, @#!"\nsaid  42-J.\n')
3  >>> t[:4]
4  [<word '"Hi,'>, <punct '@#!"'>, <word 'said'>, <number '42'>]
```

`type()`	`'word'`, `'number'`, `'hyphen'`, or `'punct'`[1]
`string()`	the original characters
`line()`	the line number (from 1)
`column()`	the column number (from 0)
`endcolumn()`	the ending column number (exclusive)
`start()`	the character offset (from 0, inclusive)
`end()`	ending character offset (exclusive)

Table 18.1: Methods of `Token`.

A token provides the methods listed in Table 18.1. For example:

```
>>> t[5].string()
'J.'
>>> t[5].line()
2
>>> t[5].column()
9
```

One can compute the space between two adjacent tokens by subtracting the endcolumn of the first from the column of the second:

```
>>> t[5].column() - t[4].endcolumn()
0
>>> t[2].string()
'said'
>>> t[3].column() - t[2].endcolumn()
2
```

This is legitimate only if the two tokens are on the same line.

18.2.2 Algorithm

To be robust to OCR errors, tokens typically mix alphanumeric and punctuation characters. The only exception is that a non-peripheral hyphen will break a token into two pieces. The definition in detail is as follows:

- *Whitespace* is a sequence of one or more characters that satisfy `isspace()`.
- A *hyphen* is a sequence of one or more hyphen characters. It is *peripheral* if it is preceded or followed by whitespace or by the beginning or end of the text. It is *embedded* if it is not peripheral.

[1] Internally, the tokenizer also creates `'space'` and `'newline'` tokens, but they are not returned. Newlines are implicit in the line numbers, and spaces can be reconstructed from the column numbers.

- A *separator* is whitespace or an embedded hyphen. Note that a peripheral hyphen is not a separator, and will be included as part of another token.
- A *regular token* is a maximal sequence of non-separators. Its type is `word` if it contains any letters, `number` if it contains digits but no letters, and `punct` if it contains neither letters nor digits.
- A *token* is either a regular token or an embedded hyphen.

The tokenizer returns a sequence of tokens. Note that whitespace is discarded. However, the tokenizer does keep track of line numbers; each token has line number as an attribute.

18.3 Stemmer: `seal.stemmer`

18.3.1 Usage

The module `seal.stemmer` contains a morphological analyzer for English inflectional morphology.

The main function is also called `stemmer`:

```
>>> from seal.stemmer import stemmer
>>> stemmer('dogs')
('dog', '-s')
>>> stemmer('baking')
('bake', '-ing')
>>> stemmer('this')
('this', None)
```

The return value is a pair of form (*stem, suffix*). The common values for *suffix* are: `'-s'`, `'-ed'`, `'-ing'`, and `None`. In addition, there are some irregular words whose suffix is `'-en'`, and the words "am" and "are" are assigned the special suffixes `'+1s'` and `'+pl'`, respectively.

18.3.2 Implementation

The general procedure is to strip a suffix, then apply a stem change.

There are two tables, loaded from files. The word table maps words to stem-suffix pairs. The stem table maps stems to stems.

In detail, the procedure is as follows. If the word is listed in the word table, one immediately returns the value. Otherwise, use Table 18.2. Notes:

- Patterns match in the order given. It will be noticed that more general patterns are always listed later; they would shadow more specific versions otherwise.
- `;` marks the end of the stem in the pattern.

Pattern	Change	Suffix
`-ss`	–	–
`C*.s`	–	–
`-[oiS];es`	Reg	–s
`-e;s`	–	–s
`-eau;s`	–	–s
`-us`	–	–
`-is`	–	–
`-;s`	–	–s
`[^e]d`	–	–
`C*ed`	–	–
`-eed`	–	–
`-[C/r]red`	–	–
`-;ed`	Reg	–ed
`not(-ing)`	–	–
`-.y;ing`	Reg	–ing
`C*ing`	–	–
`-[C/r]ring`	–	–
`-;ing`	Reg	–ing
`-man;`	`men`	–s

Table 18.2: Suffix patterns

- V is a category in context. It matches `[aeiou]`, but not u immediately preceded by q. It also matches y when it is preceded and followed by `[#aeiou]`, where `#` is word boundary.
- C is a category in context. It matches anything that V does not match.
- S matches `szxh`.
- `[C/r]` represents a single character that matches C but does not match r.
- The "`men`" stem change converts *–men* to *–man.*

The procedure represented by the "`Reg`" stem change is as follows. If the stem is listed in the stem-change table, return the value given there. Otherwise, use the rules listed in Table 18.3.

- The pattern M stands for a monosyllable: a string containing only one V.

Pattern	Replacement
-;i	y
-u;	e
-[aeo]	–
-x;x	ϵ
-x;	ϵ
-[tz]z	–
-z;	e
-ss	–
-s;	e
-[ei]t	–
-v;v	ϵ
-g;g	ϵ
-c;c	ϵ
-[vgc];	e
-f;f	ϵ
-[wre]l|-[ui]al|Mll	–
-l;l	ϵ
-Cl;	e
-r;r	ϵ
-Cr;	e
-th;	e
.[yw];	e
-[yw]	–
-VCC?ic;k	ϵ
-C;C and -\1;\1	ϵ
-Cy.;	e
-CC	–
-iaC;|-u[ai]C;	e
-VVC	–
-[eo][mnr]	–
-;	e

Table 18.3: The Reg stem change.

Chapter 19

Finite-state automata: `seal.fsa`

This chapter documents the module `seal.fsa`. The examples assume that one has done:

```
>>> from seal.fsa import *
>>> from seal.io import ex, contents
```

19.1 Using automata

19.1.1 Basics

The most familiar representation of a finite-state automaton is the state graph. An example is given in Figure 19.1. One can create this automaton manually, as follows.

```
>>> a = DFsa()
>>> a.edge('1', '2', 'the')
<Edge 1 2 the>
>>> a.edge('2', '2', 'big')
<Edge 2 2 big>
>>> a.edge('2', '2', 'red')
<Edge 2 2 red>
>>> a.edge('2', '3', 'dog')
<Edge 2 3 dog>
>>> a.final_state('3')
>>> a.dump()
DFsa:
  ->[0]  1
    [1]  2
    [2]# 3
    1 2 the
    2 2 big
```

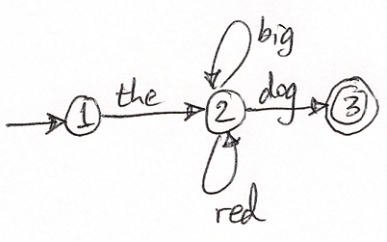


Figure 19.1: The automaton `fsa1`.

```
    2 2 red
    2 3 dog
```

An automaton can be represented as a **transition matrix** that maps states and input symbols to next states. For the automaton just defined, the matrix is:

	the	big	red	dog
1	2			
2		2	2	3
3				

The automaton can be accessed like a matrix:

```
>>> a['1']['the']
<DFsa.State 2 [1]>
>>> a['2']['dog']
<DFsa.State 3 [2]>
>>> a['2']['the']
>>>
```

A row of the matrix is represented by a state, so one can access a state by name using the same idiom:

```
>>> a['1']
<DFsa.State 1 [0]>
```

Final states are distinguished by their value for the attribute `is_final`:

```
>>> a['3'].is_final
True
```

The transition matrix is used to define the behavior of the automaton when given a sequence of input symbols, as follows. The automaton begins in the start state. For each symbol in the input sequence in turn, the new state of the automaton, given old state `q` and input symbol `sym`, is `q[sym]`. If at any point there is no next state, the automaton **blocks**, and the input sequence is rejected. At the end of the input, the sequence is **accepted** if the state is a final state, and rejected otherwise. Here is the full definition of the `accepts` method:

```
def accepts (self, input):
    q = self.start
    for sym in input:
        q = q[sym]
        if q == None: return False
    return q.is_final
```

Here are some examples of the behavior of `accepts`:

```
>>> a.accepts(['the', 'dog'])
True
```

```
>>> a.accepts(['the', 'cat'])
False
>>> a.accepts(['the', 'red', 'big', 'red', 'dog'])
True
>>> a.accepts(['the'])
False
```

The `accepts` method takes a sequence of symbols as input. One can get such a sequence from a string containing whitespace-separated symbols using `split`:

```
>>> 'the big dog'.split()
['the', 'big', 'dog']
```

A string is treated as a sequence of characters, so

```
>>> a.accepts('the')
```

behaves as if it were

```
>>> a.accepts(['t', 'h', 'e'])
```

19.1.2 Fsa file format

An automaton can be stored in a file in **fsa file format**. The module `seal.sh` contains convenience functions for accessing shell functionality, and `seal.ex` contains variables representing various example files. One of the example files is `fsa1`:

```
>>> print(contents(ex.fsa1), end='')
1       2       the
2       2       big
2       2       red
2       3       dog
3
```

The fsa file format is an example of a **tabular format**. The file consists of **records** terminated by single newline characters, and each record is separated into **fields** by single tab characters. The number of fields is one more than the number of tabs. An empty field is created by two tabs with nothing intervening, or by a tab at the beginning or end of the line.

There are two kinds of records in an fsa file. A record containing three fields is an **edge** record, and represents one edge in the graph. A record containing one field is a **final-state** record. The initial state is identified as the state in the first field of the first record (which may be either an edge or a final-state record).

One can load the file simply by passing the filename to the `DFsa` constructor:

```
>>> a = DFsa(ex.fsa1)
>>> a.dump()
```

```
DFsa:
  ->[0]  1
    [1]  2
    [2]# 3
    1 2 the
    2 2 big
    2 2 red
    2 3 dog
```

19.1.3 More about states

Note that state names are strings, not numbers. One can actually use anything one likes as state names, but state names read from files are always strings, so we have used strings to now for consistency's sake. The automaton, viewed as a matrix, is accessed by state name:

```
>>> a['3']
<DFsa.State 3 [2]>
```

In the printed representation of the state, the "3" is the state's name, and the "2" in brackets is its **index**. The automaton contains a list of states, in order of creation, and the index is the position of the state in that list:

```
>>> a.states
[<DFsa.State 1 [0]>, <DFsa.State 2 [1]>, <DFsa.State 3 [2]>]
>>> q = a.states[2]
>>> q
<DFsa.State 3 [2]>
>>> q.name
'3'
>>> q.index
2
```

Unlike `edge` and `final_state` methods, accessing a state by label does not automatically create new states. It signals an error if there is no existing state with the given label:

```
>>> a['5']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/cl/python/seal/fsa.py", line 137, in state
    return self.state_dict[label]
KeyError: '5'
```

Again, be careful not to confuse strings and numbers:

```
>>> a['2']
<DFsa.State 2 [1]>
>>> a[2]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/cl/python/seal/fsa.py", line 137, in state
    return self.state_dict[label]
KeyError: 2
```

To create a new nonfinal state, use the method `state`. It takes a name as argument, and returns the state that has that name, creating a new state if necessary. Any immutable object can be used as a label.

```
>>> a.state('6')
<DFsa.State 6 [3]>
>>> a.state('hi')
<DFsa.State hi [4]>
>>> a.state(2)
<DFsa.State 2 [5]>
>>> a.state(frozenset([1,2,4]))
<DFsa.State {1,2,4} [6]>
>>> for q in a.states: print(repr(q))
...
<DFsa.State 1 [0]>
<DFsa.State 2 [1]>
<DFsa.State 3 [2]>
<DFsa.State 6 [3]>
<DFsa.State hi [4]>
<DFsa.State 2 [5]>
<DFsa.State {1,2,4} [6]>
```

One can "clean up" the state names by calling the method `rename_states`. It sets each state name to be the string corresponding to the state's index:

```
>>> a.rename_states()
>>> a.states
[<DFsa.State 0 [0]>, <DFsa.State 1 [1]>, <DFsa.State 2 [2]>, <DFsa.State 3 [3]>, <
>>> a.states[0].name
'0'
>>> a.states[0].index
0
```

19.1.4 Nondeterministic automata

Suppose we wish to define an automaton that accepts any string of 0's and 1's that ends in "01." The easy way to do it is with the automaton `fsa2`, shown in Figure 19.2. It consumes 0's and 1's for a time, then nondeterministically "guesses" that a given "0" is the next to last symbol in the input. If it guesses right, and if that "0" is immediately followed by a "1," then the automaton arrives in a final state at the end of the input, and the string is accepted.

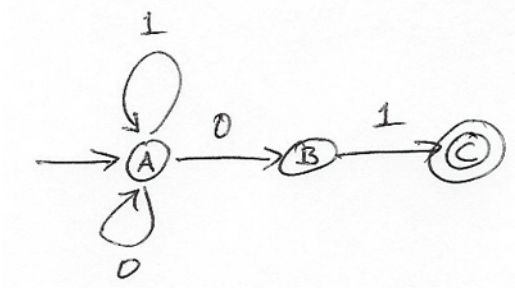


Figure 19.2: Automaton `fsa2`, an example of a nondeterministic automaton.

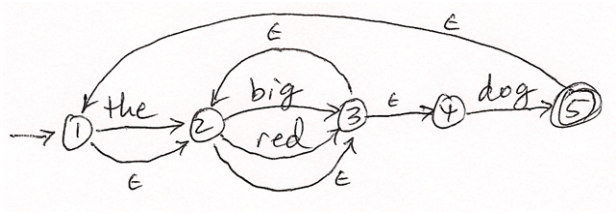


Figure 19.3: The automaton `ex.fsa3`, containing ϵ-edges.

Suppose that the string in fact ends in "01," but the automaton guesses wrong. The result is an alternative computation that ends in failure. Hence we must be more explicit about what it means for an automaton to accept a string: it accepts an input string if there is *any* valid computation that leads to success. The existence of alternative computations that end in failure is immaterial.

The previous automaton is **nondeterministic** because there are two edges out of state "A" that are both labeled "0." In general, an automaton is nondeterministic if there is any state that has multiple outgoing edges with the same label.

(Note that an otherwise deterministic automaton that had, say, two edges labeled "0" both of which go from state "A" to state "B" would satisfy our definition of nondeterminism. To keep the definition simple, we indeed consider such an automaton to be nondeterministic, even though the nondeterminism is in a sense spurious.)

There is one other way in which an automaton may be nondeterministic. It may contain **epsilon edges**. The automaton `ex.fsa2`, shown in Figure 19.3, provides an example.

A (possibly) nondeterministic automaton is represented by the class `Fsa`,

rather than DFsa. For example, we may load fsa1 as an Fsa, and add an edge to make it nondeterministic:

```
>>> a = NFsa(ex.fsa1)
>>> a.edge('2', '3', 'red')
<Edge 2 3 red>
>>> a.dump()
NFsa:
  ->[0]  1
    [1]  2
    [2]# 3
    1 2 the
    2 2 big
    2 2 red
    2 3 dog
    2 3 red
```

An edge is an ϵ-edge if its label, coerced to a boolean, is False. That is, the labels None, '', False, 0, (), etc., are all equivalent. (The label '0', however, is not boolean false.) The label parameter for the edge method defaults to None, so one can also create an ϵ-edge by omitting the label.

```
>>> a.edge('1', '2')
<Edge 1 2 None>
>>> a.dump()
NFsa:
  ->[0]  1
    [1]  2
    [2]# 3
    1 2
    1 2 the
    2 2 big
    2 2 red
    2 3 dog
    2 3 red
```

If we try to add either kind of edge to a DFsa, an error is signalled:

```
>>> d = DFsa(ex.fsa1)
>>> d.edge('2', '3', 'red')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/cl/python/seal/fsa.py", line 103, in edge
    return src.edge(label, dest)
  File "/cl/python/seal/fsa.py", line 85, in edge
    raise Exception, 'Attempt to add multiple edges with same label'
Exception: Attempt to add multiple edges with same label
>>> d.edge('1', '2')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/cl/python/seal/fsa.py", line 103, in edge
    return src.edge(label, dest)
  File "/cl/python/seal/fsa.py", line 82, in edge
    if not label: raise Exception, 'Attempt to add empty edge'
Exception: Attempt to add empty edge
```

The next-state operation on an `Fsa` returns a list of states, rather than a single state. That is true even if there is only one next state.

```
>>> a['2']['red']
[<Fsa.State 2 [1]>, <Fsa.State 3 [2]>]
>>> a['2']['big']
[<Fsa.State 2 [1]>]
>>> a['1']['']
[<Fsa.State 2 [1]>]
>>> a['1']['dog']
[]
```

Note that this operation does not automatically follow epsilon edges. There is no version of the next-state operation that follows epsilon edges. Instead, one should convert the `Fsa` to a `DFsa`.

```
>>> d = determinize(a)
>>> d.dump()
DFsa:
  ->[0]  0
    [1]  1
    [2]# 2
    [3]# 3
    0 1 big
    0 1 the
    0 2 dog
    0 3 red
    1 1 big
    1 2 dog
    1 3 red
    3 1 big
    3 2 dog
    3 3 red
>>> d.accepts(['red'])
True
>>> d.accepts(['red', 'dog'])
True
>>> d.accepts(['dog', 'red'])
False
```

19.2 Conversion to DFSA

The call `determinize(a)`, given a nondeterministic automaton a, produces an equivalent deterministic automaton. If a is not ϵ-free, `determinize()` will first call `eliminate_epsilons()` on it.

The call `minimize(d)` takes a deterministic automaton and creates an equivalent automaton that is minimal, in the sense that there is no other equivalent deterministic automaton that has fewer states.

In this section, we examine these three main transformations: ϵ-elimination, determinization, and minimization.

19.2.1 ϵ-Elimination

To convert a nondeterministic automaton to a deterministic automaton, a preliminary step is the elimination of ϵ-edges.

The function `eliminate_epsilons` does the following. Each state in the old automaton is replaced by a set of states, namely, those which can be reached crossing only ϵ-edges. That set of states is known as the **epsilon closure** of the original state. The method `eclosure` computes the epsilon closure of a state. We illustrate with `fsa3` (Figure 19.3).

```
>>> a = NFsa(ex.fsa3)
>>> [q.name for q in sorted(a['1'].eclosure())]
['1', '2', '3', '4']
>>> [q.name for q in sorted(a['2'].eclosure())]
['2', '3', '4']
>>> [q.name for q in sorted(a['4'].eclosure())]
['4']
```

The function `eliminate_epsilons` creates a new automaton whose states are the epsilon closures of the original states. Its edges are computed as follows. If there is an edge $i \xrightarrow{\ell} j$ in the original automaton, and if i' is any new state that has i in the set of original states that it came from, and if j' is the ϵ-closure of j, then there is an edge $i' \xrightarrow{\ell} j'$ in the new automaton. Here is the function definition:

```
def eliminate_epsilons (old_fsa):
    if old_fsa.epsilon_free: return old_fsa

    new_fsa = NFsa()
    table = []

    for q in old_fsa.states:
        if q.index != len(table): raise Exception, "Bad index"
        table.append(new_fsa.intern(frozenset(q.eclosure())))

    for q1 in new_fsa.states:
```

```
        for q in q1.label:
            for e in q.edges:
                if e.label:
                    q1.edge(e.label, table[e.dest.index])
            if q.is_final:
                q1.is_final = True

    return new_fsa
```

Here is an example of its use:

```
>>> b = eliminate_epsilons(a)
>>> b.dump()
NFsa:
  ->[0]  {1,2,3,4}
    [1]  {2,3,4}
    [2]  {4}
    [3]# {1,2,3,4,5}
    {1,2,3,4} {2,3,4} big
    {1,2,3,4} {2,3,4} red
    {1,2,3,4} {2,3,4} the
    {1,2,3,4} {1,2,3,4,5} dog
    {2,3,4} {2,3,4} big
    {2,3,4} {2,3,4} red
    {2,3,4} {1,2,3,4,5} dog
    {4} {1,2,3,4,5} dog
    {1,2,3,4,5} {2,3,4} big
    {1,2,3,4,5} {2,3,4} red
    {1,2,3,4,5} {2,3,4} the
    {1,2,3,4,5} {1,2,3,4,5} dog
```

Incidentally, `eliminate_epsilons` immediately returns the input automaton if it is already ϵ-free. This is possible because `Fsa`s keep track of whether they are ϵ-free or not. When an Fsa is created, it has no edges, hence is ϵ-free. States record which fsa they belong to. When an ϵ edge is added to a state, the state flags its automaton as no longer being ϵ-free. This assumes that states and edges are never deleted from an automaton, and states are never transplanted from one automaton to another.

19.2.2 Determinization

Here is the definition of the function `determinize`:

```
def determinize (x):
    if not x.epsilon_free:
        x = eliminate_epsilons(x)
        x.rename_states()
```

```
5           x = determinize1(x)
6           x.rename_states()
7           return x
```

We have already discussed ϵ-elimination and relabeling; only determinization proper (the function `determinize1`) remains to be discussed.

States Q of the new automaton are labeled with sets of states from the old automaton. For simplicity, we treat Q as being a set of states from the old automaton. We begin by adding the state $\{q\}$, where q is start state of the old automaton. Then we add edges to the new state Q. The outgoing edges from Q are of form $Q \xrightarrow{\ell} R$, where ℓ is one of the input symbols and R is the set of old states r such that $q \xrightarrow{\ell} r$ is an edge in the old automaton for some $q \in Q$. A new state is final just in case it contains an old state that is final. Here is the complete function definition:

```
1   def determinize1 (old_fsa):
2       new_fsa = DFsa()
3       q = new_fsa.intern(frozenset([old_fsa.start]))
4       ndone = 0
5
6       while ndone < len(new_fsa.states):
7           q1 = new_fsa.states[ndone]
8           ndone += 1
9           table = {}
10          for q in q1.label:
11              for e in q.edges:
12                  if e.label in table:
13                      table[e.label].add(e.dest)
14                  else:
15                      table[e.label] = set([e.dest])
16          for (label, dests) in table.iteritems():
17              q1.edge(label, new_fsa.intern(frozenset(dests)))
18          if q.is_final: q1.is_final = True
19
20      return new_fsa
```

To illustrate the behavior of `determinize1`, we use `fsa2`. (The automaton `b` of the previous example is already deterministic.)

```
1   >>> a = NFsa(ex.fsa2)
2   >>> b = determinize1(a)
3   >>> b.dump()
4   DFsa:
5     ->[0]  {A}
6       [1]  {A,B}
7       [2]# {A,C}
8       {A} {A} 1
```

```
{A} {A,B} 0
{A,B} {A,B} 0
{A,B} {A,C} 1
{A,C} {A} 1
{A,C} {A,B} 0
```

19.2.3 Minimization

Within every equivalence class of automata (an equivalence class being the set of automata that generate a given language), there is a unique minimal automaton, in the sense of an automaton with the fewest states. The minimization algorithm finds that automaton for any given deterministic automaton.

In a deterministic automaton, a given string x maps a state q to a unique state $\delta*(q, x)$. We define the **language of a state** q to be the set of strings that take q to a final state; that is, the set of strings x such that $\delta*(q, x)$ is a final state. By this definition, the language of the automaton is obviously equal to the language of its start state.

We define two states to be **equivalent** if they have the same language. Two states have the same language L just in case every string x either takes both states to a final state (in which case $x \in L$) or takes both to a nonfinal state (in which case $x \notin L$). To avoid having a special case for blocking, we add a special "sink" state $\perp$. For any state q and input symbol w such that $q[w]$ is undefined, we define $q[w] = \perp$. In particular, $\perp[w] = \perp$ for all input symbols w. Once a string leads to $\perp$, it stays there. Moreover, $\perp$ is nonfinal, so any string that leads to $\perp$ is rejected.

Two states are **distinguished** by a string x just in case x takes one of the states to a final state, and the other state to a nonfinal state. The idea of the algorithm is to systematically find distinguishable state pairs, which we call **incompatible pairs**. When all incompatible state pairs have been identified, all remaining state pairs involve equivalent states.

Systematicity is achieved by recursing on string length. We first identify all state pairs that are distinguished by strings of length zero. There is only one string of length zero, the empty string, and it distinguishes a state pair only if one of the states is final and the other is nonfinal.

Then we recurse. Assume that we know all state pairs that are distinguished by strings of length $\leq n$. We will identify any additional state pairs that are distinguished by strings of length n.

Consider an input symbol w, and states q and r with

$$q[w] = s,$$

$$r[w] = t.$$

If q and r are equivalent, then clearly s and t are equivalent. Namely, q and r being equivalent means that every string $x = wy$ takes q and r to the same kind of state (final or nonfinal); hence every string y takes s and t to the same kind of state. Hence if st is an incompatible pair, then qr must be. If we propagate

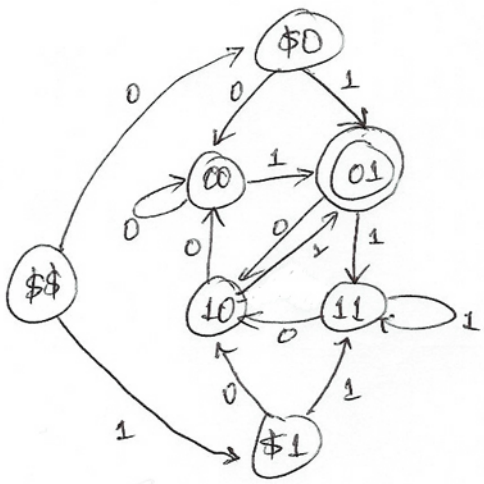


Figure 19.4: Automaton `fsa4`, a DFSA that recognizes strings ending in 01.

---

incompatibility in this way, we will eventually identify every incompatible pair. When the propagation peters out, any remaining pair is equivalent.

We will illustrate using automaton `fsa4`, shown in Figure 19.4. The states of this automaton intuitively represent the most two recently encountered input symbols, and the automaton is in a final state only if the last two symbols were "01." That is, the automaton is equivalent to `fsa2`. Figure 19.5 shows the same automaton with single-letter state names, which will be more convenient for illustrating minimization.

Propagation goes "backwards" along edges: incompatibility between $q[w]$ and $r[w]$ implies incompatibility between q and r. Hence we construct an **incompatibility table** of "reverse edges." The table is indexed by state and input symbol, and entry (s, w) contains all source states q such that $q[w] = s$. Here is the table for `fsa4`:

	0	1
a		
b	a	
c	bce	
d		bce
e	dfg	
f		dfg
g		a

For example, there is an edge $c \xrightarrow{1} d$, hence the entry "c" in the cell $(d, 1)$.

Here is how we use the incompatibility table. Suppose we determine that d and f are incompatible. Then we compare the rows for d and f:

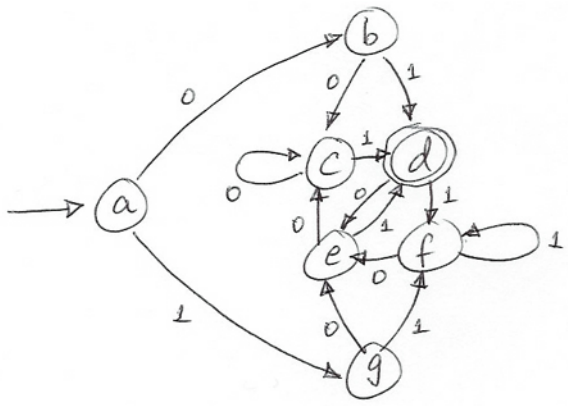


Figure 19.5: Automaton `fsa4` with states renamed for convenience.

---

d	bce
f	dfg

Any states q and r in the same column are such that $q[w] = s$ and $r[w] = t$, where $s = d$ and $t = f$ or the other way around. In short, since d and f are incompatible, it follows that q and r are incompatible. In particular, we propagate incompatibility to the following pairs: *bd, bf, bg, cd, cf, cg, ed, ef, eg.*

The incompatibility table is implemented as the class `Incompatibility`. Here is an example of its use. Note that states are represented by their index.

```
>>> a = DFsa(ex.fsa4)
>>> t = Incompatibility(a)
>>> a['d']
<DFsa.State d [4]>
>>> a['f']
<DFsa.State f [6]>
>>> p = pair(4,6)
>>> for newp in t.propagate(p): print(newp)
...
(2, 1)
(3, 2)
(5, 2)
(4, 1)
(4, 3)
(5, 4)
(6, 1)
(6, 3)
(6, 5)
```

Now we use the incompatibility table to compute a list of compatible pairs. We use two data structures: a map from state pairs to "compatible" or "incompatible", and a "todo" list of the incompatible pairs that have been discovered but not yet used for propagation. Initially, all pairs are marked as compatible and the todo list is empty. Then we systematically go through pairs consisting of one final state and one nonfinal state, mark each as incompatible, and add them to the todo list. Here is the result of initialization on our example:

```
>>> m = Minimizer(DFsa(ex.fsa4))
>>> m.itab.dump()
0 : {} {}
1 : {0} {}
2 : {} {0}
3 : {1,3,5} {}
4 : {} {1,3,5}
5 : {2,4,6} {}
6 : {} {2,4,6}
7 : {7} {7}
>>> m.todo
[(7, 4), (4, 0), (4, 1), (4, 2), (4, 3), (5, 4), (6, 4)]
```

The next step is propagation. One takes a pair from the todo list, and one uses the incompatibility table to propagate to new pairs. For each new pair, one checks whether it has been previously encountered. If so, it is discarded, but if not, it is added to the table of known pairs as well as to the todo list. The process ends when the todo list is exhausted.

Here is the trace of the computation for our example. We initialize with final-nonfinal pairs:

$$da,\ db,\ dc,\ ed,\ fd,\ gd$$

Then we begin propagating. Most pairs propagate nothing; here are the exceptions, noting only the new pairs:

$$\begin{array}{lcl} fd & \rightarrow & fb,\ fc,\ fe,\ gb,\ gc,\ ge \\ gd & \rightarrow & ba,\ ca,\ ea \end{array}$$

After that, no futher propagation is possible.

```
>>> m.propagate()
>>> m.marked.dump()
(1, 0) True
(2, 0) None
(2, 1) True
(3, 0) True
(3, 1) None
(3, 2) True
(4, 0) True
(4, 1) True
```

```
(4, 2) True
(4, 3) True
(5, 0) True
(5, 1) None
(5, 2) True
(5, 3) None
(5, 4) True
(6, 0) None
(6, 1) True
(6, 2) None
(6, 3) True
(6, 4) True
(6, 5) True
```

When the final list of incompatible pairs has been computed, every pair not on the list is equivalent. One creates a mapping from old states to new states, such that equivalent old states get mapped to the same new state. That mapping is used to copy edges from the old automaton to the new automaton, as well as final-state information.

For our example, the compatible pairs are:

cb, eb, ec, fa, ga, gf

We go through these pairs, assigning new-state indices to the members of each, so that the members of a pair both receive the same index. The pair *cb* causes us to create a new index (0) and assign it to both *b* and *c*. The pair *eb* causes us to assign that index to *e*, and the pair *ec* causes no new assignments, but is campatible with previous assignments. The pair *fa* causes a new index (1) to be created; it is further assigned to *g* when we encounter *ga.* The result is:

a	*b*	*c*	*e*	*f*	*g*
1	0	0	0	1	1

```
>>> m.create_map()
>>> m.state_map
[0, 1, 0, 1, 2, 1, 0]
```

We then create new indices for any old states that have not yet been assigned an index. In this case, only *d* remains. The new automaton has three states. State 0 corresponds to old states *b, c, e;* state 1 corresponds to old states *a, f, g;* and state 3 corresponds to old state *d.*

```
>>> out = m.create_newfsa()
>>> out.dump()
DFsa:
  ->[0]  0
    [1]  1
    [2]# 2
```

```
0 0 1
0 1 0
1 1 0
1 2 1
2 0 1
2 1 0
```

For the table where we keep track of state-pair compatibility, and for the mapping from state pairs to new states, an upper triangular matrix (UTM) provides an efficient data structure. It consists of an array with as many cells as state pairs (namely, $n(n-1)$ for n the number of states), and a pair (i, j) of state indices maps to the array location

$$\frac{i(i-1)}{2} + j.$$

It is assumed that $i > j$.

Here is why that calculation of array location maps every index pair to a unique array location. The order of pairs is

$$(1,0), (2,0), (2,1), (3,0), (3,1), (3,2), \ldots$$

There is one pair with $i = 1$, two pairs with $i = 2$, and so on. Hence there are zero pairs that precede $(1, 0)$, one pair that precedes $(2, 0)$, $1 + 2$ pairs that precede $(3, 0)$, and so on. In general, there are $1+2+\ldots+n = n(n+1)/2$ pairs that precede $(n + 1, 0)$. The array location is equal to the number of preceding pairs in the enumeration.

19.3 Finite-state transducers

19.3.1 Definition, transductions

A finite-state transducer (FST) has an input alphabet Σ_1 and output alphabet Σ_2; a set of states Q, one of which is the initial state, and some number of which are final states; and a set of edges of form $(q_1, \alpha, \beta, q_2)$ with $q_1, q_2 \in Q$, $\alpha \in \Sigma_1*$, and $\beta \in \Sigma_2*$. An FST T defines a relation $R_T(\alpha, \beta)$ in $\Sigma_1* \times \Sigma_2*$. Two FSTs are defined to be equivalent just in case they define the same relation.

For a transducer, an *empty edge* is one labeled $\epsilon : \epsilon$.

Rational transduction. A *transduction* is a function from strings to sets of strings. There is an obvious one-one correspondence between string relations R and transductions f_R, where

$$f_R(\alpha) \triangleq \{\beta \mid R(\alpha, \beta)\}.$$

A transduction computable by an FST is called a *rational transduction.*

Rational function. A relation R is a function just in case any input that R associates with an output, it associates with a unique output. We write $R(\alpha)$ for that unique output:

$$R(\alpha) \triangleq \begin{cases} \beta & \text{if } R(\alpha, \beta) \\ \bot & \text{if } \alpha \text{ is not in the domain of } R \end{cases}$$

An FST is *functional* if it computes a function. A *rational function* is a function computed by some FST. Note that a rational function is not a transduction: a rational function maps strings to strings, whereas a transduction maps strings to sets of strings. But we have:

$$f_R(\alpha) = \{R(\alpha)\}.$$

19.3.2 Derived FSAs

Underlying FSA. The *underlying FSA* of an FST results from viewing edges $(q_1, \alpha, \beta, q_2)$ as FSA edges labeled with symbol pairs: $(q_1, \alpha{:}\beta, q_2)$.

Projections. The *first projection* is the FSA obtained by replacing transducer edges $(q_1, \alpha, \beta, q_2)$ with FSA edges (q_1, α, q_2). The *second projection* keeps β instead of α.

Letter transducer. Note that edges of an FST are labeled with *strings,* not single symbols. An FST whose edges are labeled with single symbols or ϵ is called a *letter transducer.* Any FST can be converted to an equivalent letter transducer, by replacing complex-string edges with sequences of letter edges.

Empty-edge elimination. Empty edges $\epsilon : \epsilon$ can be eliminated by applying the ϵ-elimination algorithm to the underlying FSA. Edges labeled $\alpha : \epsilon$ or $\epsilon : \beta$ for nonempty α, β, may still remain.

19.3.3 Basic operations on FSTs

Union. An FST computing the union of two FSTs T_1 and T_2 can be constructed by creating a new initial state with empty edges leading to the initial states of T_1 and T_2.

Inversion. Given an FST computing T, an FST computing T^{-1} can be constructed by interchanging the input and output labels on all edges.

Composition. The *composition* $T_1 \circ T_2$ of two transductions is the function that maps α to $T_2(T_1(\alpha))$. Given FSTs T_1 and T_2, an FST T_3 that computes their composition can be constructed as follows. First, convert each FST to a letter transducer and eliminate empty edges; let T_1' and T_2' be the results. We now construct an FST T_3 representing the composition. Its states are pairs

(q_1, q_2) where q_1 is a state of T_1' and q_2 is a state of T_2'. We keep a to-do list of states needing expansion. The initial state of T_3 pairs the initial state of T_1' with the initial state of T_2'; it is the first entry in the to-do list. Then, until the to-do list is empty, pop a state (q_1, q_2) off the to-do list and do the following, where $a, b, c \neq \epsilon$.

- If q_1 and q_2 are both final states, add (q_1, q_2) to the list of final states for T_3.
- For every c such that (q_1, a, c, r_1) is an edge of T_1' and (q_2, c, b, r_2) is an edge of T_2', intern state (r_1, r_2) and add edge $((q_1, q_2), a, b, (r_1, r_2))$.
- For every edge (q_1, a, ϵ, r_1) in T_1', intern state (r_1, q_2) and add edge $((q_1, q_2), a, \epsilon, (r_1, q_2))$.
- For every edge (q_2, ϵ, b, r_2) in T_2', intern state (q_1, r_2) and add edge $((q_1, q_2), \epsilon, b, (q_1, r_2))$.

To intern a state, do nothing if it is already present. Otherwise, add it to T_3 and also add it to the to-do list.

19.4 The Fst class

The class `seal.fsa.Fst` inherits from `seal.fsa.Fsa`. An Fst can be created from a file. The format is similar to the Fsa format, except that there are two label columns instead of one. For example, here are the contents of `ex.fst1`:

```
1       2       the     d
2       1               er
1       3       big     gross
3       1               e
1       1       dog     Hund
1
```

One can load it and examine it just as with an Fsa:

```
>>> fst = Fst(ex.fst1)
>>> fst.dump()
Fst:
  ->[0]# 1
     [1]  2
     [2]  3
     1 1 dog : Hund
     1 2 the : d
     1 3 big : gross
     2 1  : er
     3 1  : e
```

One can call the transducer as a function. The output is a list of symbol sequences.

```
>>> fst(['the', 'big', 'dog'])
[['d', 'er', 'gross', 'e', 'Hund']]
```

Part VI

Grammars

Chapter 20

Features: `seal.features`

This chapter documents the module `seal.features`. The examples assume you have done:

```
>>> from seal.features import *
>>> from seal.io import iter_tokens, StringIO
```

20.1 Categories and values

To handle phenomena such as agreement and movement, we need to enrich syntactic categories with features. We take a limited approach, in which only non-recursive features are permitted.

20.1.1 Atoms and atom sets

The values of features will be strings or sets of strings. Sets of strings represent nodes in a lattice, with `None` representing bottom and the distinguished string `'*'` representing top.

The class `AtomSet` is used to represent a set of strings. `AtomSet` is a specialization of `tuple`. An atom set will not work correctly unless its elements are lexicographically sorted; use the function `atomset()` to create one.

```
>>> x = atomset(['sg', 'du', 'pl'])
```

Atom sets print out in notation that looks like this:

```
>>> x
du/pl/sg
```

The function `atomset()` returns a string instead of an `AtomSet` if it is given a singleton list:

```
>>> atomset(['hi'])
'hi'
```

An atom set behaves like a sorted tuple of strings:

```
>>> len(x)
3
>>> x[0]
'du'
>>> x[2]
'sg'
>>> 'du' in x
True
```

The basic operation on atom sets is to take their **meet**, or intersection. This is done with the "*" operator.

```
>>> y = atomset(['du', 'pauc', 'pl'])
>>> x * y
du/pl
```

The meet operation is symmetric.

```
>>> y * x
du/pl
```

It also works with atoms as second argument.

```
>>> x * 'du'
'du'
>>> x * '*'
'*'
>>> x * 'foo'
>>>
```

There is also a "join" operation (union):

```
>>> x + y
du/pauc/pl/sg
>>> x + 'foo'
du/foo/pl/sg
>>> x + '*'
'*'
```

20.1.2 Values

A general value is one of: `'*'` (top), an atom set, a string, or `None` (bottom). Three functions are provided that handle general values.

Meet. The function `meet()` returns the meet of two values. The meet is the intersection, viewing the values as sets of atoms.

```
>>> meet('du', x)
'du'
>>> meet('du', 'pl')
>>> meet('*', x)
du/pl/sg
>>> meet(None, x)
>>>
```

Join. The function `join()` returns the join of two values. The join is the union, viewing the values as sets of atoms.

```
>>> join('du', x)
du/pl/sg
>>> join('du', 'pl')
du/pl
>>> join('*', x)
'*'
>>> join(None, x)
du/pl/sg
```

Subsumption. The function `subsumes()` tests whether one value subsumes another. The subsuming value is more general: a superset, viewed as a set of atoms.

```
>>> z = x + y
>>> subsumes(z, x)
True
>>> subsumes(x, z)
False
>>> subsumes(x, x)
True
```

20.1.3 Category

A category consists of a **type** (symbol) and a list of **features**. An example is `v[sg,i,0]`. Instead of using named attributes, we use positional attributes, and implement categories as tuples. For the example just given, the tuple is `('v', 'sg', 'i', '0')`.

More precisely, we define the class `Category` to be a specialization of `tuple`, and we define the method `__repr__()` so that categories print out in the square-bracket format.

```
>>> cat = Category(['np', x, 'fem'])
>>> cat
np[du/pl/sg,fem]
```

A category can be accessed the same way one accesses a tuple:

```
>>> cat[0]
'np'
>>> cat[1]
du/pl/sg
>>> len(cat)
3
```

The features by themselves can be accessed this way:

```
>>> cat[1:]
(du/pl/sg, 'fem')
```

20.1.4 Variables and bindings

The categories in a rule may contain variables, in addition to constant values. An example of a rule with variables is the following. Here we use the convention that variables are capitalized and constants are lowercase.

```
vp[F] -> v[F,i,0]
```

We permit variables *only* in categories in rules. They may not appear in categories in trees.

In this and in subsequent examples, we use lowercase category names, such as `vp` and `v` in the example just given. We have no choice where feature values are concerned: a capitalized feature value would be interpreted as a variable. But the same is not true for category names. We could capitalize category names, if desired, and they would continue to be interpreted as category names.

We choose a representation for variables that maximizes simplicity. When we digest a rule, we number the variables that we encounter, and we use the variable number (starting from 0) to represent the variable. This has the virtue that a set of bindings for a variable can simply be a list, indexed by the variables. For example, the rule

```
vp[F] -> v[F,i,P] pp[P]
```

is internally represented as

```
('vp', 0) -> ('v', 0, 'i', 1) ('pp', 1)
```

If the variable `F` has value `'sg'` and `P` has value `'to'`, then the bindings are represented by the list

```
['sg', 'to']
```

Here is an example of creating a category that contains a variable:

```
>>> v = Category(['v', 0, 'i', '0'])
>>> v
v[X0,i,0]
>>> v[0]
```

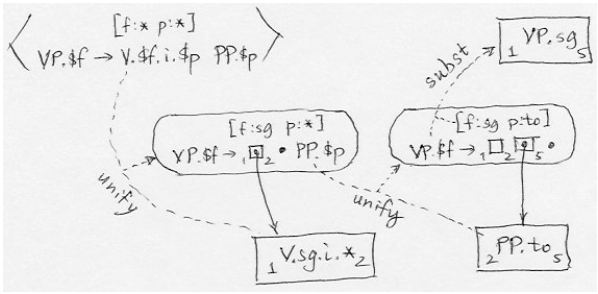
```
'v'
>>> v[1]
0
>>> v[3]
'0'
```

Note that variable `0` prints out as `X0`.

20.2 Unification

20.2.1 Overview

Categories do not have to be identical to match. Consider the following example.



We begin with the node $_1$`v[sg,i,*]`$_2$. Note that "`*`" is a wildcard value: it matches any value. After creating this node, the parser performs the `start` action, which looks up continuations of `V[sg,i,*]`. It finds the rule shown. Written as tuples, the rule categories and child-node category look like this:

```
('vp', 0) -> ('v', 0, 'i', 1) ('pp', 1)
('v', 'sg', 'i', '*')
```

The rule also contains bindings for the variables. Initially, both values are wildcards: `['*', '*']`. Matching the child category against the first righthand side category is called **unification**:

```
('v', 0, 'i', 1) * ('v', 'sg', 'i', '*')
```

This is equivalent to replacing the variables with their values, and comparing each of the corresponding pairs of features. If all pairs match, a new set of bindings is created:

```
('v', '*', 'i', '*') * ('v', 'sg', 'i', '*')
b[0] = '*' * 'sg'
b[1] = '*' * '*'
```

Unification is a non-destructive process. Its output is the new set of bindings. In this case:

```
['sg', '*']
```

The `start` operation creates the first edge. The next step is to `combine` that edge with the second child. We unify the category after the dot with the category of the second child:

```
('pp', 1) * ('pp', 'to')
```

which is:

```
('pp', '*') * ('pp', 'to')
b[1] = '*' * 'to'
```

The unification succeeds, and the output is the set of bindings:

```
['sg', 'to']
```

The result of the `combine` operation is the new edge, with the dot at the end.

Finally, we call the `complete` operation on the finished edge. This creates a new node whose category is obtained by **substituting** the edge bindings into the lefthand side category:

```
('vp', 0) * ['sg', 'to'] = ('vp', 'sg')
```

20.2.2 Meet

With this overview in mind, we turn to a more detailed consideration of the implementation. The most basic function is **meet**, which we have already discussed. It combines two values u and v. Specifically, if $u = v$, it returns u, and if either u or v is the wildcard, it returns the other one. Otherwise, it fails (returns `None`).

```
>>> n1 = Category(['n', 0, atomset(['du', 'pl'])])
>>> n1
n[X0,du/pl]
>>> n2 = Category(['n', 'fem', atomset(['sg', 'pauc', 'pl'])])
>>> n2
n[fem,pauc/pl/sg]
>>> meet(n1[2], n2[2])
'pl'
>>> meet(n1[2], '*')
du/pl
>>> meet(n1[2], None)
```

20.2.3 Unify

The function **unify**(x, y, b) takes two categories and a set of bindings, and returns a new set of bindings if the categories match, or `None` if they do not match. Specifically:

- Make a fresh copy of the bindings, so that updates to the bindings do not affect the original.
- It fails if the types are different: i.e., if $x[0] \neq y[0]$.
- Otherwise, it calls `meet()` on each element $u = x[i]$ and $v = y[i]$, for $i > 0$. If u is a variable, call it "the variable," and let u be its value: $u = b[u]$.
- If v is a variable, signal an error
- Let the new value be `meet`(u, v, b); fail if `meet` fails.
- If there is a variable, store the new value back into b.

The return value is the new set of bindings, or `None` on failure.

```
>>> b = unify(n1, n2, ['*'])
>>> b
['fem']
```

20.2.4 Subst

Next, we require the function **subst**(b, x), which is used by `complete()` to create the category for a new node. It returns a copy of the category (tuple) in which each variable is replaced with its value.

```
>>> n1
n[X0,du/pl]
>>> subst(b, n1)
n[fem,du/pl]
```

20.3 Declarations

A `Declarations` object supports the following functionality:

- Defining names for atom sets. Top (`'*'`), bottom (`None`), the atoms, and all atomsets that can be formed from them, constitute the feature lattice. Being able to name atom sets means that we can assign a name to any node in the lattice. With the addition of defined names, we can think of feature names as **types**, the extension of a type being the set of atoms that it subsumes.
- Defining the number of attributes that a category takes, along with their types and default values. This permits us to use keyword feature specifications in addition to positional specifications. It is also useful for detecting errors in grammars, when an inappropriate value is assigned to an attribute.

A declaration consists of two pieces: a feature table and a category table.

20.3.1 Feature Table

A `FeatureTable` contains named features, including both atoms and features that name sets of atoms. Each feature may be assigned a default value. The basic method is `define()`. It takes the name to define, its definition, and a default value. The default value must be subsumed by the definition.

```
>>> ftab = FeatureTable()
>>> ftab.define('vform', atomset(['sg', 'pl', 'ing']), 'sg')
>>> print(ftab)
Features:
    <Feature vform ing/pl/sg sg>
```

One can access the table as one accesses a dict.

```
>>> ftab['vform']
<Feature vform ing/pl/sg sg>
```

The value is an object of type `Feature`. It has `name`, `value`, and `dflt` attributes.

```
>>> vform = ftab['vform']
>>> vform.name
'vform'
>>> vform.value
ing/pl/sg
>>> vform.dflt
'sg'
```

One can also access a feature table using the method `intern()`, which records the name as an atom, if it is not already present in the table.

```
>>> ftab.intern('sg')
'sg'
>>> print(ftab)
Features:
    <Feature sg sg sg>
    <Feature vform ing/pl/sg sg>
```

20.3.2 Category Table

A `CategoryTable` contains categories, associated with information about the number of features they take, and type restrictions. Default values come from the type restrictions. The main method is `define()`. It takes the category name and a list of `Parameter` instances. A `Parameter` consists of a name (string) and a type (of class `Feature`).

```
>>> ctab = CategoryTable()
>>> ctab.define('vp', [Parameter('form', vform)])
>>> print(ctab)
Categories:
    <Entry vp[form:vform]>
```

A category table is a specialization of dict. The values are of type `CategoryTable.Entry`.

```
>>> ent = ctab['vp']
>>> ent.name
'vp'
>>> ent.params
[form:vform]
>>> ent.params[0].name
'form'
>>> ent.params[0].type
<Feature vform ing/pl/sg sg>
```

20.3.3 Declarations

A `Declarations` instance combines a feature table and a category table. If the feature table and category table are not provided, empty ones will be created.

```
>>> decls = Declarations(ftab, ctab)
>>> decls.features == ftab
True
>>> decls.categories == ctab
True
>>> print(decls)
Features:
    <Feature sg sg sg>
    <Feature vform ing/pl/sg sg>

Categories:
    <Entry vp[form:vform]>
```

20.4 Scanning

The function `scan_category()` scans a category from a token stream. It uses a syntax in which the only special characters are `[:/,]`. It restores the original syntax after scanning.

```
>>> tokens = iter_tokens(StringIO('np[{},a/b] {hi}'))
>>> scan_category(tokens)
np[{},a/b]
>>> next(tokens)
'{'
```

The function `unscan_category()` writes a category to an outfile in a format that will be correctly scanned. This is actually used by the `__repr__()` method of `Category`.

```
>>> cat = Category(['np', 'hi', atomset(['/', ','])])
>>> cat
np[hi,',','/','/']
```

The `__repr__()` method essentially does the following:

```
>>> from seal.io import outfile
>>> f = outfile()
>>> unscan_category(cat, f)
>>> f.getvalue()
"np[hi,',','/','/']"
```

A `Declarations` instance also has a `scan_category()` method. It allows one to use defined features and keyword features.

For completeness, there is also an `unscan_category()` method, but it is identical to the `unscan_category()` function.

Chapter 21

Attribute-Value Structures: `seal.avs`

The examples assume you have done:

```
>>> from seal.avs import *
```

21.1 Implementation

My current implementation is probably much too complicated. I should do a more traditional implementation in which there are variables only when explicitly required (for re-entrancies and for empty values), and in which an AVS is an AV list combined with a symbol table.

21.1.1 Rationale

We want to support both parsing and generation. We associate an AV *state* with each rule, which contains an AVS for the parent node, and attribute paths indicating where to unify in each of the children.

We wish to be use AVS's to represent semantic translations, which grow as the sentence grows. But to be efficient, we need to be able to assure that the space and time involved in processing any single node does not increase as the tree gets larger.

The approach we have taken is to copy pieces of structure in a lazy fashion. One can construct pathological cases in which the amount that must be copied grows without bound, but most natural cases should require copying only the upper reaches of the input structures.

21.1.2 Data structures

Here is an example of a typed attribute-value structure:

$$\textcircled{0} \begin{bmatrix} \text{foo hi} \\ \text{bar } \textcircled{1} \quad [\text{foo bye}] \\ \text{baz } \textcircled{1} \end{bmatrix} \tag{21.1}$$

Our implemention is similar to the implementation of a `Category`. The atoms are either *variables* (the circled numbers in the example) or *constants* (strings). As with categories, we use the simple expedient of representing variables as integers. The variable 0 has special significance; it represents the root of the structure.

We will at times be dealing with variables from multiple AVS's. Typically one AVS will be the *local* AVS and the other is the *foreign* AVS. A particular numeric value v represents two different variables: one a local variable and the other a foreign variable.

The bracketed structures are *AV lists.* An AV list is a list of attribute-value pairs. An attribute is a string. A value is an atom: that is, either a constant or a variable. For the sake of efficiency, we keep attributes alphabetically sorted.

An AV list also keeps a pointer to the AVS that it belongs to. The variables in an AV list are local to its AVS. An AV list belonging to a foreign AVS is to be considered immutable.

An AVS is just a symbol table. Since variables are ints, we can represent the symbol table as a list. Each value in the symbol table may be an atom, an AV list, or `None`. In the case where the value is another variable, we say that the original variable has been *redirected.* In the case where the value is `None`, the variable is called a *dangling variable.* The numeric value -1 is special, and represents a value that is temporarily unavailable.

The function `parse_avs()` can be used to create the AVS of (21.1) from a string representation, as follows.

```
>>> avs1 = parse_avs('[foo hi; bar [foo bye]; baz = bar]')
```

The AVS prints in its string form:

```
>>> print(avs1)
[bar [foo bye]
 baz = bar
 foo hi]
```

The `raw()` method returns a string showing the internal structure.

```
>>> print(avs1.raw())
AVS 0:
    0 -> [bar:1, baz:1, foo:hi]
    1 -> [foo:bye]
```

To summarize, an attribute value or **atom** is one of:

- a constant, or
- a variable (foreign if the AV list is foreign).

A **variable value** is one of:

- a constant,
- a variable (local),
- an AV list,
- `Top`, or
- -1.

21.2 Unification

The basic operation on AVS's is unification, which essentially takes the meet of two AVS's. For the sake of having a running example, let A be the AVS of (21.1), and let B be the following AVS:

$$⓪ \begin{bmatrix} \text{bar } ① & \begin{bmatrix} \text{cat } ② & \begin{bmatrix} \text{meow } ④ \end{bmatrix} \end{bmatrix} \\ \text{baz } ③ & \begin{bmatrix} \text{dog } ② \end{bmatrix} \end{bmatrix} \tag{21.2}$$

The AVS (21.2) is represented internally as:

```
0  →  [('bar', 1), ('baz', 3)]
1  →  [('cat', 2)]
2  →  [('meow', 4)]                    (21.3)
3  →  [('dog', 2)]
4  →  None
```

21.2.1 Lazy copying

One use for AVS's is to represent semantic interpretations, which grow as the sentence grows. We would also like to use unification interleaved with parsing and generation operations. This is especially important for generation, where the AVS specifies what should be generated. Hence we would like an implementation of unification that is nondestructive, but only copies a limited amount of structure no matter how large the AVS's grow.

The approach we take is to allow a variable's value to be a foreign AV list, but to copy a variable into the working AVS whenever we need to change its value. We keep a temporary *import table* for the working AVS, containing entries of form

$(avs,\ u) \to v$

meaning that variable u of AVS *avs* has been imported as variable v of the working AVS.

Importing a variable v means that every reference to v in the original AVS must be replaced. That is, every AV list containing a reference to v must be imported into the working AVS. We can find such AV lists by keeping track of the *parents* of a variable, defining u to be a parent of v just in case u's value is an AV list containing a reference to v.

Importing an AV list may cause us to import additional variables. Each AV list belongs to a particular AVS, and each variable in the AV list is a local variable with respect to that AVS. When we import the AV list, we must replace its variables with variables that are local to the working AVS.

Let us consider an example. Suppose we wish to create an AV list containing a reference to variable 2 of B. To do so, we need to import $(B, 2)$. If we import $(B, 2)$, we must also import its parents and the parents' values. There are two parents: $(B, 1)$ and $(B, 3)$. Their values contain references to $(B, 2)$ that will need to be replaced, but no new variables. However, we must also import the parent of $(B, 1)$ and the parent of $(B, 3)$—they have the same parent, namely, $(B, 0)$. Assume that the working AVS already has one entry, so that we start with local variable 1. Here are the resulting import table entries:

$(B, 2) \to 1$
$(B, 1) \to 2$
$(B, 3) \to 3$
$(B, 0) \to 4$

Further, we add the following entries to the working AVS:

$1 \to$ `[`$_{\texttt{B}}$ `('meow', 4)]`
$2 \to$ `[('cat', 1)]`
$3 \to$ `[('dog', 1)]`
$4 \to$ `[('bar', 2), ('baz', 3)]`

Notice that the value of variable 1 is the foreign AV list, belong to AVS B. When we access the value, the result will be a *foreign variable* represented by the pair $(B, 4)$.

21.2.2 Normalization

Dereferencing. Dereferencing consists in chasing a chain of redirects until we arrive at a variable whose value is something other than another variable. An error is signalled if -1 is encountered. If the value is a constant, the result of dereferencing is the constant. Otherwise, the result is the variable itself. In short, the possible return values are:

- *constant, constant*
- *v, avlist*
- *v,* `None`.

The first value is the dereferenced atom, and the second is its value. A constant's value is the constant itself. A dereferenced variable's value is an AV list or `None`.

In our example, both variables have AV lists as values, so dereferencing has no effect.

21.2.3 The unification algorithm

We begin by creating a new, empty AVS to hold the result of unification. The two input AVS's are not to be modified. We will consider the unification of A (21.1) and B (21.2). The initial task is to unify $(A, 0)$ with $(B, 0)$. We first import both foreign variables, then we unify the resulting local variables.

Unifying atoms. The first step in unifying two atoms is to dereference each. After dereferencing, each argument is each either a constant, a variable naming an AV list, or a *dangling variable* whose value is `None`.

- If both atoms are one and the same object, we are done. Return the atom.
- Else if either argument is a dangling variable, redirect the dangling variable to the other atom and return the other atom.
- Else if either value is a constant, unification fails.
- Otherwise, we have two AV lists. Redirect the second variable to the first, and set the value of the first to the result of unifying the lists. While unifying the two substructures, the value of the first variable is set to -1, representing "unavailable." If -1 is encountered when dereferencing a variable, we have detected a cycle in the structure, and unification fails.

Unifing AV lists. The first step in unifying two AV lists is to make sure that both are local. If either belongs to a foreign AVS, import it into the local AVS. Then one iterates through the two (local) AV lists together, constructing a new output list. Recall that the attributes are alphabetically sorted. If the alphabetically next key appears in only one of the lists, copy it and its value unmodified into the output list. If it appears in both lists, unify the values, and copy the attribute along with the result of unification into the output list. The values in an AV list are atoms (either variables or constants), and we have already discussed the unification of atoms.

21.2.4 Example

Let us consider the algorithm applied to our example. The first step is merging, resulting in the structure ??.

Unify 0 and 2. We now unify the variables 0 and 2. Both have AV lists as values, so dereferencing has no effect. Redirect 2 to 0, and set the value of 0 temporarily to -1.

$$\begin{array}{lcl} 0 & \rightarrow & -1 \\ 2 & \rightarrow & 0 \end{array}$$

Now we unify the original values of 0 and 2, namely, `['bar', 1, 'baz', 1, 'foo', 'hi']` and `['bar', 3, 'baz', 5]`. Only the first list has a value for `foo`, so that goes unmodified into the result. The values for `bar` are 1 and 3, and the values for `baz` are 1 and 5. Hence we have two recursive unifications to perform.

Unify 1 and 3. The values for 1 and 3 are `['foo', 'bye']` and `['cat', 4]`. There are no common attributes, so the output is simply the concatenation of the two lists. Variable 3 is redirected to 1, and the output is stored in 1.

$$\begin{array}{lcl} 1 & \rightarrow & \texttt{['cat', 4, 'foo', 'bye']} \\ 3 & \rightarrow & 1 \end{array}$$

Unify 1 and 5. Now we unify 1 and 5. Variable 5 is redirected to 1 and 1 is temporarily set to -1:

$$\begin{array}{lcl} 1 & \rightarrow & -1 \\ 5 & \rightarrow & 1 \end{array}$$

The values to be unified are `['cat', 4, 'foo', 'bye']` and `['dog', 4]`. There are no shared attributes, so the unification is again simply the concatenation of the lists. It is stored in 1.

$$1 \rightarrow \texttt{['cat', 4, 'dog', 4, 'foo', 'bye']}$$

Finish unifying 0 and 2. We have now completed the two recursive calls. The value for `bar` is set to 1, and the value for `baz` is also set to 1. The output list is stored in 0. The final outcome is:

$$\begin{array}{lcl} 0 & \rightarrow & \texttt{['bar', 1, 'baz', 1, 'foo', 'hi']} \\ 1 & \rightarrow & \texttt{['cat', 4, 'dog', 4, 'foo', 'bye']} \\ 2 & \rightarrow & 0 \\ 3 & \rightarrow & 1 \\ 4 & \rightarrow & \texttt{['meow', 6]} \\ 5 & \rightarrow & 1 \\ 6 & \rightarrow & \texttt{None} \end{array}$$

21.2.5 Packing

To make future unifications a little more efficient, we may *pack* the result. We first propagate "reachability" from variables to the variables mentioned in their values, starting from variable 0. The result is:

```
[True, True, False, False, True, False, True]
```

That is, variables 0, 1, 4, and 6 are reachable. Then we define replacement variables by numbering the reachable variables. The result is:

```
[0, 1, False, False, 2, False, 3]
```

Finally, we create a reduced symbol table, in which all variables have been replaced with their new numbers.

```
0  →  ['bar', 1, 'baz', 1, 'foo', 'hi']
1  →  ['cat', 2, 'dog', 2, 'foo', 'bye']
2  →  ['meow', 3]
3  →  None
```

This last step can be destructive, as long as we are sure to copy all AV lists from *both* of the original input structures when we do the initial merge.

21.2.6 In Python

Create the second AVS:

```
>>> avs2 = parse_avs('[bar [cat [meow []]]; baz [dog = bar.cat]]')
>>> print(avs2)
[bar [cat [meow []]]
 baz [dog = bar.cat]]
>>> print(avs2.raw())
AVS 1:
    0 -> [bar:1, baz:4]
    1 -> [cat:2]
    2 -> [meow:3]
    3 -> Top
    4 -> [dog:2]
```

Unify:

```
>>> avs3 = unify(avs1, avs2)
>>> print(avs3)
[bar [cat [meow []]
      dog = bar.cat
      foo bye]
 baz = bar
 foo hi]
```

21.3 AV state

An AV state represents an intermediate state during the construction of the AVS for a node with children. The second argument to `parse_avstate()` is the number of children.

```
>>> s = '[subj $1 [foo hi; bar [foo bye]; baz = subj.bar]]'
>>> q = parse_avstate(s, 2)
>>> print(q)
(AvState . * subj - : [subj [bar [foo bye]; baz = subj.bar; foo hi]])
```

The method `extend()` is given the AVS for the next child in line.

```
>>> q2 = q.extend(avs2)
>>> print(q2)
(AvState . subj * - : [subj [bar [cat [meow []]; dog = subj.bar.cat;
foo bye]; baz = subj.bar; foo hi]])
```

Chapter 22

Grammars: `seal.grammar`

This chapter documents the module `seal.grammar`. The examples assume that one has done:

```
>>> from seal.grammar import *
>>> from seal.features import Category, atomset
>>> from seal.io import ex, contents
```

22.1 Lexicon

22.1.1 Lexical entry

A lexical entry has type `Lexicon.Entry`. It consists of a word, a part of speech, and an optional semantic translation.

```
>>> ent = Lexicon.Entry('dog', Category(['n']), 'DOG')
>>> ent.word
'dog'
>>> ent.pos
n
>>> ent.sem
'DOG'
```

22.1.2 Lexicon

A `Lexicon` consists of a set of lexical entries. The basic method is `define()`; it takes a word, a part of speech (category), and an optional semantic value.

```
>>> lex = Lexicon()
>>> lex.define('cat', Category(['n','sg']))
>>> print(lex)
cat n[sg]
```

The lexicon can be accessed by word. The value is a list of entries.

```
>>> lex['cat']
[<Entry cat n[sg]>]
```

An error is signalled if the word is not present.

The length of the lexicon is the number of entries.

```
>>> len(lex)
1
```

For purposes of iteration, the elements of a lexicon are entries.

```
>>> list(lex)
[<Entry cat n[sg]>]
```

22.2 Grammar

22.2.1 Rule

Grammar rules are represented by instances of the class `Rule`. A `Rule` has five attributes: `lhs`, `rhs`, `bindings`, `variables`, and `sem`. The `lhs` is a single category, and the `rhs` is a list of categories. The value for `bindings` is a list containing `*`'s, one for each variable used in the rule. The value for `variables` is a list of string representations for the variables, or `None`. The value for `sem` is an expression.

The constructor takes a lhs, rhs, sem, and a symbol table. The symbol table is a dict that maps variable names to integers from 0 to the size of the table. The symbol table is optional; if omitted, variables are anonymous. The length of the bindings list is the size of the symbol table, if provided. Otherwise, it is one greater than the largest numeric variable occurring in either the lhs or rhs.

```
>>> r = Rule('vp', ['v', 'np'], 'foo')
>>> r.lhs
'vp'
>>> r.rhs
['v', 'np']
>>> r.bindings
[]
>>> r.sem
'foo'
```

22.2.2 Grammar

The `Grammar` class has a similar structure to the `Lexicon` class. Internally, it maintains two indices. A rule of form $X \rightarrow Y_1 \ldots Y_n$ is indexed by X in the lefthand side index, and it is indexed by Y_1 in the righthand side index.

The basic method is `define()`. It takes a lhs, rhs, an optional semantic translation, and an optional symbol table.

```
>>> def C (s): return Category([s])
>>> g = Grammar()
>>> g.define(C('s'), [C('np'), C('vp')])
>>> g.define(C('vp'), [C('v'), C('np')])
>>> print(g)
Start: s

Rules:
    [0] s -> np vp
    [1] vp -> v np
```

The attribute `start` contains the start category. It defaults to the lhs of the first rule defined.

```
>>> g.start
s
```

The method `expansions()` takes a string X and returns the list of rules of form $X \to Y_1 \ldots Y_n$. Note that the input is just a string, not a full category.

```
>>> g.expansions('vp')
[<vp -> v np>]
```

The method `continuations()` returns the list of rules whose righthand side begins with a given symbol. For example:

```
>>> g.continuations('v')
[<vp -> v np>]
```

A grammar also has attributes `declarations` and `lexicon`. The value of `declarations` is generally `None`, unless the grammar is created by the grammar loader (§22.3) from a file that contains declarations.

22.3 Grammar loader

The `GrammarLoader` reads a grammar file. Here is a simple example of the format. This is the contents of `ex.g9.g`. In the section headers (e.g., "`% Features`"), the space following the percent sign is optional, and the capitalization of the section name does not matter.

```
% Features
nform = sg/pl
vform = nform/ing
trans = i/t
bool = +/- default -
% Categories
s   []
np  [form:nform, wh:bool]
```

```
vp  [form:vform]
v   [form:vform, trans:trans]
n   [form:nform]
det [form:nform]
% Rules
s -> np[F] vp[F]
np[F] -> det[F] n[F]
vp[F] -> v[F,i]
vp[F] -> v[F,t] np
% Lexicon
the det
a det[sg]
cat n[sg]
dog n[sg]
dogs n[pl]
barks v[sg,i]
chases v[sg,t]
```

The grammar loader is called by the `Grammar` constructor when a filename is provided. For example:

```
>>> g = Grammar(ex.g9.g)
>>> print(g)
Start: s

Rules:
    [0] s -> np[F,-] vp[F]
    [1] np[F,-] -> det[F] n[F]
    [2] vp[F] -> v[F,i]
    [3] vp[F] -> v[F,t] np[pl/sg,-]

Lexicon:
    a det[sg]
    barks v[sg,i]
    cat n[sg]
    chases v[sg,t]
    dog n[sg]
    dogs n[pl]
    the det[pl/sg]
```

Chapter 23

Grammar Development: `seal.gdev`

23.1 Executable

The usual way to run gdev is from the shell:

```
$ python -m seal.gdev
```

When it starts up, it prints out the usage message, followed by a prompt (`>`). The commands are as follows.

`ls`	List the existing grammars. It looks in the directories on its grammar path for files with suffix "`.g`." The initial path includes the current directory, `/cl/examples`, and `/cl/data/eng`.
`r`	Reload the grammar and sentence files, and reparse.
`n`	Next: go forward one sentence.
`p`	Previous: go back one sentence.
number	Go to the sentence with that number.
grammar	Load the grammar.
expression	Evaluate the given semantic expression in the model.
sentence	Parse and evaluate a temporary sentence.
`c`	Print the current sentence. Discard the temporary sentence, if any.
`g`	Print the grammar.
`m`	Print the model.

`s`	Print the sentences.
`t`	Save the translations to *grammar*`-trans.txt`
`h`	Print a help message. Question mark or an empty command also cause the help message to be printed.
`trace`	Takes zero or more arguments, from the following list: `on` turns tracing on (the default), `off` turns tracing off, `parse` affects tracing of parse-tree construction, `unif` affects tracing of unifications, *number* specifies a particular rule to trace. If both `on` and `off` are specified, the one that comes later overrides the one that comes earlier.
`^D`	Quit.

23.2 Dev

In what follows, the examples assume:

```
>>> from seal.gdev import *
```

When one calls `seal.gdev` from the shell, it instantiates the class `Dev` and calls its `run()` method. The `run()` method repeatedly reads a line from stdin and passes it to the `com()` method. Here is an example. First we instantiate `Dev`:

```
>>> d = Dev()
```

Load grammar `g9`, along with its example sentences:

```
>>> d.com('g9')
```

Show the sentences. The numbers not in brackets indicate how many parses the grammar assigns to the sentence.

```
>>> d.com('s')
[0]   1  a cat barks
[1]   0 *a dogs barks
[2]   1  the cat chases the dog
```

Show the parse tree(s) for the current sentence:

```
>>> d.com('c')

[0]   a cat barks
#Parses: 1
Parse 0:
0   (s
1      (np[sg,-]
```

```
2         (det[sg] a)
3         (n[sg] cat))
4      (vp[sg]
5         (v[sg,i] barks)))
```

23.2.1 Sentences and labels

When the command is the name of a grammar file, `Dev` expects two files to exist: *prefix*`.g` should contain a grammar, and *prefix*`.sents` should contain a list of sentences. Each line of the sentence file is considered to be a sentence, except that empty lines and lines beginning with `#` are ignored. Leading and trailing whitespace is ignored. If the first non-whitespace character is `*`, it indicates that the example is ungrammatical. For example:

```
>>> from seal.io import contents
>>> print(contents(ex.g9.sents), end='')
 a cat barks
*a dogs barks
 the cat chases the dog
```

`Dev` creates a parser from the grammar file, and uses it to parse each of the sentences in the sentence file. The predicted label is `'OK'` if the parser deems the sentence to be grammatical, and `'*'` if the parser rejects it. The predicted labels are compared to the true labels, and the results are printed out.

Chapter 24

English Grammar

24.1 First grammars

Grammars 5, 6, and 7 represent a sequence of grammars covering additional phenomena. In each case, there are three files: for example, `ex.g5`, `ex.lex5`, and `ex.text5`.

Grammar 5 adds pronouns and names, noun modification, a richer set of subcategorization, including complements of adjectives, and subordinate clauses.

```
Root -> S;
Root -> NP;
S -> NP[n:$n] VP[f:$n];
NP[n:$n] -> Pron[n:$n];
NP[n:$n] -> Name[n:$n];
NP[n:$n] -> Det[n:$n] Nom[n:$n];
NP[n:pl] -> NP Conj NP;
Nom[n:$n] -> Adj1 Nom[n:$n];
Nom[n:$n] -> N[n:$n];
VP[f:$f] -> V[f:$f,t:n,s:null];
VP[f:$f] -> V[f:$f,t:y,s:null] NP;
VP[f:$f] -> V[f:$f,t:y,s:$p] NP PP[f:$p];
VP[f:$f] -> V[f:$f,t:y,s:np] NP NP;
VP[f:$f] -> V[f:$f,t:n,s:$p] PP[f:$p];
VP[f:$f] -> V[f:$f,t:n,s:adj] AdjP;
VP[f:$f] -> V[f:$f,t:y,s:$c] NP SC[f:$c];
VP[f:$f] -> V[f:$f,t:n,s:$c] SC[f:$c];
PP[f:$p] -> P[f:$p] NP;
AdjP -> Adj1[s:null];
AdjP -> Adj1[s:$p] PP[f:$p];
Adj1[s:$p] -> Deg Adj[s:$p];
Adj1[s:$p] -> Adj[s:$p];
SC[f:$c] -> C[f:$c] S;
```

```
24  SC[f:inf] -> P[f:to] VP[f:base];
```

Here are examples of coverage (see `text5`):

this dog
*this dogs
these dogs
the dog
the dogs
this dog barks
these dogs bark
*these dogs barks
*these dogs bark the cat
these dogs chase the cat
the black dog
Fido chases the cat
he thinks about the dog
she thinks that the dog chases the cat
*she thinks the dog
*she tells that the dog chases the cat
she tells the dog that the cat barks
the cat thinks
the cat wants to bark
Fido is black
the cat is happy about the toy
we gave the dog a toy
we gave a toy to the dog

Grammar 6 adds only one rule to grammar 5:

```
1  VP[f:$f] -> Aux[f:$f,t:n,s:$v] VP[f:$v];
```

This provides coverage of auxiliary verb sequences in English. Here are examples (`text6`):

Fido chases Spot
Fido has chased Spot
Fido is chasing Spot
Fido will chase Spot
Spot will be chased
Fido will be chasing Spot
Fido will have been chasing Spot
Spot will have been being chased
*Spot will be had been chased

Grammar 7 adds movement: yes-no questions, wh-questions, and relative clauses. Here are examples of its coverage:

what did you chase
which cat did you chase
*what did you bark
did you bark
the dog that Max chased
the dog that chased Max
these black dogs that Max chased

24.2 Numbers

One digit numbers. These are simply the digits *zero, one, ..., nine.* Zero is not embeddable: we cannot say **twenty zero.* Let us define `digit` to exclude zero: it consists of the embeddable digits.

Teens. These are the numbers *ten, eleven, ..., nineteen.* The category is `teen`.

Two digit numbers. They begin with twenty: *twenty, twenty one, ..., twenty nine, ... ninety nine.* Note that "zero" does not count as a digit here: a two-digit number consists of a `tens` and a `digit`. When embedded, wherever we can use a two-digit number, a `teen` or `digit` can also be used. So we define the category `num2` to include two-digit numbers properly speaking, as well as `teen` and `digit`.

Hundreds. Examples: *one hundred, a hundred, one hundred one, one hundred and one, ... one hundred ninety nine, ... nine hundred ninety nine, eleven hundred ninety nine, ... ninety nine hundred ninety nine.* The example *ten hundred* does not really sound bad; perhaps it should not be excluded.

There is an alternation between "one" and "a," though "a" is not a `digit`. What follows "hundred" cannot be "a": **one hundred a.* Also, when embedding a three-digit number, the form beginning with "a" cannot be used: **two thousand a hundred and six.* Let us distinguish between embeddable and non-embeddable three-digit numbers. The non-embeddable case includes the embeddable case, but also ones beginning with "a hundred."

If a number greater than nine precedes the word "hundred," then the result is also not embeddable: we cannot say **six thousand thirteen hundred.*

What follows the word "hundred" may be a `digit`, a `teen`, or a two-digit number: that is, the class `num2`. Between "hundred" and `num2` there is an optional "and." Let us use the category `tail3` for "hundred" followed by optional "and" followed by `num2`.

Let us call the embeddable case `hundreds`. The pattern is a `digit` followed by a `tail3`. The non-embeddable case is `ne-hundreds`, which consists of "a" or `num2` followed by `tail3`. Note that the prefix cannot be omitted: *`hundred six.`

Let us use `num3` for the union of `hundreds` and `num2`.

Thousands. Examples: *a thousand, two thousand, thirteen thousand, ninety nine thousand, six thousand and four, six thousand three hundred, nine hundred ninety nine thousand nine hundred ninety nine.* But not **six thousand thirteen hundred* or **six thousand ninety nine hundred.*

Again, "a" is not permissible when the number is embedded: **four million a thousand three.* The pattern for `thousands` is `num3` followed by `tail4`, where `tail4` is the word "thousand" followed by an optional `num3`. There may be an "and" between "thousand" and the trailing `num3`.

The non-embeddable case is `ne-thousands`, which is "a" followed by `tail4`.

Define `num5` to be the union of `thousands` and `num3`.

Millions and higher. For higher numbers, the pattern is repeated. Define `tail6` to be "million" followed by an optional `num5`, with an optional "and" in between. Then `millions` is `num3` followed by `tail6`, and `ne-millions` is "a" followed by `tail6`. The union of `millions` with `num5` is `num8`.

For `billions`, the helping category is `tail9`, and the result is `num11`. For `trillions` they are `tail12` and `num14`, and so on.

Numbers. The general case, `num`, is `num14` or `ne-trillions` or `ne-billions` or `ne-millions` or `ne-thousands` or `ne-hundreds` or `ne-digit`.

24.3 Translation to German

24.3.1 Example

The book known in English as *Heidi* by Johanna Spyri was originally published in German as two separate volumes: *Heidis Lehr– und Wanderjahre* (Gutenberg ebooks 7500) and *Heidi kann brauchen, was es gelernt hat* (Gutenberg ebooks 7512). This is the beginning of the first chapter of the first volume.

> Vom freundlichen Dorfe Maienfeld führt ein Fußweg durch grüne, baumreiche Fluren bis zum Fuße der Höhen, die von dieser Seite groß und ernst auf das Tal herniederschauen. Wo der Fußweg anfängt, beginnt bald Heideland mit dem kurzen Gras und den kräftigen Bergkräutern dem Kommenden entgegenzuduften, denn der Fußweg geht steil und direkt zu den Alpen hinauf.

Before tackling the first sentence, we should be able to handle:

das freundliche Dorf
the friendly village

das Dorf ist freundlich
the village is friendly

vom freundlichen Dorfe
from the friendly village

das Dorf Maienfeld
the village Maienfeld

der Fußweg führt hoch
the path leads up

vom Dorf führt ein Fußweg hoch
a path leads up from the village

There are judgment questions, such as whether to leave place names untranslated, or to attempt to translate them (e.g., Mayfield for Maienfeld).

One of the things that is needed for German is morphology. For example, we want to map the category-semantics pair (`det.m.dat.sg`, `ein`) to `einem` and back again. The reverse mapping may be ambiguous.

24.3.2 German morphology

Determiners.

	Masc	**Neut**	**Fem**	**Pl**
Nom	dieser	dieses	diese	diese
Gen	dieses	dieses	dieser	dieser
Dat	diesem	diesem	dieser	diesen
Acc	diesen	dieses	diese	diese

This declension is also used for *jener, mancher.* The declension for the definite article differs in neuter singular nom/acc (*das,* not *des*) and fem/pl nom/acc (*die*, not *de*). The plural is used for *viele, beide.*

Adjectives. Adjectives have weak and strong declensions: weak in *das gute Bier,* strong in *gutes Bier.* The strong declension:

	Masc	**Neut**	**Fem**	**Pl**
Nom	guter	gutes	gute	gute
Gen	guten	guten	guter	guter
Dat	gutem	gutem	guter	guten
Acc	guten	gutes	gute	gute

The weak declension:

	Masc	**Neut**	**Fem**	**Pl**
Nom	der gute	das gute	die gute	die guten
Gen	des guten	des guten	der guten	der guten
Dat	dem guten	dem guten	der guten	den guten
Acc	den guten	das gute	die gute	die guten

Chapter 25

Grammar Lab: `seal.glab`

The following examples assume:

```
>>> from seal.glab import *
```

25.1 Invocation

25.1.1 Web interface

First, create a working directory for glab. Also create a subdirectory with your username, and copy the example notebook into it.

```
$ mkdir glab
$ cd glab
$ mkdir abney
```

Launch glab:

```
$ python -m seal.glab
```

To use it, visit `http://localhost:8000/`.

25.1.2 Batch mode

The first line of a `.gl` file is a notebook name prefixed with "`#T`," and each subsequent line is a glab statement. For example, `ex.notebook.gl` contains:

```
#T My Notebook
set _x <a,b,c>
_x . <b,a>
```

It is interpreted as follows:

```
>>> from seal import ex
>>> interpret_file(ex.notebook.gl, show_syntax=True)
| #T My Notebook
| set _x <a,b,c>
: set(_x, seq(a, b, c))
| _x . <b,a>
: concat(_x, seq(b, a))
<a, b, c, b, a>
```

The original line of text is echoed with "|" as prompt, and the parsed expression is echoed with ":" as prompt. Then any return value or error is printed.

By default, echo is on, meaning that each statement and value is printed. It also means that errors are printed instead of terminating processing. Echo can be turned off by providing `echo=False`. In that case, the only printing is what is explicitly done with `show` statements, and any exceptions immediately terminate processing.

25.2 Functionality

GLab displays inputs and outputs in a *notebook*. The inputs are editable blocks of text and the outputs follow, but are not editable. In the following, I will use "|" as a generic prompt, even though there is no actual prompt.

25.2.1 Syntax

An **atom** is one of the following:

- An **operator**, as listed in Table 25.1. Example: `+`
- A **variable**, which must begin with underscore. Example: `_a1`
- A **string** in single or double quotes. There is no significance to the choice between single quotes and double quotes, though the start and end quotes must of course match. If the string contains internal whitespace, it is interpreted as a sequence literal in which the elements of the sequence are the whitespace-separated symbols in the string. If the string contains no internal whitespace, it is interpreted as representing a literal symbol (after trimming any leading and following whitespace). Example: `'foo bar'`
- A **symbol** literal, which is any unquoted word that is not an operator or variable. Example: `a`

Atoms are grouped into **expressions**. The following are the expression types:

1. Two subexpressions with an infix operator between them, representing an operator expression with two operands. Example: `a . b`
2. A subexpression followed by a postfix operator, representing an operator expression with one operand. Example: `a *`

`:`	Cross-product
`*`	Kleene star (suffix operator)
`.`	Concatenation
`x`	Cross-product
`+`	Addition
`-`	Subtraction
`\`	Set difference
`=`	Equality
`in`	Set membership

Table 25.1: The operators, listed from highest to lowest precedence classes. The lines divide the precedence classes; operators in the same precedence class group left to right.

3. A list of subexpressions in vertical bars, representing a size expression. Example: `|{a,b}|`

4. A symbol followed by a parenthesized list of subexpressions, representing a function call. Example: `f(_x, _y)`

5. A symbol followed by a bracketed list of subexpressions, representing a category literal. Example: `VP[sg]`

6. A list of subexpressions in angle brackets, representing a sequence literal. Example: `<c,a,t>`

7. A list of subexpressions in braces, representing a set literal. Example: `{a,b}`

8. A list of subexpressions in slashes, representing a language literal. Example: `/a . b/`

Expression types (1)–(4) are normalized to a function-call form, in which the operator or function symbol serves as operator. In case (3), the operator is `||`. The remaining expression types represent literal objects: categories, sequences, sets, or languages.

A **command statement** cannot be embedded. There are two types of command statement:

- The first expression is a symbol representing a prefix command, as listed in the top half of Table 25.2, and the remaining expressions in the line are its arguments.

- The second expression is an infix command, as listed in the bottom half of Table 25.2. The first expression, along with the third and following expressions, represent arguments of the command.

`set`	Set the value of a variable
`include`	Include another notebook
`incr`	Increment a variable
`show`	Show the value of a variable
`parse`	Parse a sentence
`trace`	Turn on tracing
`good`	Mark a sentence as good
`bad`	Mark a sentence as bad
`results`	Show the results of parsing
`->`	Define a grammar rule
`<-`	Define a lexical entry
`=>`	(I forget)

Table 25.2: The commands. These may appear only at the top level. The ones below the line are infix commands; the others are prefix commands.

At the highest level, a notebook consists of newline-terminated **lines**. A line beginning with `#` is a **comment**. The title of the notebook must be the first line and begin with `#T` followed by a space and the actual title. Every other line is a *statement,* which may be either a command statement or an expression.

25.2.2 Variables and symbols

Atoms are variables or symbols. Variables begin with underscore; symbols do not. Symbols evaluate to themselves. In a notebook, one would see:

```
| a
a
| _a
ERROR: Unbound variable: _a
```

For debugging purposes, we can use the function `interpret()`:

```
>>> interpret('a', '_a')
| a
a
| _a
ERROR: Unbound variable: _a
```

Note that *operator* is a specialization of *symbol,* so any "stray" operators that are not part of a well-formed operator expression are treated as symbol literals.

```
>>> interpret('*')
| *
*
```

A variable can be set using the command `set`.

```
>>> interpret('set _a hi', '_a')
| set _a hi
| _a
hi
```

Note that = is used for equality testing only, not assignment:

```
>>> interpret('set _a hi', '_a = hi')
| set _a hi
| _a = hi
True
```

25.2.3 Sequences, strings, sets

A sequence literal is marked with angle brackets. The elements may be separated by commas, though commas may be omitted if no ambiguity results. (Commas are generally optional wherever they occur.) Sequences evaluate to themselves.

```
>>> interpret('<hi, there>', '<hi there>')
| <hi, there>
<hi, there>
| <hi there>
<hi, there>
```

A quoted string that contains internal whitespace is interpreted as a sequence of symbols separated by whitespace. If there is no internal whitespace, a quoted string is interpreted as a symbol.

```
>>> interpret("'hi there'", "'hi'")
| 'hi there'
<hi, there>
| 'hi'
hi
```

Braces introduce literal sets.

```
>>> interpret('{a,b,c}')
| {a,b,c}
{a, b, c}
```

25.2.4 Operator expressions

Dot is used for concatenation.

```
>>> interpret('<a,b> . <c,d>')
| <a,b> . <c,d>
<a, b, c, d>
```

The operator "in" is used for set membership.

```
>>> interpret('set _S {a,b,c,d,e}', 'a in _S')
| set _S {a,b,c,d,e}
| a in _S
True
```

Plus is used for set union.

```
>>> interpret('set _S {a,b,c}', '_S + {b,c,d}')
| set _S {a,b,c}
| _S + {b,c,d}
{a, b, c, d}
```

25.2.5 Operator precedence

"Multiplication" operators have higher precedence than "addition" operators.

```
define fsa a1 <1 the 2, 2 big 3, 2 black 3,
    3 cat 4, 3 dog 4, 4 F>
accepts(a1, 'the big cat')
computation(a1, 'the big cat')
let sents = <the big cat,
             the black cat,
             the big big black cat>
accepts(a1, sents)
let a2 = (a1 . 'foo')
(8 - ((3 * 2) + 1))
|s|
|sents|
|"dog"|
|"c a t" . "fish"|
let A = {'a', 'b c', 'c a'}
{x in A where |x| = 2}
(A x B)
(A . &)
0
(A *)
(* A)
let B = (a + e + i + o + u)
let C = {a, e, i, o, u}
B = C
(A *) . 'f'
(A *) * f
<'a'>
(g . o:e . o:e . s . e)
{{0}}
```

Numbers, strings, variables (local and global), sets, languages, transductions, regular expressions, FS automata, FS transducers, context-free grammars. If a

set contains only strings, it is of class Language. Taking the Kleene closure of a language produces an InfLang, which is internally represented as an automaton.

An unquoted word is taken to be a variable if it appears in the symbol table, and otherwise is taken to be a string.

A string is coerced to an RE if an RE operator is applied to it.

25.3 Transducers

25.4 Implementation

25.4.1 Expression classes

Expressions are constructed from the following classes. Atomic expressions are subclasses of `str`:

`Symbol`	A symbol
`Op`	An operator
`Var`	A variable

Delimited expressions are subclasses of `tuple`:

`String`	`'...'`
`Expr`	`[...]`
`ParenExpr`	`(...)`
`SetExpr`	`{...}`
`SeqExpr`	`<...>`
`AbsExpr`	`|...|`
`ToplevelExpr`	from beginning to end

In addition, there are special cases for special syntax:

`SetAbstraction`	A set abstraction
`Funcall`	A function applied to arguments

25.4.2 Tokenization

The function `tokenize()` takes a string and returns an iteration containing tokens (strings). It assumes that its input represents a single line of input. If any newlines happen to be present, they are treated like spaces.

```
1 >>> list(tokenize('_a1 = fsa()\n_a1'))
2 ['_a1', '=', 'fsa', '(', ')', '_a1']
```

In detail, the kinds of token are as follows:

- A **special character**, of which there are three sorts:
  - A **separator**, which is any of the following: `| , . :`
  - A **multi-character special**, which is any of the following: `-> <- =>`

  - A **grouping character**, which is any of the following: `( ) [ ] { } < >`

- A **string** in single or double quotes. For example, `'hi there'` or `"bye"`.
- A **word**, which is any stretch of characters excluding whitespace or any of the tokens already mentioned.

Note: in the above example, the spaces around the `=` are essential, since `=` is not a separator:

```
>>> list(tokenize('_a1=fsa()'))
['_a1=fsa', '(', ')']
```

25.4.3 Grouping

After tokenization, pairs of grouping characters are mated to create a syntactic skeleton. Paired square brackets produce a `BracketExpr`, paired parentheses produce a `ParenExpr`, paired braces produce a `SetExpr`, paired angle brackets produce a `SeqExpr`, and paired vertical bars produce an `AbsExpr`. The result is wrapped in a `ToplevelExpr`.

```
>>> group(tokenize('g.[f, {a,b}]'))
<ToplevelExpr 'g' '.' <BracketExpr 'f' ',' <SetExpr 'a' ',' 'b'>>>
```

25.4.4 Normalization

The function `normalize()` takes the output of tokenization and converts it into a fully parsed expression. There are three parts of normalization:

- **Word normalization** categorizes each word as a `Symbol`, a string (that is, a `Seq` containing `Symbol`s), an operator (`Op`), or a variable (`Var`). A quoted string is categorized as a symbol if it contains no whitespace, and as a string containing whitespace-separated symbols otherwise. An unquoted string is categorized as a variable if it begins with underscore, as an operator if it is listed in Table 25.1, and as a symbol otherwise.
- In **expression normalization**, functions are grouped with their arguments (`ParenExpr`) to create a `Funcall` expression, and the relative precedence of operators is used to group them with their arguments.
- In **toplevel normalization**, the toplevel commands are recognized and converted to function calls. The commands are listed in Table 25.2.

```
>>> g = group(tokenize('set _x {<a>.<b>}'))
>>> print(g)
Toplevel[set _x {<a>, ., <b>}]
>>> n = normalize(g)
>>> print(n)
set(_x, {concat(<a>, <b>)})
```

25.4.5 Digesting

The function `digest()` simplifies the syntax by replacing all expressions, including complex literals, with `Funcall` objects.

```
>>> print(digest(n))
set(_x, makeset(concat(seq(a), seq(b))))
```

25.4.6 Parsing

The function `parse()` performs the sequence of steps just discussed: tokenization, grouping, normalization, and digesting.

```
>>> expr = parse('set _x {<a>.<b>}')
>>> print(expr)
set(_x, makeset(concat(seq(a), seq(b))))
```

25.4.7 Evaluation

There are four interrelated functions: `evaluate()`, `apply()`, `symeval()`, and `assign`. All take an `env` argument, which is simply a dict mapping names to values. Variables, constants, and function names are all included in `env`. They are easy to tell apart because variables begin with underscore, constants are nonalphabetic, and function names are alphabetic. The user can only change the values of variables.

Of those four functions, the only one of any complexity is `apply()`. It takes a function name and an argument list. It goes through the following steps:

- The function name is looked up in the environment to get the actual function f. An error is signalled if the name is not found, or if its value is not a function. It is also permissible for the function "name" to be an actual `Function` object, in which case no lookup is done.
- Checks are done to make sure that the argument list includes at least f`.min_narg` arguments, but not more than f`.max_narg` arguments. (The latter may have the value `Unlimited`.)
- Each argument is evaluated, unless f`.eval` exists and has the value `False` for the argument position in question.
- If f`.types` exists, the types of the arguments are checked.
- If f`.envarg` is `True`, the environment itself is added to the argument list as a new final argument.
- f`.implementation` is called on the argument list, and the result is returned.

To get an environment populated with the standard functions, call `make_env()`.

```
>>> env = make_env()
>>> print(evaluate(expr, env))
None
>>> env['_x']
<Set <Seq <Symbol a> <Symbol b>>>
>>> print(_)
{<a, b>}
```

25.4.8 Interpreter

Evaluator. An `Evaluator` instance behaves like a function with an internal environment. It can be used to evaluate a sequence of statements.

```
>>> e = Evaluator()
>>> e('set _x <a,b,c>')
>>> e('_x')
<Seq <Symbol a> <Symbol b> <Symbol c>>
```

When initialized, it uses `make_env()` to create an environment, and each time it is called it uses `parse()` to turn the string into an expression and `evaluate()` to evaluate it in the environment.

Interpreter. An `Interpreter` also evaluates statements. Unlike an Evaluator, it traps exceptions and captures the output of commands that do direct printing, like `show`. It also echoes the input statements, and if created with the setting `show_syntax=True`, it also echoes the parsed version of each input line (for debugging). The return value is a string containing all output.

```
>>> i = Interpreter()
>>> i('set _x <d,o,g>')
'| set _x <d,o,g>\n'
>>> i('_x')
'| _x\n<d, o, g>\n'
>>> i('_y')
'| _y\nERROR: Unbound variable: _y\n'
```

It can either be called with a single string (as in the examples just shown), or with an iteration over strings, such as an open file:

```
>>> with open(ex.notebook.gl) as file:
...     print(i(file))
...
| #T My Notebook
| set _x <a,b,c>
| _x . <b,a>
<a, b, c, b, a>
```

The Interpreter calls two lower-level functions:

`interpret_file` Takes three arguments: `file`, `output`, `env`. `File` may be a filename or an iterator over strings (e.g., an open file). The strings are parsed as input lines and evaluated. Processing continues even if an exception is encountered. All output is trapped and returned at the end as a string.

`parse_file` This is used by `interpret_file()` to parse the input. It takes an iterator over strings as input, and returns an iterator over triples, one for each input line. If the input line is empty or a comment, the triple is `(None, None,` *line*`)`. If there is an error during parsing, the triple is `(None,` *excep, line*`)`. Otherwise, the triple is `(`*expr,* `None,` *line*`)`.

Part VII

Constituency Parsing and Interpretation

Chapter 26

Parser: `seal.parser`

This chapter documents the module `seal.parser`. The examples assume that one has done:

```
>>> from seal.parser import *
>>> from seal.grammar import Grammar, Lexicon
>>> from seal.io import ex
```

26.1 Chart parsing

26.1.1 The algorithm

Chart parsing uses two basic data structures.

Nodes. Nodes (chart entries) are used to keep track of completed subtrees. A node represents all subtrees with common values for category, start position, and end position. For example, ${}_2\text{NP}_5$ represents all subtrees of category NP spanning positions 2 to 5 in the sentence. If the sentence is

$$_0\ \text{Max}\ _1\ \text{chases}\ _2\ \text{the}\ _3\ \text{big}\ _4\ \text{dog}\ _5$$

then ${}_2\text{NP}_5$ covers "the big dog."

Edges. Edges represent partially-parsed rewrite rules. For example, ($\text{VP} \rightarrow {}_1\text{V}_2 \bullet \text{NP PP}$) represents the parsing of the rule VP → V NP PP at a point where only the V has been found.

The parsing cycle can be illustrated by considering the parsing of the rule $X \rightarrow YZW$. Suppose that the chart contains nodes ${}_2Y_4$, ${}_4Z_7$, and ${}_7W_8$. The sequence of events is:

start:			(α)	$(X \rightarrow {}_2Y_4 \bullet Z\ W)$
combine:	$(\alpha) + {}_4Z_7$	$\Rightarrow$	(β)	$(X \rightarrow {}_2Y_4\ {}_4Z_7 \bullet W)$
combine:	$(\beta) + {}_7W_8$	$\Rightarrow$	(γ)	$(X \rightarrow {}_2Y_4\ {}_4Z_7\ {}_7W_8 \bullet)$
complete:	(γ)	$\Rightarrow$		$[{}_2X_8]$

Incidentally, the use of dotted rules accommodates unary branching as well, without further modification.

Rule	$X \rightarrow Y$
start:	$(X \rightarrow {}_2Y_4 \bullet)$
complete:	${}_2X_4$

A complete parse is illustrated in figure 26.1.

In short, there are four main operations.

- **Shift.** Takes a word at position i with pos X, and creates node ${}_iX_{i+1}$.
- **Start.** Whenever a new node ${}_iY_j$ is created, for every continuation $X \rightarrow Y \ldots$ of Y, we can create an edge $(X \rightarrow {}_iY_j \bullet \ldots)$.
- **Combine.** An edge $(\ldots_k \bullet Z \ldots)$ and node ${}_kZ_j$ can be combined to create edge $(\ldots\ {}_kZ_j \bullet \ldots)$.
- **Complete.** Whenever we have an edge $(X \rightarrow {}_i \ldots_j \bullet)$ with the dot at the end, we can create a node ${}_iX_j$.

A major issue is assuring systematicity—making sure that each necessary operation is performed, and performed only once.

- We keep a chart of nodes; this prevents us creating more than one node ${}_iX_j$.
- Shifts are not problematic: we do one shift for each part of speech of each word.
- Starts are not problematic: when we add a new node to the chart (as opposed to re-using an old node), we immediately do all the starts for that node.
- Completions are not problematic: when the dot reaches the end, we immediately create node ${}_iX_j$. If the node already exists, we add a new expansion rather than creating a new node.

The only operation remaining is combination. Here systematicity is an issue. We need to make sure all combinations get done exactly once.

A combination involves an edge $(\ldots_k \bullet Z \ldots)$ ending at k, and a node ${}_kZ_j$ beginning at k. The danger is that we combine all the edges *we know about* ending at k with nodes beginning at k, but later another edge (ending at k) gets created "behind our back," and never gets combined with nodes beginning at k.

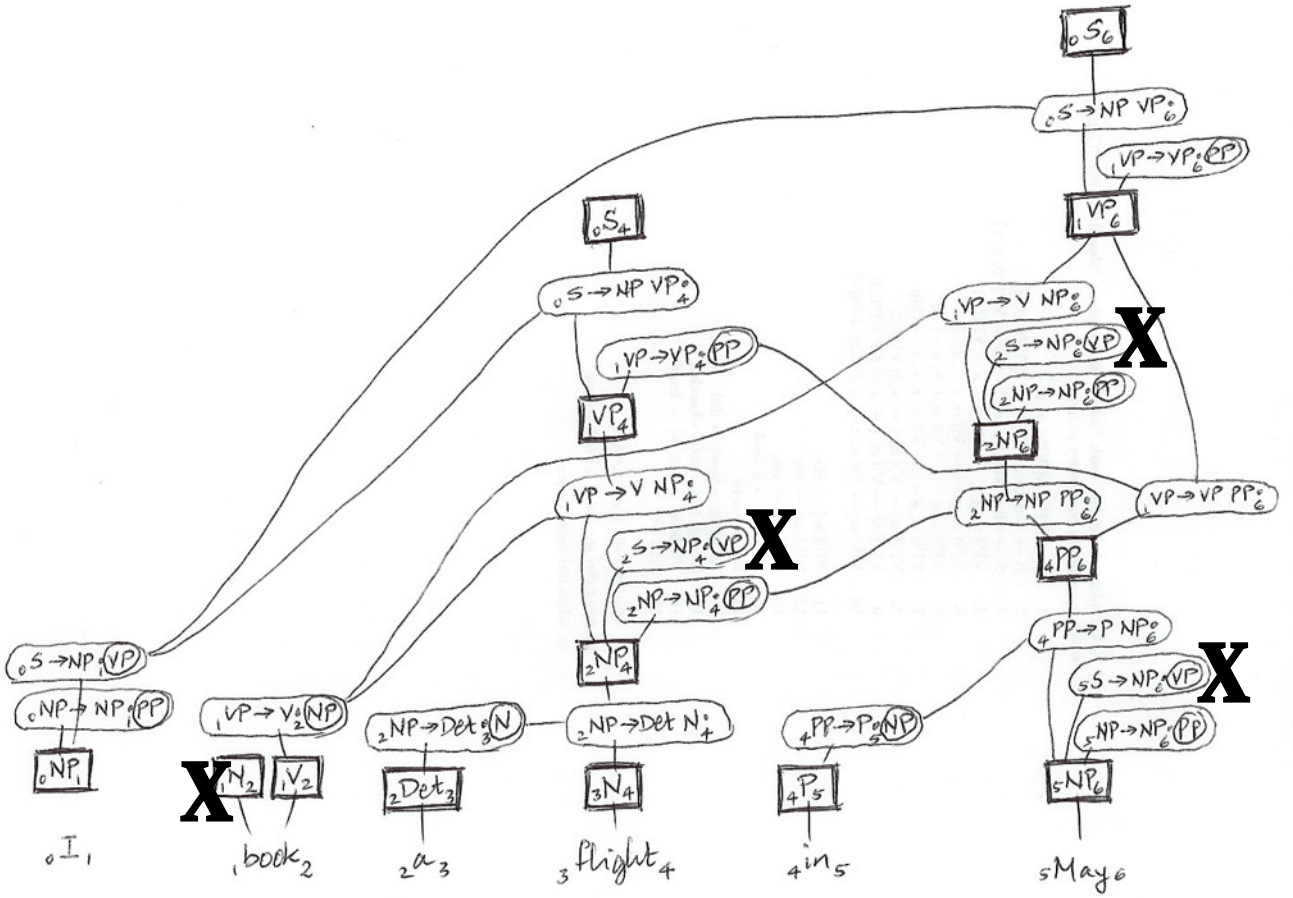


Figure 26.1: Filled chart for sentence "I book a flight in May." The node and edges marked with "X" are filtered out when topdown prediction is used. (Not shown is an initial prediction $\rightarrow \bullet_0 S$.)

Our strategy is to iterate through sentence positions j from 1 to n, and to process all nodes *ending at* j at time j. In particular, at time j, we create nodes of form ${}_iX_j$, and we create edges of form $(\ldots{}_j \bullet \ldots)$. That way, after creating node ${}_kZ_j$, we know that all edges $(\ldots{}_k \bullet Z \ldots)$ that could combine with it already exist, because the relevant edges were all created at time k, where $k < j$.

26.1.2 Node

A `Node` has four attributes: `cat`, `expansions`, `i`, and `j`. The `__init__()` method sets these four attributes:

```
>>> node = Node('n', Lexicon.Entry('dog', 'n', 'dog'), 0, 1)
>>> node
0.n.1
>>> node.cat
'n'
>>> node.expansions
[<Entry dog n : dog>]
```

Despite the name, the attribute `expansions` contains a list whose elements are not expansions per se, but either lexical entries (class `Lexicon.Entry`) or edges (class `Edge`). A lexical entry represents the "expansion" of a part of speech to a word. An edge contains both an expansion and the rule that it corresponds to. Importantly, both a lexical entry and an edge contain not only an expansion (a list of children) but also a semantic translation.

There is a method for adding an additional expansion to an existing edge:

```
>>> node.add(Lexicon.Entry('dog', 'n', 'dog2'))
>>> node.expansions
[<Entry dog n : dog>, <Entry dog n : dog2>]
```

26.1.3 Edge

An `Edge` has three attributes: `rule`, `expansion`, and `bindings`. The expansion is a list of nodes. The dot is implicit: the length of the expansion tells us how many children we have collected, which determines the position of the dot.

For example, suppose we are working on the rule `np -> det n`, and we have encountered the `det` node.

```
>>> g = Grammar(ex.sg0)
>>> g.lexicon['the']
[<Entry the Det[*] : the>]
>>> ent = _[0]
>>> node = Node(ent.pos, ent, 0, 1)
```

We will also need the relevant rule.

```
>>> g.continuations('Det')
[<NP[N] -> Det[N] N[N]>]
>>> r = _[0]
```

Recall that a rule contains a list of initial bindings.

```
>>> r.bindings
['*']
```

Now we are ready to create the edge.

```
>>> edge = Edge(None, r, [node], r.bindings)
>>> edge
(NP[X0] -> 0.Det[*].1 * N[X0] {*})
```

(The first argument is the predecessor of the edge, which we are here ignoring.)

An edge has the methods `cat()`, `sem()`, `start()`, `end()`, and `afterdot()`. The value of `cat()` is the lefthand side of the rule, and `sem()` also comes from the rule. The value of `afterdot()` is `None` when the dot is at the end.

```
>>> edge.cat()
NP[X0]
>>> edge.start()
0
>>> edge.end()
1
>>> edge.afterdot()
N[X0]
```

26.1.4 Parser

We now describe the parser in more detail. The following are more precise definitions for the four main operations. These are methods of the class `Parser`.

- **shift**(w, j). For each part of speech X of the word w, do **add node**$(X, w, j-1, j)$.
- **start**(n), where n is a node of category Y. For each rule of form $X \rightarrow Y' \ldots$, create an edge with partial expansion $[n]$, and do **add edge**. More precisely, we use $Y[0]$ to get continuations from the grammar. The category Y' in the continuation will have the same type as Y, but may differ in features. We unify the two, and if unification succeeds, the result is a binding that is carried along in the new edge.
- **combine**(n), where n is a node of form ${}_kZ_j$. For each old edge at k with Z' after the dot, where Z and Z' have the same type, unify Z and Z'. If the unification succeeds, create a new edge in which n has been added to the expansion, and do **add edge**. The bindings that result from unification are stored in the new edge.

- **complete**(e), where e is an edge of form $(X \rightarrow {}_i \ldots {}_j \bullet)$. Do **add node**$(X', \ldots, i, j)$, where X' is the result of substituting the edge bindings into the lefthand side of the edge's rule.

Two additional methods provide the glue that connects the operations:

- **add node**(X, α, i, j), where α is either a word or a list of nodes. If there is already a node (X, i, j) in the chart, add α to its list of expansions. Otherwise, create a new node, and call **start** and **combine** on it.

- **add edge**(e), where e is a newly created edge. If the dot is at the end, call **complete**. Otherwise, e is of form $(X \rightarrow \ldots {}_k \bullet Y \ldots)$; add e to the edge table at index (k, T), where T is the type of Y. Note that edges and nodes on this point: nodes are indexed by the full category, not just the type symbol. E.g., ${}_2$`NP.sg`$_5$ and ${}_2$`NP.pl`$_5$ are two separate nodes.

With the mutual calls between the main operations and the "glue" methods, it now suffices to pass through the sentence, calling `shift()` for each word in turn:

```
for j in range(1, len(words)+1):
    self.shift(words[j-1], j)
```

Simply doing `shift`(w, j) starts a cascade: shift calls `add_node()`, which calls `combine()` and `start()`. Both `combine()` and `start()` call `add_edge()`, which may call `complete()`, which may call `add_node()` again. For example:

```
>>> p = Parser(ex.sg0)
>>> p.shift('the', 1)
>>> print(p)
0.Det[*].1
(NP[X0] -> 0.Det[*].1 * N[X0] {*})
```

Printing the parser displays the nodes and edges in the order in which they were created.

```
>>> p.shift('dog', 2)
>>> print(p)
0.Det[*].1
(NP[X0] -> 0.Det[*].1 * N[X0] {*})
1.N[sg].2
0.NP[sg].2
(S -> 0.NP[sg].2 * VP[X0] {sg})
```

The nodes and edges are stored in two tables. The nodes are in `p.chart`, which is a dict. Keys are triples `(cat,i,j)`, and values are nodes. There is a unique value for any given key. The edges are in `p.edges`, which is an `Index`. Keys are pairs `(k, Z)`, where k is the sentence position corresponding to the dot, and Z is the category after the dot. Values are edges. We use an `Index` because there may be multiple values associated with a given key.

26.1.5 Unwinding

After going through the sentence, if there is a node with category S, spanning the full sentence, the parser unwinds that node and return the resulting list of trees.

The unwinding operation is a bit tricky. Consider a parent node with category X and expansion `[child1, child2]`. `Child1` and `child2` are both nodes, so we will call the list `[child1, child2]` a **node expansion**, as distinct from a **tree expansion**, which is a list of `Tree` instances. Each node may correspond to multiple trees. A tree expansion is the value for `children` for a `Tree`.

Suppose `child1.trees()` is `[t1,t2]` and `child2.trees()` is `[u1,u2]`. Each way of combining a `child1` tree t with a `child2` tree u gives us a tree-expansion $[t, u]$. Then each tree-expansion $[t, u]$ gives us a tree $[_X\ t\ u]$ corresponding to the parent node. If the parent node has multiple node-expansions, we simply pool all the trees coming from each.

The key step is computing all combinations of child1 trees `[t1,t2]` with child2 trees `[u1,u2]`. The tree-expansions are the **cross product**: `(t1,u1), (t1,u2), (t2, u1), (t2, u2)`. (Note that this remains valid even if there are more than two child-nodes involved.)

We use the `cross_product()` function from `seal.misc` to define the following, which takes a single node expansion and returns the list of all corresponding tree expansions.

```
def tree_expansions (node_expansion):
    choices = [n.trees() for n in node_expansion]
    return cross_product(*choices)
```

The function `tree_expansions()` is used, in turn, to define the `trees()` method of `Node`:

```
def trees (self):
    out = []
    for e in self.expansions:
        if isinstance(e, str):
            out.append(Tree(self.cat, word=e))
        elif len(e) == 0:
            out.append(Tree(self.cat))
        else:
            for childlist in tree_expansions(e):
               out.append(Tree(self.cat, childlist))
    return out
```

26.1.6 Toplevel call

For parsing, the parser is used as a function. (It implements the `__call__()` method.)

```
>>> p = Parser(ex.sg0)
>>> p('every dog chases a cat')
[<Tree S ...>]
>>> print(_[0])
0   (S                      : (!qs ($2 $1))
1      (NP[sg]                 : (!q $1 @ ($2 @))
2         (Det[sg] every)         : every
3         (N[sg] dog))            : dog
4      (VP[sg]                 : (lambda @ ($1 @ $2))
5         (V[sg,t,0] chases)      : chase
6         (NP[sg]                 : (!q $1 @ ($2 @))
7            (Det[sg] a)             : some
8            (N[sg] cat))))          : cat
```

26.2 Top-down filtering (Earley parser)

Top-down filtering is not implemented in the current parser, but we describe the algorithm here for reference.

A dotted rule not only keeps track of children that have been constructed, it also establishes expectations about what will come next: if Y is the category after the dot, then only nodes of category Y, or which might form the leading edge of a tree rooted in Y, could ever be used to extend the dotted rule.

For example, consider position 1 in Figure 26.1. The edges whose dot is at position 1 are in the column above the first word. There are two possible continuation categories: VP and PP. They are circled in the two edges in the first column. Now consider the two parts of speech for "book," whose start position is 1. The category V can be the first category in a VP, so it fits expectations. But the category N cannot be initial in either VP or PP, so it violates expectations.

Consider also the edge ${}_2S \rightarrow NP \bullet_4 VP$. If we subsequently find a VP and use the completed edge to construct an S, the start position of the S node will be position 2. The only edge with the dot at position 2 is the one above "book." It expects an NP, not an S; nor can S be the initial category in an NP. Accordingly, we can filter out the S edge immediately.

In short, we can use expectations to avoid creating the nodes and edges marked with an "X" in Figure 26.1. In cases where (unlike here) the nodes and edges in question spawn significant downstream construction, a lot of work can be saved by filtering them out immediately.

In the original Earley algorithm, one works top-down from expectations. For example, we expect a VP at position 1, and there is a rule VP $\rightarrow$ V NP; hence we would install an edge ${}_1$VP $\rightarrow \bullet_1$ V NP in the chart, spanning no material, but indicating that a V will also satisfy expectations.

Instead of installing these edges in the chart, an alternative is to precompute a table called the **left corner table** giving all predictions that follow from a symbol following the dot. For `g1`:

After dot	*Predicted*
S	S, NP, Det
NP	NP, Det
VP	VP, V
PP	PP, P
Det	Det
N	N
P	P

We use predictions to filter at two points:

- In `shift`, do not install a part of speech unless it is predicted
- In `bu_predict`, do not create edge ${}_iX \rightarrow Y \bullet_j \beta$ unless X is predicted.

Here is a recursive definition for "lc-predicts":

a. X lc-predicts X, for all categories X

b. if X lc-predicts Y, and there is a rule $Y \rightarrow Z\beta$, then X lc-predicts Z

Note that this definition is very similar to some of the relations involved in the conversion to CNF. The discussion here can readily be applied to those relations as well.

The relation "lc-predicts" can be thought of as a collection of pairs (X, Y) such that X lc-predicts Y. We build a table that takes pairs (X, Y) and returns true or false, depending on whether the pair is present in the table. We can use a python `set` containing pairs to implement it. Consider:

```
>>> pairs = set()
>>> ('S','NP') in pairs
False
>>> pairs.add(('S','NP'))
>>> ('S','NP') in pairs
True
```

We initialize the table using the base clause (a) above: for every category in the grammar, add pair (X, X). Then every time we add a new pair (Y, Z), clause (b) comes into play. Namely, we then look for all rules $X \rightarrow Y\beta$, and for each, we recursively add the pair (X, Y). Note that the rules $X \rightarrow Y\beta$ are the **continuations** of Y in a `cf.Grammar`. Before adding a pair (X, Y), however, we should check whether it is already present. If so, we do nothing.

In short, we define a class **LCTable** that has the following methods.

- **t = LCTable(g)**. Store the grammar as `t.grammar`, set `t.pairs` to the empty set, then cycle through the categories X of the grammar, calling `add_pair`(X, X) on each.
- **t.add_pair(Y,Z)**. First, it checks whether (Y, Z) is already present in the set of pairs. If so, it does nothing. If not, it adds the pair to the

set, then cycles through the continuations $X \rightarrow Y\beta$, recursively calling `addPair`(X, Z) for each. Note that the recursion will eventually be terminated, even if there is a cycle in the grammar: eventually, we will encounter pairs that have already been added.

- **t.predicts(X,Y)**. Takes an expected category X, and returns `True` if X lc-predicts Y, otherwise `False`. This is what we use after the table has been completed.

Finally, the class `earley.Parser` is a modification of `chart.Parser` that adds top-down filtering. It uses an LC table to implement a method `is_expected()` that determines whether a given category is expected at a given position or not, and it modifies the `shift()` and `bu_predict()` methods to test whether a node or edge is expected, before installing it in the chart.

26.3 Random generation

Random generation from a feature grammar is a little tricky. It is not currently implemented, but the algorithm is described here.

To do random generation with feature grammars, we require both a downward and upward pass, analogous to the upward chart-filling step followed by the downward unwinding step in parsing. In parsing, we "enter" an expansion from the lower left corner (the first) child, and proceed from child to child, finally "exiting" at the top. The "enter" steps involve unification of a node with a child category, and the "exit" step involves instantiation of the lefthand side category. In generation, we enter from the top, unifying the parent node with the lefthand side category.

In random generation, we fully instantiate rules as we generate a tree. Full instantiation means eliminating not only variables but also disjunctions, leaving a unique value for each attribute. The choice among disjuncts is made stochastically.

We begin by fully instantiating the start category. We create a root node for the start category, but leave the children as yet unspecified.

Then, at each point, we have a parent node with a fully instantiated category, and we find the rules that could expand it. Rules are indexed by symbol in the grammar, not by complete feature sets, so we must scan through a candidate list to determine which ones actually match the given parent category. The result of each successful match is an updated symbol table. We make a list of the matching rules, and the symbol table for each.

We make a stochastic choice among the rules that match the parent category. We use the updated symbol table to fully instantiate each righthand-side category in turn, keeping track of further symbol-table updates as we go. As we instantiate each child category, we create a node possessing the category and insert it into the parent's child array. Then we recursively expand each child in turn.

It is possible for generation to fail. For example, consider the following little grammar.

```
S -> A[f ?x] B[f ?x];
S -> A[f ?x];
A[f 1] -> foo;
A[f 2] -> bar;
B[f 2] -> baz;
```

The `f` attribute ranges over `1` and `2`; but if one choose the first expansion for `S`, then one must choose value `2`. If one happens to choose `1`, though, the problem is not detected until one attempts to generate a subtree from `B[f 1]`.

Chapter 27

Generation: `seal.gen`

This is currently broken.

27.1 Algorithm

A **node** represents the results of generating from a pair (*cat, sem*). We may consider indexing nodes: a given node may well be needed multiple times, because many choices of first child may lead to the same requirements for the next child. The calling state is recorded with the node. If additional states call for the same node, they will also be recorded as callers.

To expand the node, we find all rules whose lhs is consistent with (*cat, sem*), and for each rule we create a new state. The node is passed along as states are expanded.

A **state** is like a parser edge. It represents a partial state of generating from a given rule. The first i children have been generated. Their categories have been merged into the current bindings, and their semantics has been unified into the rule semantics.

To advance a state, we substitute the current bindings into the category of the next child, and we unify the appropriate sub-avs of the current one with the semantics of the next child. Then we generate from that (*cat, sem*) pair. The result is a list of trees. For each tree, we create a new state in which the tree category is used to update the bindings, and the tree semantics is unified into the semantics of the previous state to create the new state's semantics.

When a state has generated all children for its rule, a new tree is created, whose category comes from substituting the bindings into the rule lhs, and whose semantics is the state's semantics. That tree is passed back to the node, which passes it to all of its callers.

We keep a stack of active states. The discipline is depth-first, so that we generate a complete tree as quickly as possible.

27.2 Example

```
>>> from seal.io import ex
>>> from seal.grammar import Grammar
>>> from seal.parser import Parser
>>> from seal.gen import Generator
>>> g = Grammar(ex.tinygen.g)
>>> p = Parser(g)
>>> ts = p('fido barks')
>>> print ts[0]
0   (s              : [subj fido; type bark]
1      (np             : fido
2         (name fido))    : fido
3      (vp             : [type bark]
4         (vi barks)))    : [type bark]
>>> sem = ts[0].sem
>>> gen = Generator(g)
>>> iter = gen(sem)
>>> t = iter.next()
>>> print t
0   (s              : [subj fido; type bark]
1      (np             : fido
2         (name fido))    : fido
3      (vp             : 6.0
4         (vi barks)))    : [type bark]
```

Chapter 28

Predicate calculus: `seal.expr`

This chapter documents the module `seal.expr`. The examples assume that one has done:

```
>>> from seal.expr import *
>>> from seal.io import ex
```

The module contains the basic data structures for representing predicate calculus expressions. It is separate from `seal.interp` (which contains the semantic interpreter), because the grammar module requires `parse_expr()`.

28.1 Variables

Expressions are composed of variables (`x`, `y`) and constants (`forall`, `if`, `cat`). We implement both as strings, but to distinguish them, we provide a subclass of `str` called `Variable`, and define anything that is not an `Expr` or a `Variable` to be a constant.

```
>>> e1 = 'x'
>>> e2 = Variable('x')
>>> e1
'x'
>>> e2
x
```

28.1.1 Anonymous variables

One may create an "anonymous" variable by calling the function `fresh_variable()`:

```
>>> fresh_variable()
_1
```

There is a global count of anonymous variables, and it is incremented each time an anonymous variable is created:

```
>>> fresh_variable()
_2
```

28.1.2 Distinguishing variables and constants

When we parse a predicate calculus expression, we require some convention for distinguishing between constants and variables. How do we know that "x" should be converted to a `Variable` instance, but "Fido" should be left as a string? The Russell & Norvig text uses the convention that variables are lowercase whereas names are uppercase. Prolog and other automated reasoners use the opposite convention: variables are uppercase and names are lowercase. The convention I will adopt is that variables consist of a single letter, such as "`x`" or "`P`," whereas names contain at least two (or none). To provide some flexibility, a variable may optionally begin with underscore (which does not count as a letter), and may optionally be suffixed with any number of digits. Hence the following are recognized as variables: `_x`, `x10`, `_x10`; but not `_Sk10` (which contains two letters).

The function `is_variable_symbol()` distinguishes constants from variables:

```
>>> is_variable_symbol('hi')
False
>>> is_variable_symbol('h1')
True
```

Note that anonymous variables have names like `_1` that are not recognized as variables by `is_variable_symbol()`. This is unproblematic: anonymous variables by definition never appear in string representations of expressions.

28.2 Predicate calculus expressions

Let us begin with a couple of examples of predicate-calculus expressions:

$$\forall x[\text{cat}(x) \rightarrow \text{animal}(x)]$$

$$\exists y[\text{cat}(y) \wedge \neg\text{orange}(y)]$$

$$\lambda x[\text{cat}(x) \vee \text{dog}(x)](\text{Max})$$

We will represent them in Lisp format: Polish prefix notation with obligatory grouping parentheses. Operator symbols like "$\rightarrow$" are replaced with names ("`if`"). The previous examples are represented as:

```
(forall x (if (cat x) (animal x)))
(exists y (and (cat y) (not (orange y))))
((lambda x (or (cat x) (dog x))) Max)
```

28.2.1 Expr class

In keeping with our philosophy of simplicity, we implement expressions as tuples. An expression such as

```
(chases Fido (the cat))
```

is implemented as

```
('chases', 'Fido', ('the', 'cat'))
```

For convenience of display, however, we use the same trick that we used for `Category`: we define a class `Expr` that is a specialization of `tuple`.

```
>>> e = Expr(['chases', 'Fido', Expr(['the', 'cat'])])
>>> e
(chases Fido (the cat))
```

The words (variables and constants) in an expression usually print out without quotes, as in the example just shown. However, quotes are included if the word contains an embedded space.

```
>>> Expr(['every', 'bird dog'])
(every 'bird dog')
```

28.2.2 Parse expression

The function `parse_expr()` that takes a string and converts it to an expression instance. Here is an example:

```
>>> parse_expr('(and (dog x) (friendly x))')
(and (dog x) (friendly x))
```

Expression parsing uses `seal.io.tokenize()` for tokenization. This means that the delimiters `[]{}` are treated as stand-alone tokens, and quoted strings are recognized as single tokens.

```
>>> parse_expr('(x[y])')
(x [ y ])
>>> parse_expr('(hi"there")')
(hi there)
```

Scan expression. The function `parse_expr()` dispatches to `scan_expr()`, which scans a single expression from a token stream.

```
>>> from seal.io import tokenize
>>> toks = tokenize('Fido (+ x y)')
>>> scan_expr(toks)
'Fido'
>>> scan_expr(toks)
(+ x y)
```

28.2.3 Load expressions

The function `load_exprs()` reads a file that contains expressions.

```
>>> es = load_exprs(ex.cnf.expr)
>>> print(es[0])
(forall x
   (if (forall y
          (if (animal y)
              (loves x y)))
       (exists y
          (loves y x))))
```

28.2.4 Printing

The `__str__()` method of `Expr` produces a pretty-printed form. For example:

```
>>> e = parse_expr('(if (forall x (loves x Fido)) (loves Fido Spot))')
>>> print(e)
(if (forall x
       (loves x Fido))
    (loves Fido Spot))
```

Chapter 29

Interpretation: `seal.interp`

This chapter documents the module `seal.interp`. The examples assume that one has done:

```
>>> from seal.interp import *
>>> from seal.io import ex
>>> from seal.expr import parse_expr
>>> from seal.parser import Parser
```

29.1 Preliminaries

29.1.1 Steps in interpretation

The interpreter takes a parse tree as input and produces a predicate-calculus expression as output. There are several steps:

- Metavariable replacement. Replace the "`@`" metavariables in the semantic fragments in the parse tree.
- Quantifier raising. This transforms the parse tree, and also eliminates the "`!qs`" and "`!q`" directives in the semantic attachments.
- Translation. Fuse the semantic attachments, recursively, to produce an initial predicate-calculus expression.
- Gap replacement. Interpret the "`$g`" and "`!g=`" directives.
- Defined term replacement. Replace defined terms with their definitions.
- Standardize variables. Make sure every variable-binding operator is associated with a unique variable. This is necessary in order to avoid accidental capture of variables during lambda reduction.
- Lambda reduction. Eliminate lambda applications. To be useful for reasoning, no lambda expressions should remain after lambda reduction.

The minor operations (metavariable replacement, translation, gap replacement, variable standardization) are discussed in the remainder of this section. The more involved operations (quantifier raising, defined-term replacement, and lambda reduction) are each accorded their own section.

29.1.2 Metavariable replacement

Metavariables are used in grammars to stand in for variables. A given grammar rule may be used several times in the course of parsing, and each time it is used, a new variable is created as instantiation of the metavariable.

The symbol "`@`" represents a metavariable. The function `replace_metavariables()` replaces all occurrences of "`@`" in a given expression with a new variable.

```
>>> e = parse_expr('(lambda @ ($1 @ $2))')
>>> vp = replace_metavariables(e)
>>> vp
(lambda _1 ($1 _1 $2))
```

Note that, if we replace metavariables again, we get a different variable:

```
>>> replace_metavariables(e)
(lambda _2 ($1 _2 $2))
```

29.1.3 Fuse and translate

The `fuse()` function expands the variables "`$1`," "`$2`," etc. It is given an expression and a list of child translations.

```
>>> fuse(vp, ['chase', 'Fido'])
(lambda _1 (chase _1 Fido))
```

The function `translation()` calls `fuse()` on each node of a tree, bottom-up, to convert the tree to a predicate calculus expression.

```
>>> p = Parser(ex.sg0)
>>> p('Fido barks')
[<Tree S ...>]
>>> t = _[0]
>>> print(t)
0   (S                    : (!qs ($2 $1))
1      (NP[sg]               : $1
2         (Name Fido))          : Fido
3      (VP[sg]               : $1
4         (V[sg,i,0] barks)))   : bark
>>> translation(t)
(!qs (bark Fido))
```

29.1.4 Gap replacement

The operator `$g` is the gap metavariable, and the operator `!g=` sets its value to a regular variable. For example:

```
>>> e = parse_expr('(lambda x (!g= x (chase Max $g)))')
>>> e
(lambda x (!g= x (chase Max $g)))
>>> replace_gaps(e)
(lambda x (chase Max x))
```

29.1.5 Standardizing variables

The function `standardize_variables()` takes an expression and returns an equivalent expression in which every variable-binding operator binds a unique variable, distinct from each other and from all globally free variables. For example:

```
>>> e = parse_expr('(forall x ((lambda y (exists x (g x y z))) x))')
>>> standardize_variables(e)
(forall _3 ((lambda _4 (exists _5 (g _5 _4 z))) _3))
```

Note that lambda reduction assumes that variables have been standardized. It does not call `standardize_variables()`, but passing an expression to lambda reduction that has non-standardized variables can lead to incorrect results.

29.1.6 Symbol table

Both `standardize_variables()` and `lambda_reduction()` make use of symbol tables. The class `Symtab` is a specialization of `dict`. It differs from `dict` in two ways. (1) It returns `None` for an undefined key (instead of signalling an error). (2) If `None` is assigned to a key as value, the key is deleted from the table.

29.2 Quantifier raising

We do quantifier raising *before* converting the tree to a predicate calculus expression.

29.2.1 Motivation

This is the parse tree for the sentence "every cat chases a dog."

```
(S                    : (!qs ($2 $1))
  (NP[sg]               : (!q $1 @ ($2 @))
    (Det[sg] every)       : every
    (N[sg] cat))          : cat
  (VP[sg]               : (lambda @ ($1 @ $2))
```

```
      (V[sg,t,0] chases)   : chase
      (NP[sg]              : (!q $1 @ ($2 @))
        (Det[sg] a)          : some
        (N[sg] dog))))       : dog
```

If we convert the tree to a predicate-calculus expression before doing quantifier raising, we get:

```
(!qs ((lambda _2 (chase _2 (!q some _3 (dog _3))))
      (!q every _1 (cat _1))))
```

After lambda-reduction, we have:

```
(!qs (chase (!q every _1 (cat _1))
            (!q some _3 (dog _3))))
```

The result is known as **quasi-logical form** (QLF). It is not an interpretable predicate-calculus expression, but will become one after the quantifiers are raised to a scope position.

Quantifier raising maps from QLF to logical form (LF). The first step is to excise each quantifier, leaving its variable behind. In this case, that leaves only:

```
(chase _1 _3)
```

Then one wraps each quantifier in turn around the scope expression. (The scope expression becomes an additional argument for the quantifier.)

```
(every _1 (cat _1)
  (some _3 (dog _3)
    (chase _1 _3)))
```

(Note that we have dropped the `!q` and `!qs` operators in the process.)

One observation that will become important is this: each quantifier *must* have a distinct variable. Consider what happens if the quantifiers share the same variable:

```
(!qs (chase (!q every _1 (cat _1))
            (!q some _1 (dog _1)))))
```

After raising, we have:

```
(every _1 (cat _1)
  (some _1 (dog _1)
    (chase _1 _1)))
```

This is logically equivalent to:

```
(some _1 (dog _1) (chase _1 _1))
```

which is not at all the correct interpretation.

Assuming that no rule explicitly creates multiple quantifiers that share a variable, each quantifier in the initial translation will have a distinct variable. We need only assure that we do not create duplicates anywhere along the line.

Now a dilemma arises concerning the ordering of quantifier raising with respect to lambda reduction. In the example just given, we did lambda reduction first, but that can be problematic. Specifically, doing lambda-reduction before quantifier raising can create duplicate quantifiers. Consider the example "some dog is a friendly slobberer." The translation is:

```
(!qs ((lambda _1 (and (friendly _1) (slobberer _1)))
      (!q some _2 (dog _2))))
```

After lambda-reduction, we have:

```
(!qs (and (friendly (!q some _2 (dog _2)))
          (slobberer (!q some _2 (dog _2)))))
```

Lambda reduction has duplicated the quantifier. To avoid erroneous interpretations, we have no choice but to rename one set of variables.

```
(!qs (and (friendly (!q some _2 (dog _2)))
          (slobberer (!q some _3 (dog _3)))))
```

But now, after quantifier raising, we end up with the wrong meaning:

```
(some _2 (dog _2)
  (some _3 (dog _3)
    (and (friendly _2)
         (slobberer _3))))
```

This says that there is a friendly dog, and there is a slobbering dog, but it does not imply that they are one and the same dog.

The obvious conclusion is that we must do quantifier raising before doing lambda reduction. But a problem arises that way as well. Consider the sentence "every cat chases a dog," with translation:

```
(!qs ((lambda _2 (chases _2 (!q some _3 (dog _3))))
      (!q every _1 (cat _1))))
```

When we raise quantifiers, they come out in the wrong order.

```
(some _3 (dog _3)
    (every _1 (cat _1)
        ((lambda _2 (chases _2 _3)) _1)))
```

This is a less devastating problem: the sentence is in fact ambiguous, and the interpretation we are getting is legitimate, but it is not the preferred interpretation.

29.2.2 QR as a tree transformation

There is actually a third alternative. We can do quantifier raising on the syntactic parse tree, before translation. We again consider "every cat chases some dog." After metavariable instantiation, we have:

```
(S                    : (!qs ($2 $1))
  (NP[sg]               : (!q $1 _1 ($2 _1))
    (Det[sg] every)       : every
    (N[sg] cat))          : cat
  (VP[sg]               : (lambda _2 ($1 _2 $2))
    (V[sg,t,0] chases)    : chase
    (NP[sg]               : (!q $1 _3 ($2 _3))
      (Det[sg] a)           : some
      (N[sg] dog))))        : dog
```

The procedure for doing quantifier raising is basically the same, but we operate on the node plus semantic attachment, not just on the semantics. First, we excise the quantifier nodes, leaving behind an empty node whose translation is the variable. The result is the body:

```
(S                    : (!qs ($2 $1))
  (NP[sg])              : _1
  (VP[sg]               : (lambda _2 ($1 _2 $2))
    (V[sg,t,0] chases)    : chase
    (NP[sg])))            : _3
```

Then we wrap the quantifiers around the body. Syntactically, the body becomes an additional child node, and we add a corresponding additional "$" variable to the translation. We also drop the "`!q`" and "`!qs`" markers.

```
(NP[sg]               : ($1 _1 ($2 _1) $3)
  (Det[sg] every)       : every
  (N[sg] cat)           : cat
  (NP[sg]               : ($1 _3 ($2 _3) $3)
    (Det[sg] a)           : some
    (N[sg] dog)           : dog
    (S                    : ($2 $1)
      (NP[sg])              : _1
      (VP[sg]               : (lambda _2 ($1 _2 $2))
        (V[sg,t,0] chases)    : chase
        (NP[sg])))))          : _3
```

Only after quantifier raising do we fuse the semantic attachments. The result is:

```
(every _1 (cat _1)
  (some _3 (dog _3)
    ((lambda _2 (chase _2 _3)) _1)
```

Now lambda-reduction is safe.

Incidentally, there is an independent motivation for this approach. Scope preferences often care about the particular English word used. For example, "each" and "every" differ not in meaning, but in that "each" prefers wide scope and "every" prefers narrow scope.

29.2.3 Raise quantifiers

The function is `raise_quantifiers()`. First we create a tree to apply it to:

```
>>> p('every cat chases a dog')
[<Tree S ...>]
>>> t = _[0]
>>> tree_replace_metavariables(t)
>>> print(t)
0   (S                    : (!qs ($2 $1))
1      (NP[sg]               : (!q $1 _6 ($2 _6))
2         (Det[sg] every)       : every
3         (N[sg] cat))          : cat
4      (VP[sg]               : (lambda _8 ($1 _8 $2))
5         (V[sg,t,0] chases)    : chase
6         (NP[sg]               : (!q $1 _7 ($2 _7))
7            (Det[sg] a)           : some
8            (N[sg] dog))))        : dog
```

Now we call `raise_quantifiers()`:

```
>>> print(raise_quantifiers(t))
0   (NP[sg]               : ($1 _6 ($2 _6) $3)
1      (Det[sg] every)       : every
2      (N[sg] cat)           : cat
3      (NP[sg]               : ($1 _7 ($2 _7) $3)
4         (Det[sg] a)           : some
5         (N[sg] dog)           : dog
6         (S                    : ($2 $1)
7            (NP[sg])              : _6
8            (VP[sg]               : (lambda _8 ($1 _8 $2))
9               (V[sg,t,0] chases)    : chase
10              (NP[sg])))))          : _7
```

29.3 Defined terms

This section describes the contents of the module `seal.defs`.

```
>>> defs = Definitions(ex.sg0.defs)
>>> e = parse_expr('(every d (dog d) (some c (cat c) (chase c d)))')
>>> e
```

```
(every d (dog d) (some c (cat c) (chase c d)))
>>> defs(e)
(forall d (if (dog d) (exists c (and (cat c) (chase c d)))))
```

29.4 Beta reduction

29.4.1 Overview

Beta reduction is the process of simplifying a **lambda application**, which is the application of a **lambda expression** to arguments. (A lambda expression is one whose first element is the symbol `lambda`.) For example, suppose that we translate "chases Max" as the lambda expression

```
(lambda x (chases x Max))
```

Applying that to "Fido" gives us the lambda application

```
((lambda x (chases x Max)) Fido)
```

which simplifies, by beta reduction, to:

```
>>> e = parse_expr('((lambda x (chases x Max)) Fido)')
>>> simplify(e)
(chases Fido Max)
```

The general form of a lambda application is

(`lambda` *params body*) *args*).

In the case of our example, *params* is `[x]` (a single-element list), *body* is `(chases x Max)`, and *args* is `[Fido]` (also a single-element list).

Here are some more examples.

```
>>> e = parse_expr('''((lambda (x y) (knows (mother y) x))
...                    Fido
...                    (the cat))''')
>>> simplify(e)
(knows (mother (the cat)) Fido)
>>> e = parse_expr('''((lambda x (and (friendly x) (slobberer x)))
...                    Fido)''')
>>> simplify(e)
(and (friendly Fido) (slobberer Fido))
```

29.4.2 Definition

Definition. Beta reduction can be defined as follows:

$$(\lambda x.t)s = t[x \to s]$$

where $t[x \to s]$ means the expression t with all free occurrences of x replaced by s. The result may be another lambda amplication, in which case it is necessary to reduce again.

Substitution is defined more precisely as follows:

a. $x[x \to r] = r$
b. $y[x \to r] = y$
c. $(ts)[x \to r] = (t[x \to r])(s[x \to r])$
d. $(\lambda x.t)[x \to r] = \lambda x.t$
e. $(\lambda y.t)[x \to r] = \lambda y.t[x \to r]$

Here, x and y are (distinct) variables; r, s, and t are (possibly complex) terms.

There is one caveat: in rule (e), the variable y must not occur free in r. If it did, it would be invalidly captured by the lambda. This is true for variable-binding operators more generally: the substitution

$$\forall y.t[x \to r]$$

would also be invalid if y occurs free in r.

We can prevent this happening by first renaming all variables involved in variable-binding expressions, so that every variable-binding operator has its own unique variable.[1] Incidentally, doing so makes (d) moot.

Infinite regress. It is possible to construct pathological expressions for which lambda-reduction never returns. Consider:

$$(\lambda x.xx)(\lambda x.xx)$$

We apply the substitution $[x \to \lambda x.xx]$ to the term xx, with the result

$$(\lambda x.xx)(\lambda x.xx).$$

That is, we are right back where we started, and repeated reductions will never terminate.

The current implementation does not attempt to prevent this.

29.4.3 Implementation

Beta reduction, reduce1. The function `beta_reduction()` assumes that variables have already been standardized. It calls `reduce1()`, and if the result is a lambda application, it continues calling `reduce1()` until it obtains something that is not a lambda application. (After 100 attempts, it signals an error.)

The function `reduce1()` represents one application of lambda reduction. We combine substitution and reduction into a single pass through an expression. I.e., while applying a substitution to an expression, if the expression happens to be a lambda application, we reduce it, adding bindings to the substitution. We

[1] The standard term for this renaming is *alpha conversion.*

assume that variables have already been standardized. The combined process can be summed up as follows:

$$
\begin{array}{ll}
a. & x[x \to r|\alpha] = r \\
b. & y[\alpha] = y \qquad \text{if } y \text{ has no value in } \alpha \\
c. & ((\lambda x.t)s)[\alpha] = t[x \to s[\alpha]|\alpha] \\
d. & (ts)[\alpha] = (t[\alpha])(s[\alpha]) \\
e. & (\lambda y.t)[\alpha] = \lambda y.t[\alpha]
\end{array}
$$

Pseudocode. In detail, `lambda_reduction(expr)` is defined as follows.

- If `expr` is a bound variable, return its value. If it is a free variable, return the variable itself.
- If `expr` is a constant (i.e., not an `Expr`), return it.
- If `expr` is a lambda application, it has the form `((lambda` *params body*`)` *args*`)`. For convenience, we permit *params* to be either a Variable or a list of Variables. If it is a Variable, treat it as a singleton list.
  - Reduce each of the arguments using the current substitution.
  - Add *param* $\to$ *arg* to the substitution, for each parameter-argument pair. The value for the parameter is the argument after reduction.
  - Reduce the body using the new substitution and return the result.
- If `expr` is headed by a variable-binding operator, return a new expression consisting of operator, parameter list, and the reduced body.
- Otherwise, return a new expression consisting of the reductions of all elements `expr`.

However, if the return value is itself a lambda application, reduce it repeatedly until we obtain something that is not a lambda application.

Helper functions. There are two helper functions. The function `is_lambda_expr()` returns `True` for an expression whose first element is `'lambda'`. The function `is_lambda_application()` returns `True` for an expression whose first element is a lambda expression.

Examples. Here is an example.

```
>>> e = parse_expr('((lambda (x y) (foo (bar y) x)) (mother jack) (father jill))')
>>> simplify(e)
(foo (bar (father jill)) (mother jack))
```

Here is a somewhat trickier example.

```
>>> e = parse_expr('''((lambda (P x) (P x))
...                    (lambda y (forall z (f y z)))
...                    Fido)''')
>>> simplify(e)
(forall z (f Fido z))
```

29.5 The interpreter

The interpreter is created from a grammar file name. It creates and stores a parser for the grammar.

```
>>> interp = Interpreter(ex.sg0)
```

It behaves as a function. It takes a sentence as input, parses it, and interprets it. The return value is a list of predicate-calculus expressions, one for each parse tree.

```
>>> interp('every cat chases a dog')
[(forall _22 (if (cat _22) (exists _23 (and (dog _23) (chase _22 _23)))))]
```

One can see the results of each step of processing by providing the keyword argument `trace=True`.

Chapter 30

Automated reasoning: `seal.logic`

This chapter documents the module `seal.logic`. The examples assume that one has done:

```
>>> from seal.logic import *
>>> from seal.io import ex
>>> from seal.expr import parse_expr
>>> from seal.interp import standardize_variables
```

30.1 Clausification

Reasoning is on the basis of **clauses**, but the output of parsing and interpretation is predicate-calculus expressions. Accordingly, we must first convert a predicate calculus expression to a set of clauses.

30.1.1 Clauses

A clause is a set of literals, interpreted disjunctively. For example,

```
+(dog Fido) +(cat Fido)
```

is a clause interpreted as "either Fido is a dog or Fido is a cat." The components of a clause are **literals.** A literal is a **term** with a polarity. A term is an expression containing only variables, constants, and function application.

Both literals in the preceding example were positive. Here is an example with mixed polarities:

```
-(human Socrates) +(mortal Socrates)
```

This represents "either Socrates is not human, or Socrates is mortal," which is equivalent to "if Socrates is human, then Socrates is mortal."

The class `Literal` represents a literal. Its attributes are `polarity` and `expr`.

```
>>> lit1 = Literal(False, parse_expr('(human Socrates)'))
>>> lit1
-(human Socrates)
>>> lit1.polarity
False
>>> lit1.expr
(human Socrates)
```

One creates a clause from a list of literals.

```
>>> lit2 = Literal(True, parse_expr('(mortal Socrates)'))
>>> c = Clause([lit1, lit2])
>>> print(c)
1. -(human Socrates) +(mortal Socrates)
```

Other attributes that a clause has include `answer_literal`, `provenance`, and `weight`.

30.2 Conversion to Clauses

Let us consider an example that will illustrate the steps of conversion. This states that every animal lover is loved by someone.

```
>>> orig = load_exprs(ex.cnf.expr)[0]
>>> print(orig)
(forall x
   (if (forall y
          (if (animal y)
              (loves x y)))
       (exists y
          (loves y x))))
```

The conversion is effected by the following functions, applied in order.

30.2.1 Check syntax

The function `check_syntax()` checks that a predicate-calculus expression is well-formed. It checks that variable-binding operators have variables where expected, and that all logical operators have the right number of arguments.

```
>>> check_syntax(orig)
>>> check_syntax(parse_expr('(forall Fido (woof))'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/cl/python/seal/logic.py", line 135, in check_syntax
    raise Exception, "Expecting variable in: " + str(expr)
Exception: Expecting variable in: (forall Fido (woof))
```

30.2.2 Standardize variables

In the our current expression, there are two quantifiers that bind the variable y. After standardization, each quantifier binds a unique variable.

```
>>> e = standardize_variables(orig)
>>> print(e)
(forall _1
   (if (forall _2
          (if (animal _2)
              (loves _1 _2)))
       (exists _3
          (loves _3 _1))))
```

The function `standardize_variables()` belongs to the module `seal.interp`. It is called in the production of an expression from a parse tree, but `clausify()` calls it for the sake of expressions that are not produced by the interpreter.

30.2.3 Query expansion

The next step is the replacement of question operators `wh` and `yn` with the answer predicate `_Ans`. Our running example does not illustrate this; we give different examples. An example with the `wh` operator is:

```
>>> wh = parse_expr('(wh x (criminal x))')
```

This expands to:

```
>>> expand_query(wh)
(forall x (if (criminal x) (_Ans x)))
```

An example with the `yn` operator is:

```
>>> yn = parse_expr('(yn (criminal West))')
```

This expands to:

```
>>> print(expand_query(yn))
(and (if (criminal West)
         (_Ans yes))
     (if (not (criminal West))
         (_Ans no)))
```

30.2.4 Eliminate implications

We replace all occurrences of $P \leftrightarrow Q$ with $(P \rightarrow Q) \vee (Q \rightarrow P)$, and then we replace all occurrences of $P \rightarrow Q$ with $\neg P \vee Q$. Returning to our running example:

```
>>> e = eliminate_implications(e)
>>> print(e)
(forall _1
   (or (not (forall _2
               (or (not (animal _2))
                   (loves _1 _2))))
       (exists _3
          (loves _3 _1))))
```

30.2.5 Lower negation

An expression of form $\neg\forall$ becomes $\exists\neg$, $\neg(P \wedge Q)$ becomes $\neg P \vee \neg Q$, etc.

```
>>> e = lower_negation(e)
>>> print(e)
(forall _1
   (or (exists _2
          (and (animal _2)
               (not (loves _1 _2))))
       (exists _3
          (loves _3 _1))))
```

30.2.6 Skolemization

Skolemization is a technique for eliminating quantifiers; that is, replacing existentially-bound variables with names, leaving all remaining variables implicitly universally bound.

We begin with two observations. First, it is common in mathematics for free variables to be interpreted as universally bound. For example,

$$x + y = y + x$$

may be interpreted as

$$\forall x \forall y [x + y = y + x]$$

The second observation is that names might be interpreted as existentially bound variables. For example, consider "Fido is a dog. Fido barks. Fido does not like any cat." We might treat this as:

$$\exists\, \text{Fido}\, [\, \text{dog}(\text{Fido}) \ \wedge\ \text{barks}(\text{Fido}) \ \wedge\ \forall c\, [\text{cat}(c) \rightarrow \neg\text{likes}(\text{Fido}, c)\,]\,]$$

Note that the "name existential" *must* take wide scope over "real" quantifiers: we do not want a different Fido for each cat.

We can use these observations to eliminate (some) quantifiers. Consider "a cat chases every dog."

$$\exists c[\text{cat}(c) \wedge \forall d[\text{dog}(d) \rightarrow \text{chases}(c, d)]]$$

We can turn c into a name, and allow "$\forall d$" to be implicit.

$$\text{cat}(C) \wedge [\text{dog}(d) \rightarrow \text{chases}(C, d)]$$

Now of course there is a second reading for the sentence.

$$\forall d[\text{dog}(d) \rightarrow \exists c[\text{cat}(c) \wedge \text{chases}(c, d)]]$$

In this reading, there is a different cat for each dog. That is, the cat C is **a function of** d.

$$\text{dog}(d) \rightarrow [\text{cat}(C(d)) \wedge \text{chases}(C(d), d)]$$

This is the key idea of Skolemization.

The general rule is this: we replace each existentially bound variable y with a **Skolem function** $Y(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are the universals that have wider scope than y. Then we can delete quantifiers. All remaining variables are interpreted as universally bound.

Applied to our running example, Skolemization produces the following:

```
>>> e = skolemize(e)
>>> print(e)
(or (and (animal (_Sk1 _1))
         (not (loves _1
                (_Sk1 _1))))
    (loves (_Sk2 _1)
           _1))
```

30.2.7 Distribute disjunctions

The function `cnf()` distributes disjunctions over conjunctions, converting to conjunctive normal form. The result is represented as a list of lists. The outer list is a conjunction, and the inner lists are disjunctions.

```
>>> e = cnf(e)
>>> type(e)
<class 'list'>
>>> for d in e: print(d)
...
[(animal (_Sk1 _1)), (loves (_Sk2 _1) _1)]
[(not (loves _1 (_Sk1 _1))), (loves (_Sk2 _1) _1)]
```

30.2.8 Convert to clauses

The final step converts the list of lists to a list of clauses. In the process, disjunctions and conjunctions containing "`True`" and "`False`" are simplified if possible, as are singleton disjunctions and conjunctions. Also, the special operator `_Ans` is recognized as marking the answer literal.

```
>>> for c in clauses(e): print(c)
...
2. +(animal (_Sk1 _1)) +(loves (_Sk2 _1)
       _1)
3. -(loves _1 (_Sk1 _1)) +(loves (_Sk2 _1) _1)
```

The result is not immediately readable. Here is how to make sense of it. First, `_Sk2` is one's best/only friend: the person who loves you, if anyone does. Hence the first clause states that either your only friend loves you, or `_Sk1` is an animal. That is, if your only friend does *not* love you, then `_Sk1` is an animal. The second clause states: if your only friend does not love you, then you do not love `_Sk1`. Combining the two: if no one loves you, then there is an animal that you do not love. The counterpositive is: if you love every animal, then someone loves you.

30.2.9 Clausify

The function `clausify()` does the complete sequence of conversions from predicate-calculus expression to clause list.

```
>>> for c in clausify(orig): print(c)
...
4. +(animal (_Sk3 _4)) +(loves (_Sk4 _4) _4)
5. -(loves _4 (_Sk3 _4)) +(loves (_Sk4 _4) _4)
```

30.3 Resolution theorem proving

Let us consider some common rules of inference. The first is modus ponens, which takes the following form.

$$\frac{\begin{array}{l}\forall x[\text{human}(x) \rightarrow \text{mortal}(x)] \\ \text{human(Socrates)}\end{array}}{\text{mortal(Socrates)}}$$

The second is modus tolens.

$$\frac{\begin{array}{l}\forall x[\text{human}(x) \rightarrow \text{mortal}(x)] \\ \neg\text{mortal(Zeus)}\end{array}}{\neg\text{human(Zeus)}}$$

A third is reasoning by case.

$$\frac{\begin{array}{l}\text{murderer(Jeeves)} \vee \text{murderer(Smith)} \\ \neg\text{murderer(Jeeves)}\end{array}}{\text{murderer(Smith)}}$$

All of these rules of inference (and many others) have a common form, which becomes even more explicit if we express them in **conjunctive normal form**

(CNF). In CNF, expressions are transformed to a conjunction of disjunctions, and variables are understood as universally bound.

$$\forall x[P(x) \rightarrow Q(x)] \quad \Rightarrow \quad \neg P(x) \vee Q(x)$$

In CNF, modus ponens has the form:

$$\frac{\begin{array}{l}\neg P(x) \vee Q(x) \\ P(a)\end{array}}{Q(a)}$$

Modus tolens:

$$\frac{\begin{array}{l}\neg P(x) \vee Q(x) \\ \neg Q(a)\end{array}}{\neg P(a)}$$

Reasoning by case:

$$\frac{\begin{array}{l}P(a) \vee Q(a) \\ \neg P(a)\end{array}}{Q(a)}$$

All three are special cases of **resolution**.

$$\frac{\begin{array}{l}\pm P(\alpha) \vee Q_1 \vee \ldots \vee Q_m \\ \mp P(\beta) \vee R_1 \vee \ldots \vee R_n\end{array}}{Q'_1 \vee \ldots \vee Q'_m \vee R'_1 \vee \ldots \vee R'_n}$$

Here, α and β need not be identical, but do need to be **unifiable**. The **unifier** is the set of variable assignments that make them identical. E.g., the unifier of x and Socrates is: $x =$ Socrates. Q'_i comes from Q_i by **substituting** the unifier. E.g., substituting ($x =$ Socrates) into mortal(x) yields mortal(Socrates). The Q's and R's may be positive or negated, and the order of disjuncts is irrelevant.

Let us consider a simple example of reasoning by resolution. The knowledge base consists of two clauses:

```
1. -(human x) +(mortal x)
2. +(human Socrates)
```

Each clause is understood disjunctively. For example, clause 1 states that either x is not human, or x is mortal. (That is equivalent to: if x is human, then x is mortal.) The knowledge base asserts the conjunction of the clauses.

To answer the query "is Socrates mortal," we try to prove that Socrates is mortal. To do that, we assume that Socrates is *not* mortal, and deduce a contradiction. That is, we adopt the assumption

```
-(mortal Socrates)
```

This resolves with clause 1, with $x =$ Socrates, yielding

```
-(human Socrates)
```

This in turn contradicts clause 2. Resolving with clause 2 yields the empty clause, which represents a contradiction.

Now let us consider the query "who is mortal." Assume that no one is mortal, and try to deduce a contradiction.

```
-(mortal y)
```

This resolves with clause 1, with $x = y$, yielding

```
-(human y)
```

This resolves with clause 2, with $y =$ Socrates, yielding the empty clause.

That proves that someone is mortal, but it does not answer the question of *who* is mortal. To do so, we add an "answer literal" to our assumption:

```
-(mortal y) ; +(_Ans y)
```

This can be read as "if y is mortal, then y is the answer." Any substitutions of values for variables apply to the answer literal as to the other literals, but the answer literal is otherwise treated as an annotation rather than a contentful literal. One does not use the answer literal for resolution, and the proof is complete when only the answer literal remains.

The above assumption resolves with clause 1, yielding:

```
-(human y) ; +(_Ans y)
```

This in turn resolves with clause 2, yielding

```
; +(_Ans Socrates)
```

At this point the proof is complete: there are no content literals left. The answer is: Socrates.

Now let us consider a more complex example. The following sentences are input to the parser.

```
every American who sells a weapon to a hostile country is a criminal
West sells Nono every missile that Nono owns
every enemy of America is a hostile country
every missile is a weapon
Nono owns a missile
West is an American
Nono is an enemy of America
who is a criminal
```

The interpreter converts them to the following predicate calculus expressions. This is the contents of the file `crime.kb`.

```
(forall x7
  (if (and (American x7)
           (exists x1
             (and (weapon x1)
```

```
                    (exists x3
                      (and (and (hostile x3) (country x3))
                           (sell x7 x1 x3))))))
      (criminal x7)))

(forall x11
  (if (and (missile x11) (own Nono x11))
      (sell West x11 Nono)))

(forall x14
  (if (enemy x14 America)
      (and (hostile x14) (country x14))))

(forall x16 (if (missile x16) (weapon x16)))

(exists x17 (and (missile x17) (own Nono x17)))

(American West)

(enemy Nono America)
```

The corresponding CNF clauses are shown when we call the solver.

```
>>> from logic import solve
>>> solve('(wh x (criminal x))', 'crime.kb', trace=True)

KB
1. -(American _1) -(weapon _2) -(hostile _3) -(country _3)
   -(sell _1 _2 _3) +(criminal _1)
2. -(missile _4) -(own Nono _4) +(sell West _4 Nono)
3. -(enemy _5 America) +(hostile _5)
4. -(enemy _5 America) +(country _5)
5. -(missile _6) +(weapon _6)
6. +(missile _Sk1)
7. +(own Nono _Sk1)
8. +(American West)
9. +(enemy Nono America)

USABLE

SOS
10. [0] -(criminal _8) ; +(_Ans _8)

Resolve 10.1 + 1.6
12. [8] -(American _9) -(weapon _10) -(hostile _11) -(country _11)
        -(sell _9 _10 _11) ; +(_Ans _9)

```

```
Resolve 12.1 + 8.1
14. [6] -(weapon _12) -(hostile _13) -(country _13)
        -(sell West _12 _13) ; +(_Ans West)

Resolve 14.1 + 5.2
16. [6] -(missile _14) -(hostile _15) -(country _15)
        -(sell West _14 _15) ; +(_Ans West)

Resolve 16.1 + 6.1
18. [4] -(hostile _16) -(country _16) -(sell West _Sk1 _16)
        ; +(_Ans West)

Resolve 18.1 + 3.2
20. [4] -(enemy _17 America) -(country _17) -(sell West _Sk1 _17)
        ; +(_Ans West)

Resolve 20.1 + 9.1
22. [2] -(country Nono) -(sell West _Sk1 Nono) ; +(_Ans West)

Resolve 22.1 + 4.2
24. [2] -(enemy Nono America) -(sell West _Sk1 Nono) ; +(_Ans West)

Resolve 24.1 + 9.1
26. [1] -(sell West _Sk1 Nono) ; +(_Ans West)

Resolve 26.1 + 2.3
28. [2] -(missile _Sk1) -(own Nono _Sk1) ; +(_Ans West)

Resolve 28.1 + 6.1
30. [1] -(own Nono _Sk1) ; +(_Ans West)

Resolve 30.1 + 7.1
32. [0] ; +(_Ans West)

Resolve 32.
ANSWER 32.  ; +(_Ans West)
```

In outline, then, the prover goes through the following steps.

- **Clausification.** Convert the predicate calculus expressions to KB clauses.
- Convert the question to a clause to be *disproved.*
- The question becomes the first **active** clause ("SOS" = "set of support"). The KB clauses are the initial **usable** clauses.
- Resolve the smallest active clause C against a usable clause, where possible, yielding new clause D. (We still need to discuss **unification**.)

- Move C to the usable list. Add new clause D to the active list.
- Keep going until you reach a contradiction.

In the following sections we fill in the remaining details. We first discuss clausification, a key step of which is **Skolemization**, and then we discuss unification.

30.4 Implementation

30.4.1 KB

The class `KB` represents a knowledge base, consisting of a list of clauses. It may be loaded from a file:

```
>>> kb = KB(ex.curiosity.kb)
>>> print(kb)
6. +(animal (_Sk5 _7)) +(love (_Sk6 _7) _7)
7. -(love _7 (_Sk5 _7)) +(love (_Sk6 _7) _7)
8. -(animal _11) -(kill _10 _11) -(love _12 _10)
9. -(animal _13) +(love Jack _13)
10. +(kill Jack Tuna) +(kill Curiosity Tuna)
11. +(cat Tuna)
12. -(cat _14) +(animal _14)
```

30.4.2 Unification

Two literals are **unifiable** if they can be made identical by some choice of assignment of values to variables. The relevant choice of values for variables is called the **unifier**. Let us start with some examples:

$$a.\quad \begin{matrix}\texttt{(knows john x)}\\ \texttt{(knows john jane)}\end{matrix} \quad \Rightarrow \{\texttt{x = jane}\} \quad \text{OK}$$

$$b.\quad \begin{matrix}\texttt{(knows john x)}\\ \texttt{(knows y bill)}\end{matrix} \quad \Rightarrow \left\{\begin{matrix}\texttt{x = bill}\\ \texttt{y = john}\end{matrix}\right\} \quad \text{OK}$$

$$c.\quad \begin{matrix}\texttt{(knows john x)}\\ \texttt{(knows y (mother y))}\end{matrix} \quad \Rightarrow \left\{\begin{matrix}\texttt{x = (mother y)}\\ \texttt{y = john}\end{matrix}\right\} \quad \text{OK}$$

We confirm that the implementation behaves as intended:

```
>>> def test (e1, e2):
...     d = {}
...     try:
...         unify(parse_expr(e1), parse_expr(e2), d)
...         for key in sorted(d):
...             print(key, '->', d[key])
...     except:
```

```
...         print('Failure')
>>> test('(knows john x)', '(knows john jane)')
x -> jane
>>> test('(knows john x)', '(knows y bill)')
x -> bill
y -> john
>>> test('(knows john x)', '(knows y (mother y))')
x -> (mother y)
y -> john
```

There is one subtlety that arises. It should be possible to substitute the unifier for the variables that it binds, and leave no occurrences of those variables. The way this can fail to be true is if there is a cyclic dependency among variables. For example:

$$d.\quad \begin{array}{ll} \texttt{(knows} & \texttt{x} \quad\quad \texttt{(mother x))} \\ \texttt{(knows} & \texttt{(mother y)} \quad\quad \texttt{y)} \end{array} \Rightarrow \left\{ \begin{array}{l} \texttt{x = (mother y)} \\ \texttt{y = (mother x)} \end{array} \right\} \quad \textbf{FAIL}$$

In this case, substitution essentially never terminates; or saying it another way, substituting the unifier would create infinite literals. Unification should fail in this case. To recognize these examples, we must check whether there a variable-value chain leading from any variable x back to x again. That is known as the **occurs check.**

```
>>> test('(knows x (mother x))', '(knows y (mother y))')
Failure
```

30.4.3 Standardize apart

Unification constitutes the central step of resolution: we combine two clauses if there is a pair of literals whose polarity is opposite but whose contents are unifiable. By setting values of variables, we unification affects all literals in both clauses. We copy all remaining literals of both clauses to create a new clause, and then we do `revert()` to undo all changes.

For safety, we also change all the variables to new variables, in the newly created clause. This is called **standardizing apart.**

The function `standardize_apart()` replaces all variables in a clause with new variables. It optionally accepts a symbol table, in which the values of bound variables are used when creating the new clause.

```
>>> print(standardize_apart(kb[2]))
13. -(animal _15) -(kill _16 _15) -(love _17 _16)
```

Let us consider an example. We create clauses for "every human is mortal" and "Socrates is human":

```
>>> c1 = parse_clause('-(human x) +(mortal x)')
>>> c2 = parse_clause('+(human Socrates)')
```

Now we unify the expressions in the "mortal" literals.

```
>>> symtab = {}
>>> c1.literals[0].expr
(human x)
>>> c2.literals[0].expr
(human Socrates)
>>> unify(c1.literals[0].expr, c2.literals[0].expr, symtab)
```

The unifier is `x = Socrates`:

```
>>> symtab
{x: 'Socrates'}
```

We copy clause 1, in the context of the unifier:

```
>>> c3 = standardize_apart(c1, symtab)
>>> print(c3)
16. -(human Socrates) +(mortal Socrates)
```

That is, we have deduced that Socrates is mortal if he is human.

30.4.4 Resolultion

The function `resolve` implements resolution.

```
>>> print(resolve(c1, 0, c2, 0))
18. +(mortal Socrates) 14.1+15.1
```

`Resolve` takes four arguments: *c1, i, c2, j,* and it resolves the i-th literal of *c1* with the j-th literal of *c2.*

There is also a function `factor`, which derives new clauses from a single input clause by identifying pairs of literals that can be unified. For example, if everyone loves Harvey or else Mary loves everyone, we can conclude that Mary loves Harvey.

```
>>> c = parse_clause('+(loves x Harvey) +(loves Mary y)')
>>> out = factor(c)
>>> print(out[0])
21. +(loves Mary Harvey) 19.1+19.2
```

The combination of resolution and factoring yields a inferentially complete theorem prover.

30.4.5 Prover

The prover encapsulates a KB. It also creates a resolver internally.

```
>>> prover = Prover(ex.curiosity.kb)
```

The argument may either be a KB object or a filename that is passed to the `KB()` constructor.

The prover behaves as a function that takes a query and answers it using the KB.

```
>>> prover('(wh x (kill x Tuna))')
['Curiosity']
```

The prover accepts two keyword arguments: `trace` and `maxsteps`. By default, `maxsteps` is 200. The "curiosity" proof requires 19 steps, though the search for additional solutions continues beyond 200.

Chapter 31

Conversational agent: `seal.bot`

This chapter documents the module `seal.bot`. The examples assume that one has done:

```
>>> from seal.bot import *
>>> from seal.io import ex
>>> from seal.expr import parse_expr
```

31.1 Using the engine

31.1.1 An interaction

The previous chapters have described the various components of a conversational agent: parser, semantic interpreter, and resolution theorem prover. The `seal.bot` module encapsulates them in a conversational agent, represented by the class `NPC` (non-player character), and embeds the agent in a bare-bones environment, in the style of an interactive fiction engine.

Here is an example of an interaction, with commentary.

```
$ python -m seal.bot
NPC enter
>
```

The engine does provide an environment for multiple characters and event types, but it is a stub. There are two characters: Player and Non-Player Character (NPC). The only event type with any non-trivial implementation is "say." Anything typed at the prompt is converted to a "say" action whose agent is Player.

```
> all humans are mortal
NPC say OK
> Socrates is human
```

```
NPC say OK
> who is mortal
NPC say Socrates
> is Socrates mortal
NPC say yes
> is Socrates human
NPC say yes
> is Zeus human
NPC say I don't know
> Zeus is not mortal
NPC say OK
> is Zeus human
NPC say no
```

31.1.2 The KB and theorem prover

Several debugging commands are available to examine the internal state of the conversational agent.

```
> :kb
1. -(human _2) +(mortal _2)
2. +(human Socrates)
27. -(mortal Zeus)
```

The KB contains three clauses. "_2" is an anonymous variable; variables are interpreted as universally bound. A clause consists of literals, which have a polarity (positive or negated), and are implicitly connected by disjunction. Clauses are connected by conjunction.

For example, clause 1 states that either _2 is not human, or _2 is mortal. That is equivalent to "if x is human, then x is mortal." Clause 2 states that Socrates is human, and clause 27 states that Zeus is not mortal. All three clauses are asserted to be simultaneously true.

Here is another example (adapted from Russell & Norvig, Chapter 9), illustrating case-based reasoning of the sort that a purely backward chaining reasoner does not support.

```
> either Smith killed Tuna or Curiosity killed Tuna
NPC say OK
> any one who loves animals does not kill animals
NPC say OK
> Tuna is a cat
NPC say OK
> every cat is an animal
NPC say OK
> Smith loves animals
NPC say OK
> does Smith love Tuna
```

```
NPC say yes
> did Smith kill Tuna
NPC say I don't know
```

It would seem NPC *ought* to know. The problem is that "any one" is interpreted to mean "any *person,*" and NPC does not know that Smith is a person.

```
> Smith is a person
NPC say OK
> did Smith kill Tuna
NPC say no
> who killed Tuna
NPC say Curiosity
```

At this point, the KB contains the following clauses.

```
> :kb
1. -(human _2) +(mortal _2)
2. +(human Socrates)
27. -(mortal Zeus)
35. +(kill Smith Tuna) +(kill Curiosity Tuna)
36. -(person _18) +(animal (_Sk1 _18)) -(animal _20) -(kill _18 _20)
37. -(person _18) -(love _18 (_Sk1 _18)) -(animal _20) -(kill _18 _20)
38. +(cat Tuna)
39. -(cat _22) +(animal _22)
40. -(animal _25) +(love Smith _25)
63. +(person Smith)
```

Backward-chaining systems like Prolog permit only Horn clauses (clauses containing exactly one positive literal). Clauses 35 and 37 are not Horn clauses; the reasoning illustrated here is not supported by Prolog.

31.1.3 Parser and interpreter

Additional debugging commands allow one to examine the most important intermediate representations. The three main components are the parser, the interpreter, and the reasoner. The parser takes a sentence and converts it to a parse tree. The process can be seen using the `:chart` command.

```
> who does every cat love
NPC say I don't know
> :chart
sent= 'who does every cat love'
Add Node [0 WhPron.sg 1] who WhPron.sg : wh
Add Edge (WhNP.$0 -> [0 WhPron.sg 1] * {sg})
Add Node [0 WhNP.sg 1] (WhNP.$0 -> [0 WhPron.sg 1] * {sg})
Add Edge (WhInv -> [0 WhNP.sg 1] * Aux.$0.$1 NP.$0 VP.$1.+ {* *})
...
```

```
Add Edge (Start -> [0 Root 5] * {})
Add Node [0 Start 5] (Start -> [0 Root 5] * {})
Add Edge (VP.$0.- -> [4 V.base.t.0 5] * NP.* MP.$1 {base 0})
Add Edge (VP.$0.+ -> [4 V.base.t.0 5] * MP.$1 {base 0})
Add Edge (VP.$0.$1 -> [4 V.base.t.0 5] * NP.* SC.$2.$1 {base * 0})
```

The interpreter takes the parse tree and converts it to a predicate-calculus expression. This is accomplished in several steps, which are shown by the `:parse` command.

```
> :parse
sent= 'who does every cat love'

who does every cat love
#Tree:
  Start : $1
    Root : (wh _9 (!g= _9 $1))
      WhInv : (!qs ($4 $3))
        WhNP.sg : $1
          WhPron.sg who : wh
        Aux.sg.base does : None
        NP.sg : (!q $1 _7 ($2 _7))
          Det.sg every : every
          N2.sg : $1
            N1.sg : $1
              N.sg cat : cat
        VP.base.+ : (lambda _8 ($1 _8 !g))
          V.base.t.0 love : love
#Raise quantifiers:
  Start : $1
    Root : (wh _9 (!g= _9 $1))
      NP.sg : ($1 _7 ($2 _7) $3)
        Det.sg every : every
        N2.sg : $1
          N1.sg : $1
            N.sg cat : cat
        WhInv : ($4 $3)
          WhNP.sg : $1
            WhPron.sg who : wh
          Aux.sg.base does : None
          NP.sg : _7
          VP.base.+ : (lambda _8 ($1 _8 !g))
            V.base.t.0 love : love
#Translation:
  (wh _9 (!g= _9 (every _7 (cat _7) ((lambda _8 (love _8 !g)) _7))))
#Replace gaps:
  (wh _9 (every _7 (cat _7) ((lambda _8 (love _8 _9)) _7)))
```

```
#Definitions:
  (wh _9 (forall _7 (if (cat _7) ((lambda _8 (love _8 _9)) _7))))
#Lambda reduction:
  (wh _9 (forall _7 (if (cat _7) (love _7 _9))))
```

Finally, the reasoner converts predicate calculus expressions to clauses, before doing inference proper. The steps in the conversion can be seen by invoking the `:clause` command.

```
> :clause
expr= (wh _3 (forall _1 (if (cat _1) (love _1 _3))))
#Standardize variables:
  (wh _13 (forall _14 (if (cat _14) (love _14 _13))))
#Expand query:
  (forall _13 (if (forall _14 (if (cat _14) (love _14 _13))) (_Ans _13)))
#Eliminate implications:
  (forall _13 (or (not (forall _14 (or (not (cat _14)) (love _14 _13)))) (_Ans _13)))
#Lower negation:
  (forall _13 (or (exists _14 (and (cat _14) (not (love _14 _13)))) (_Ans _13)))
#Skolemize:
  (or (and (cat (_Sk2 _13)) (not (love (_Sk2 _13) _13))) (_Ans _13))
#Conjunctive normal form:
  [[(cat (_Sk2 _13)), (_Ans _13)], [(not (love (_Sk2 _13) _13)), (_Ans _13)]]
#Clauses:
  4. +(cat (_Sk2 _13)) ; +(_Ans _13)
  5. -(love (_Sk2 _13) _13) ; +(_Ans _13)
4. +(cat (_Sk2 _13)) ; +(_Ans _13)
5. -(love (_Sk2 _13) _13) ; +(_Ans _13)
```

31.1.4 Grammar files

The parser and interpreter are controlled by a grammar, a lexicon, and a set of defined symbols. To give a sense of the contents, I give the first few lines of each from the current default grammar, beginning with `sg2.g`:

```
Start -> Root : $1
Start -> NP.* : $1
Start -> PP.* : $1
Start -> Greeting : ($1)

# Clauses
Root -> S.-               : $1
Root -> YN                : (yn $1)
Root -> WhInv             : (wh @ (!g= @ $1))
Root -> Wh                : (wh @ (!g= @ $1))
```

Each line is a grammar rule, which consists of a syntactic portion and a semantic attachment, separated by a colon. The format is discussed in more detail below.

The first few lines of `sg2.lex` are as follows:

```
a        Det.sg : some
a        IndefArt
all      Det.pl   : every
am       Aux.1s.pred
am       Aux.1s.ing
am       Aux.1s.enp
```

The generalized quantifiers `some` and `every` are defined in terms of the basic quantifiers `forall` and `exists` in the file `sg2.defs`:

```
every x R S: (forall x (if R S))
some x R S: (exists x (and R S))
nsome x R S: (not (exists x (and R S)))
```

31.2 Agents and events

31.2.1 The event model

In our model, an **agent** is essentially a function that takes a percept and returns an action. A percept is an **event**, which is the combination of an agent and an action. An **action** is a tuple whose first element is a string representing the action type, and whose remaining elements are determined by the type. Currently, the primary action type is `'say'`; it takes a single argument, which is a string representing the utterance. Two other action types occur. The system generates an `'enter'` action when the game begins, and the user generates a `'quit'` action by hitting control-D.

The class `Event` represents an event. It is created from an agent and action:

```
>>> eng = Engine()
>>> p = eng.player
>>> e = Event(p, ('say', 'hi'))
>>> e.agent == p
True
>>> e.action
('say', 'hi')
```

31.2.2 NPC

The conversational agent is an instance of the class `NPC` ("non-player character"). It requires a grammar:

```
>>> npc = NPC(ex.sg2)
```

It creates an interpreter (which contains a parser), a KB, and a prover.

```
>>> npc.interpreter
<seal.interp.Interpreter object at 0x10320ed90>
>>> npc.kb
<seal.logic.KB object at 0x103271590>
>>> npc.prover
<seal.logic.Prover object at 0x1032715d0>
```

The `__call__()` method accepts a percept. The NPC responds only if the type is `'say'`. Otherwise it returns `None`.

```
>>> npc(e)
('say', 'hello')
>>> npc(Event(p, ('foo',)))
>>>
```

In the case of a `'say'` event, the argument of the event is the utterance. The NPC applies the interpreter to the utterance to get a list of expressions. If the sentence does not parse, the NPC responds "I don't understand."

```
>>> npc(Event(None, ('say', 'sdfsdf')))
('say', "I don't understand")
```

If there are multiple interpretations, the NPC simply takes the first. Then it calls `speech_act()` on the expression to classify it as `'ask'`, `'greet'`, or `'inform'`.

```
>>> speech_act(parse_expr('(greeting)'))
'greet'
>>> speech_act(parse_expr('(wh x (human x))'))
'ask'
>>> speech_act(parse_expr('(human Socrates)'))
'inform'
```

In response to a greeting, the NPC says "hello." In response to a question, the NPC queries its KB and speaks the answer or answers. If no answer is found, it says "I don't know." Finally, in response to an inform, the NPC adds the expression to the KB and says "OK." If anything throws an exception, the NPC traps the exception and says "Ugh, my brain hurts."

31.2.3 Player

The class `Player` is an avatar of the user. It is given access to the engine to allow the user to examine the internal state of the engine, including the internal state of the NPC, via the debugging commands described below.

```
>>> p = Player(eng)
```

The player is an agent, meaning that it has a `__call__()` method that expects a percept and returns an action. It simply prints the percept, and then prompts the user to "say" something.

```
>>> p(Event(npc, ('say', 'hello')))
NPC say hello
>
```

Whatever the user types (a single line) is wrapped in a `'say'` action and returned:

```
>>> p(Event(npc, ('say', 'hello')))
NPC say hello
> hello
('say', 'hello')
```

The user's input is "hello" (in boldface), and `('say', 'hello')` is the return value from the original call.

If the user types a line beginning with a colon, it is interpreted as a debugging command. Debugging commands produce some output, and then a new prompt is generated. However, the call to the player does not return until an utterance—a line not beginning with colon—is typed.

```
>>> p(Event(npc, ('enter',)))
NPC enter
> :help
:? - this help message
:help - this help message
:clauses - show the clauses from the prev sent
:kb - show the knowledge base
:parse - show the parse & interp of the prev sent
:reload - reload .g, .lex, .defs
:err - print the previous error
> :kb

> the dog barked
('say', 'the dog barked')
```

The debugging commands print out information about the internal state of the NPC: the parse tree and its interpretation, the mapping from expression to clauses, the KB, the identity of the error if an error was encountered.

If the user presses control-D in response to the player prompt, the player returns the action `(quit,)`.

31.3 Engine

The class `Engine` runs the simulation. It creates an NPC and player, and an initial event, in which the NPC enters. Then it enters a loop in which it alternates between agents. It calls the current agent with the current event, and the combination of current agent and the action that it returns, constitutes the next event, which is passed to the other agent. The loop continues until a `'quit'` action is encountered.

Part VIII

Web Server

Chapter 32

Web server: `seal.server`

This chapter documents the module `seal.server`. **Warning:** this module is deprecated. It has been replaced by `seal.wsgi`.

The examples assume that one has done:

```
>>> from seal.server import *
```

In the following text, classes belonging to several modules are referred to in unqualified form. Here are the module associations:

Class	Module
`BaseHTTPRequestHandler`	`BaseHTTPServer`
`FieldStorage`	`cgi`
`HTTPServer`	`BaseHTTPServer`
`StreamRequestHandler`	`SocketServer`
`TCPServer`	`SocketServer`

We begin with a discussion of the Python web server, which provides a foundation for the seal server.

32.1 The Python TCP server

The Seal web server builds on facilities provided by the Python standard library. We begin with the Python TCP server, which handles the lowlevel connection to the client (that is, to the browser).

32.1.1 Sockets

The TCP server creates a **socket**, which is an endpoint for communication. It binds the socket to a **port**, and associates it with a hostname. (The empty string can be used for localhost.) This initial socket is known as the **listening socket**.

When a client sends a TCP request to the port, the listening socket accepts the connection, and spawns a new socket, called the **connection socket**, that

represents the connection to this particular client. The listening socket then continues listening for new connections, while the connected socket processes the request from the client.

The port remains bound until the listening socket and any connected sockets are closed. An attempt to create a new socket bound to the same port will fail with an error.

32.1.2 TCP server

A `TCPServer` is created with an address and a handler class. The address is a pair (*host, port*). One can use the empty string for localhost. This becomes the initial value for the attribute `server_address`; the attribute is updated after the socket is bound. Here is an example:

```
server = TCPServer(('', 8000), TCPTestHandler)
server.serve_forever()
```

(Instead of calling `serve_forever()`, one could call `server.handle_request()` to process a single request.)

When the server's listening socket receives a connection, spawning a connected socket, the server instantiates the handler class, and the handler instance is wrapped around the connected socket. The handler class should be a specialization of `StreamRequestHandler`. In the above example, the handler class is `TCPTestHandler`.

A `StreamRequestHandler` has the following attributes:

- `request` is the connection socket.
- `client_address` is a (*host, port*) pair.
- `server` is the `TCPServer` instance. The server, in turn, has the attribute `server_address`, which is a (*host, port*) pair.
- `connection` is set equal to `request` by `StreamRequestHandler.setup()`.
- `rfile` and `wfile` get set by `StreamRequestHandler.setup()`. These are streams that read from and write to the connection socket.
- `handle()` is a no-op method that is intended to be overridden.

32.1.3 TCP test handler

The `TCPTestHandler` provides an implementation of `handle()` that prints out information about the handler, and generates a simple HTTP response. Point a browser at:

`http://localhost:8000/`

The server should generate output that looks something like this:

```
Client address: ('127.0.0.1', 51958)
Server address: ('0.0.0.0', 8000)
BEGIN REQUEST
GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive

END REQUEST
```

The test handler also sends an HTTP response (using the utility function `write_test_response()`). In the browser, you should see a web page containing the text "Hello, World!"

32.1.4 Start and stop

The server method `serve_forever()` processes TCP requests forever. If one calls it in the main thread, one must press control-C to break the loop. The utility function `start()` calls it in a new thread, so that it can be stopped again more gracefully.

```
>>> server = TCPServer(('', 8000), TCPTestHandler)
>>> start(server)
```

This is essentially the definition of the function `tcp_test()`, which creates and starts a TCP server using the TCP test handler:

```
>>> server = tcp_test()
```

One can do manually what `start()` does, as follows:

```
>>> from thread import start_new_thread
>>> start_new_thread(server.serve_forever, ())
-1341648896
```

The first argument to `start_new_thread` is a function, and the second is an argument list for it, which in this case is empty. The return value is the thread ID.

Once the server is running, we can send it a request by using a browser. Alternatively, we can issue a TCP request programmatically:

```
>>> s = GET('http://localhost:8000/')
Client address: ('127.0.0.1', 51952)
Server address: ('0.0.0.0', 8000)
BEGIN REQUEST
GET / HTTP/1.0
```

`start(s)`	Start server *s* running in a new thread.
`stop(s)`	Stop a server that was started using `start()`.
`GET(url)`	Request a URL.

Table 32.1: Generally useful functions. These can be used with any kind of server.

```
Host: localhost:8000
User-Agent: Python-urllib/1.17

END REQUEST
```

Note that the printing comes from the TCP test handler, not from `GET`. The string `s` contains the response from the test handler:

```
>>> print s,
<html><head><title>Hello</title></head>
<body>Hello, World!</body>
</html>
```

The function `GET()` is merely a convenience. One can do the same thing manually like this:

```
>>> from urllib import urlopen
>>> s = urlopen('http://localhost:8000/').read()
```

To stop the server gracefully, and free the port, Seal provides the utility function `stop()`.

```
>>> stop(server)
```

It calls the method `shutdown()` to stop the server, and it calls the method `server_close()` to cause the port to be released. It may take a few seconds for the port to be freed. After that, one can create a new server.

32.2 HTTP Server

32.2.1 Format of HTTP requests

In the above examples of the TCP test handler print-out, the "REQUEST" portions represent HTTP requests. For example:

```
GET / HTTP/1.0
Host: localhost:8000
User-Agent: Python-urllib/1.17

```

An HTTP request consists of three parts:

- The **request**, which is `GET` or `POST` followed by a **pathname** followed by an HTTP version. In our example: "`GET / HTTP/1.0`."
- The **mime headers** with various additional information. They are terminated by an empty line. In our example, there are two mime headers ("Host" and "User-Agent").
- The **data**, which begins after the empty line. The data section is empty for a `GET` request, but contains form information for a `POST` request. In our example, the data section is empty.

GET requests. As we have just seen, one can issue a GET request by visiting

`http://localhost:8000/`

The URL may contain an arbitrary pathname—the request handler may interpret it however it likes. The HTTP request contains only mime headers, no data.

POST requests. To see an example of an HTTP POST request, use `tcp_test()` to start up the TCP server, and visit the URL

`file:///cl/examples/form.html`

The form on that page looks like this:

```
<form method="POST" action="http://localhost:8000/foo/bar">
User: <input type="text" name="user" size="20" value="James & Nancy Kirk"></input><br/>
User2: <input type="text" name="user2" size="20"></input><br/>
Vote: <input type="radio" checked name="vote" value="Y">Yes</input>
      <input type="radio" name="vote" value="N">No</input><br/>
Pets: <input type="checkbox" checked name="pets" value="dog">Dog</input>
      <input type="checkbox" checked name="pets" value="cat">Cat</input>
      <input type="checkbox" name="pets" value="iguana">Iguana</input><br/>
Comments: <textarea name="comments"></textarea><br/>
<input type="submit" value="OK">
</form>
```

If you simply click "OK," the print-out from the test handler will include a request section that looks something like this:

```
BEGIN REQUEST
POST /foo/bar?hi=john%20doe HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 67

user=James+%26+Nancy+Kirk&user2=&vote=Y&pets=dog&pets=cat&comments=
END REQUEST
```

The entire form is sent as a single line of text. The format of the POST data is called "urlencoded"; it is the same as the format of the query string following the "?" in the URL of a GET request. Note that spaces in the text value for "user" get replaced with "+" characters, and %26 is the code for ampersand.

Upload requests. A special case of a POST request is a file upload. To generate an upload request, visit

file:///cl/examples/upload.html

The form on this webpage is as follows:

```
<form method="POST" enctype="multipart/form-data"
      action="http://localhost:8000/foo/bar">
File: <input type="file" name="myfile"></input><br/>
<input type="submit" value="OK"></input>
</form>
```

Click on "browse" to specify the file. A convenient choice is

/cl/examples/text1

Then click "OK." The resulting request looks like this:

```
BEGIN REQUEST
POST /foo/bar HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 F
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Type: multipart/form-data; boundary=---------------------------98494365811
Content-Length: 264

-----------------------------98494365811441089304702112 72
Content-Disposition: form-data; name="myfile"; filename="text1"
Content-Type: application/octet-stream

This is a test.
It is only a test.

-----------------------------98494365811441089304702112 72--
```

```
20
21      END REQUEST
```

32.2.2 HTTP server

The Python `HTTPServer` is almost identical to `TCPServer`. The only difference is that it looks up the server host name, and sets the attributes `server_name` and `server_port`.

The main difference is not in the server but in the request handler. The appropriate class is `BaseHTTPRequestHandler`, which builds on `StreamRequestHandler`. It reads the mime headers from `rfile` and parses them. (It knows it has reached the end when it reads an empty line.)

The parsed headers are of class `mimetools.Message`. For basic purposes, they can be treated simply as a dict. For example:

```
1       for key in headers:
2           print key, headers[key]
```

The values are strings.

The function `http_test()` is defined as follows:

```
1       def http_test ():
2           server = HTTPServer(('', 8000), HTTPTestHandler)
3           start(server)
4           return server
```

If one visits `http://localhost:8000/`, the output from the HTTP test handler looks like this:

```
1       Client address: ('127.0.0.1', 51072)
2       Server address: ('0.0.0.0', 8000)
3       Server name: skye.local
4       Mime:
5           requestline: GET / HTTP/1.1
6           command: GET
7           path: /
8           request_version: HTTP/1.1
9           Headers:
10              accept-language: 'en-us,en;q=0.5'
11              accept-encoding: 'gzip, deflate'
12              host: 'localhost:8000'
13              accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
14              user-agent: 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 Fi
15              connection: 'keep-alive'
```

The handler reads and digests the mime-headers portion of the request. Note, however, that in the case of a `POST` request, the data section of the request is left unread in `rfile`.

32.2.3 Processing the data section

Python provides the class `FieldStorage` to process the data section of `POST` requests. It also handles the query string portion of a `GET` request, to provide a uniform interface to key-value information regardless of the request method. The class `CGITestHandler` in `seal.server` gives examples of using `FieldStorage` to process `GET` and `POST` requests.

```
class CGITestHandler (BaseHTTPServer.BaseHTTPRequestHandler):

    def do_GET (self):
        (path, qs) = parse_path(self.path)
        self.form = cgi.FieldStorage(fp=None,
                                     headers=None,
                                     keep_blank_values=True,
                                     environ={'REQUEST_METHOD':'GET',
                                              'QUERY_STRING':qs})
        print_request_info(self, 'GET')

    def do_POST (self):
        ctype = self.headers['Content-Type']
        self.form = cgi.FieldStorage(fp=self.rfile,
                                     headers=self.headers,
                                     keep_blank_values=True,
                                     environ={'REQUEST_METHOD':'POST',
                                              'CONTENT_TYPE':ctype})
        print_request_info(self, 'POST')
```

The information contained in the resulting `FieldStorage` object can be accessed as follows:

```
for key in form:
    print key, repr(form.getlist(key))
```

The method `getlist()` returns a list of strings. There is also a method `getfirst()` which returns a single string.

Query string example. The function `cgi_test()` is identical to `http_test()`, except that it uses `CGITestHandler` as its request handler. Start `cgi_test()` and visit

`http://localhost:8000/foo?x=42&y=10`

The handler prints out:

```
Client address: ('127.0.0.1', 51086)
Server address: ('0.0.0.0', 8000)
Server name: skye.local
Mime:
```

```
    requestline: GET /foo?x=42&y=10 HTTP/1.1
    command: GET
    path: /foo?x=42&y=10
    request_version: HTTP/1.1
    Headers:
        accept-language: 'en-us,en;q=0.5'
        accept-encoding: 'gzip, deflate'
        host: 'localhost:8000'
        accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
        user-agent: 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 Fi
        connection: 'keep-alive'
    Form:
        y ['10']
        x ['42']
```

The "form" portion comes from the query string in the URL path.

Form example. Visit `file:///cl/examples/form.html` and click "OK." The handler prints out:

```
Client address: ('127.0.0.1', 51090)
Server address: ('0.0.0.0', 8000)
Server name: skye.local
Mime:
    requestline: POST /foo/bar?hi=john%20doe HTTP/1.1
    command: POST
    path: /foo/bar?hi=john%20doe
    request_version: HTTP/1.1
    Headers:
        content-length: '67'
        accept-language: 'en-us,en;q=0.5'
        accept-encoding: 'gzip, deflate'
        host: 'localhost:8000'
        accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
        user-agent: 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/20100101 Fi
        connection: 'keep-alive'
        content-type: 'application/x-www-form-urlencoded'
    Form:
        vote ['Y']
        user2 ['']
        user ['James & Nancy Kirk']
        pets ['dog', 'cat']
        comments ['']
```

Note that the `FieldStorage` object hides the fact that the information is coming from the form on the web page instead of from the query string at the end of the URL path. Observe also that there are multiple values for `pets`. The value

for user2 is the empty string because we specified keep_blank_values=True. If we had not specified keeping blank values, the key user2 would have been entirely absent.

Upload example. Finally, visit file:///cl/examples/form.html and browse to /cl/examples/text1. Click "OK." The handler prints out:

```
Client address: ('127.0.0.1', 51091)
Server address: ('0.0.0.0', 8000)
Server name: skye.local
Mime:
    requestline: POST /foo/bar HTTP/1.1
    command: POST
    path: /foo/bar
    request_version: HTTP/1.1
    Headers:
        content-length: '256'
        accept-language: 'en-us,en;q=0.5'
        accept-encoding: 'gzip, deflate'
        host: 'localhost:8000'
        accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
        user-agent: 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:11.0) Gecko/2
        connection: 'keep-alive'
        content-type: 'multipart/form-data; boundary=---------------------------16
    Form:
        myfile ['This is a test.\nIt is only a test.\n']
```

Observe that the contents of the uploaded file is returned as a single string.

32.3 Secure HTTP

The Secure Socket Layer (SSL) protocol runs on top of TCP. HTTP requests and responses are sent via TCP, whereas HTTPS consists simply of HTTP requests and responses sent via SSL.

32.3.1 SSL server

The function ssl.wrap_socket() wraps a TCP socket, returning an SSL socket. All writes on the SSL socket are encrypted and written as ciphertext to the TCP socket, and all reads from the SSL socket read ciphertext from the TCP socket, decrypt it, and return the plaintext.

If one wraps a listening socket, rather than a connection socket, then the result is a SSL listening socket. When a connection is accepted, it creates a TCP connection socket and automatically wraps it in an SSL connection socket.

The class `SSLServer` is a specialization of `TCPServer` that contains an SSL socket. All communication with clients is encrypted. Here is an example of creating an `SSLServer`:

```
def ssl_test ():
    server = SSLServer(('', 8003), TCPTestHandler)
    start(server)
    return server
```

Note that `ssl_test()` and `tcp_test()` are identical except for the server class. In particular, they both use the same TCP test handler. After starting `ssl_test()`, visit the url:

```
https://localhost:8003/
```

The results are also the same as for `tcp_test()`, except that among the other information printed out, one will see:

```
Cipher: ('AES256-SHA', 'TLSv1/SSLv3', 256)
```

32.3.2 Secure HTTP Server

The class `SecureHTTPServer` is a specialization of `HTTPServer`. The only modification is in the init method: the secure server wraps the socket and sets `self.socket` to the resulting SSL socket.

There is, again, a test function:

```
def https_test ():
    server = SecureHTTPServer(('', 8003), HTTPTestHandler)
    start(server)
    return server
```

Note that there is again no special handler: one uses the same HTTP test handler as in `http_test()`. After starting `https_test()`, visit the url:

```
https://localhost:8003/
```

The result is the same as for `http_test()`, except that "Cipher" is now present.

Incidentally, `SecureHTTPServer` also emulates `HTTPServer`. If it is created with the keyword argument `use_ssl=False`, it uses TCP without SSL, and listens (by default) to port 8000 instead of 8003.

32.4 The Seal web server

The Seal web server builds on the functionality examined in the previous sections.

32.4.1 Overview

The Seal web server (class `Server`) is a specialization of `SecureHTTPServer`. The handler it uses is of class `HttpConnection`, which is a specialization of `BaseHTTPRequestHandler`. `HttpConnection` supports a more abstract model of the interaction with the browser. Its behavior is controlled by a user-provided **main function** that takes a request (§32.4.4) as input and produces a response (§32.4.6) as output.

The function `testfun()` provides an example of a main function. Its definition is as follows:

```
def testfun (req):
    return StringResponse('Received: ' + str(req))
```

One instantiates the Seal server as follows:

```
>>> server = Server(testfun, logfile=None)
>>> start(server)
```

Then issue a `GET` request:

```
>>> print GET('http://localhost:8000/foo/bar?x=42&y=10')
Received: <Request RC('foo') RC('bar', {x:['42'] y:['10']})>
```

The mime type of the response is `text/plain`, and it renders as plain text in a browser.

In a little more detail, when the server receives an HTTP request, it creates a new `HttpConnection` instance to handle it. The connection instance processes the incoming HTTP request and encapsulates it as a `Request` object, which it passes to the main function. The `Request` object contains the digested path from the URL, along with any key-value pairs derived from the query string (in the case of `GET`) or the form (in the case of `POST`).

The return value from the main function is a `Response` object, which has a method `render()` that generates an HTTP response. The `seal.server` module provides one specialization, `StringResponse`, which generates an HTTP response of type `text/plain`. Additional types are defined in `seal.html`, which is documented in the next chapter.

The class `Server` has the following attributes.

- `server_address`: a pair (*host, port*). The *host* component is the IP address of the server, as a string.
- `server_name`: the hostname of the server.
- `server_port`: the second element of `server_address`.

Server	The `Server` constructor expects a main function.
Main function	A main function is given a `Request` as input and produces a response as output.
Request	A `Request` behaves as a list of request components.
Request component	A `RequestComponent` has a `filename()` (a string). The last component may also have a form, which is accessed by the methods `keys()`, `has_key()`, `getvalue()`, and `getlist()`.
Response	A response is usually a specialization of `Response`, but the only hard requirement is that it provide a `render()` method that takes a connection as input.
Connection	A connection is anything that supports the methods `send_response()`, `send_error()`, `send_header()`, `end_headers()`, and `write()`. The classes `HttpConnection` and `PseudoConnection` both qualify.

Table 32.2: A summary of the Seal server, the main function, requests, responses, and connections.

32.4.2 Invocation details

The `Server` constructor takes a few optional arguments. The keyword `logfile` is used to specify a pathname for the log file. A value of `None` specifies a pseudo-file that collects what is written to it into a string. The string can be retrieved as follows:

```
1 >>> server.logfile.getvalue()
2 'localhost - - [16/Apr/2012 21:44:27] "GET /foo/bar?x=42&y=10 HTTP/1.0" 200 -\n'
```

The default value for `logfile` is "`-`," which stands for standard output.

Specifying `use_ssl=True` causes the server to use SSL. By default, it uses TCP without SSL.

The keyword `port` can be used to specify a port to listen on. The default port is 8000 without SSL, and 8003 with SSL.

The function `run()` creates the server and starts it.

```
1 >>> run(testfun)
```

It takes the same keyword arguments as the `Server` constructor.

32.4.3 The HTTP connection

The class `HttpConnection` is a subclass of `BaseHTTPRequestHandler`. A request handler has input-side functionality and output-side functionality. On the input side, it reads and processes an incoming HTTP request. This is done

by the methods `do_GET()` and `do_POST()`. `HttpConnection` defines both methods to dispatch to the main function provided by the user. It digests the request and packages it as a `Request` object, which is passed to the main function.

On the output side, the `HttpConnection` generates an HTTP response to the client. The main function returns a `Response` object. A `Response` has a `render()` method which takes an `HttpConnection` and calls the methods that `HttpConnection` provides for generating the actual HTTP response.

An `HttpConnection` has the following attributes.

- `server`: the `Server` instance.
- `requestline`: the first line of the HTTP request.
- `command`: the first word in the request line. Usually `GET` or `POST`.
- `path`: the path component of the request line. Includes query string.
- `request_version`: the last component of the request line. Typically "`HTTP/1.1`."
- `headers`: the mime headers. The value for a given key can be accessed as `headers[key]`. One may iterate over keys with "`for key in headers`."
- `client_address`: a (*host, port*) pair for the client.
- `connection`: the connected socket.
- `rfile`: an input file that reads from the socket. In the case of a `POST`, it still contains the data section.
- `wfile`: an output file that writes to the socket.

To make it easier to generate well-formed HTTP responses, the following methods are provided. They write to `wfile`. To generate an error response, call `send_error()`. Otherwise, one should send a response, followed by some number of headers, followed by end-headers, followed by some number of writes.

- `send_response()`: takes a code and an optional message, and writes the HTTP response line. There is a large set of response codes; they can be found in the table:

```
1     >>> HttpConnection.responses
```

- `send_header()`: takes a keyword and value, and sends a mime header. After calling `send_response()`, one calls the method `send_header()` repeatedly to send mime headers.
- `end_headers()`: called when all headers have been sent. It writes an empty line on `wfile`.

- `write()`: takes a string and writes it to `wfile`. It converts the string to UTF-8 encoding if necessary. This is called repeatedly to send the actual contents of the web page.

- `send_error()`: takes a code and an optional message. This is called as an alternative to the above methods.

PseudoConnection. There is also a class `PseudoConnection` that is useful for debugging. Its constructor takes a pathname, including query string:

```
>>> c = PseudoConnection('/foo/bar?x=42')
```

The attribute `request` contains a request that is appropriate as input to the main function of a Seal server:

```
>>> c.request
<Request RC('foo') RC('bar', {x:['42']})>
```

The pseudo-connection also implements the response-generation methods that are required by the `render()` method of a `Response` object.

```
>>> r = StringResponse('Hi there')
>>> r.render(c)
HTTP/1.1 200 OK
Content-Type: text/plain

Hi there
```

One may alternatively generate an error response:

```
>>> c.send_error(404)
HTTP/1.1 404 Not Found
Content-Type: text/html
Connection: close

<html>
<head><title>Error</title></head>
<body>
<h1>Error</h1>
Not Found - Nothing matches the given URI
</body>
</html>
```

The output of these methods is sent to the connection's `wfile` attribute, which is stdout by default, but can be initialized to a file by creating the pseudo-connection with `outfn=`*filename.*

The following illustrates using the pseudo-connection with `testfun()`.

```
>>> response = testfun(c.request)
>>> response.render(c)
HTTP/1.1 200 OK
Content-Type: text/plain

Received: <Request RC('foo') RC('bar', {x:['42']})>
```

32.4.4 Requests

The main function receives a `Request` object as input. One may create a request manually using `parse_request()`, which takes pathname including query string:

```
>>> r = parse_request('/foo/bar?x=10&y=20&x=42')
>>> r
<Request RC('foo') RC('bar', {x:['10', '42'] y:['20']})>
```

The pathname portion can be obtained as a string using the method `pathname()`:

```
>>> r.pathname()
'foo/bar'
```

Otherwise, a `Request` is essentially a list of `RequestComponent`s.

```
>>> len(r)
2
>>> r[0]
RC('foo')
>>> r[1]
RC('bar', {x:['10', '42'] y:['20']})
```

Each component has a `filename()`:

```
>>> r[0].filename()
'foo'
>>> r[1].filename()
'bar'
```

The last component (only) may also have form information, which is accessed using the following methods:

```
>>> r[1].keys()
['y', 'x']
>>> r[1].has_key('z')
False
>>> r[1].getvalue('x')
['10', '42']
>>> r[1].getvalue('y')
'20'
```

Note that `getvalue()` returns a list, if there are multiple values, but the value itself, if there is only one value. There is an alternative method `getlist()` that always returns a list, even when there is only one value.

Empty components are generally suppressed: multiple slashes in sequence are treated as a single slash:

```
>>> parse_request('foo//bar')
<Request RC('foo') RC('bar')>
```

However, a trailing slash is not ignored; it causes a single empty-string component to be added:

```
>>> parse_request('foo/bar/')
<Request RC('foo') RC('bar') RC('')>
```

The motivation is that browsers do not consider "`/foo/bar`" and "`/foo/bar/`" to be equivalent. To see this, suppose the server returns the same page to both queries, and that the page contains a link to the relative pathname `./baz.html`. If the browser thinks the current URL is "`/foo/bar`," then it interprets the link as referring to "`/foo/baz.html`." But if the browser thinks the current URL is "`/foo/bar/`," then it interprets the link as referring to "`/foo/bar/baz.html`."

Adding an empty-string component when there is a trailing slash allows us to distinguish the two cases, without referring back to the original (unparsed) URL. Instead of producing the same web page in response to both URLs, the server can issue a redirect if it receives a request ending in a directory name, and return the web page in response to a request ending in an empty filename.

It is permissible to take slices of a request. Trailing slices are particularly common, since the path is often used as a dispatch hierarchy. That is, the toplevel function examines the first element in the path to determine what function to pass the request off to, and the second function may do likewise with the next element of the path, and so on. In the cascade of dispatches, one usually wants to leave the form untouched until one reaches the end; hence the utility of making the form information part of the final component.

```
>>> r[1:]
<Request RC('bar', {x:['10', '42'] y:['20']})>
```

32.4.5 Request components

The components in a request are of class `RequestComponent`. Only the last component may have a form. The methods of a component are:

- `filename()`: returns the string value of this component.
- `form()`: returns the form, if any. It is of class `FieldStorage`.
- `pathname()`: returns the pathname up to and including this component.
- `keys()`: returns the keys in the form, if there is a form.

- `has_key()`: whether or not a given key is present.
- `getvalue()`: returns the value of a key, if it has a unique value, or a list of values, if it has multiple values.
- `getlist()`: always returns a list, even if there is only one value.

32.4.6 Responses

The class `Response` has a single method, `render()`, which is intended to be overridden by subclasses. (The return value from the main function does not actually need to be a specialization of `Response`, so long as it implements the `render()` method.)

One implementation of `Response` is provided in `seal.server`: namely, `StringResponse`. Here is the complete definition, which serves as an illustrative example:

```
class StringResponse (Response):

    def __init__ (self, string):
        self.string = string

    def render (self, http):
        http.send_response(200)
        http.send_header('Content-Type', 'text/plain')
        http.end_headers()
        http.write(self.string)
        if not self.string.endswith('\n'):
            http.write('\r\n')
```

An HTTP response has a similar structure to an HTTP request: it consists of a **response line**; then some number of **mime header** lines, terminated by an empty line; then data. The `HttpConnection` provides the following methods for generating an HTTP response within a `render()` method.

In the above example, `render()` first calls `send_response()` with code 200 ("OK"). Then it sends a single header, using `send_header()`. Next is a call to `end_headers()`, and finally the data is sent by a call to `write()`.

In the case of an error, a `render()` method should not call `send_error()` directly, but should rather raise an `HttpException`.

The class `Response` also implements the `__str__()` method. It creates a `PseudoConnection` whose `wfile` writes to a string, calls its own `render()` method on that connection, and returns the resulting string. For example:

```
>>> r = StringResponse('Hi there')
>>> type(r)
<class 'seal.server.StringResponse'>
>>> print r
HTTP/1.1 200 OK
Content-Type: text/plain
```

```
7
8       Hi there
9
```

Chapter 33

WSGI and CGI

This chapter documents the module `seal.wsgi`.

33.1 Applications

33.1.1 WSGI applications

The WSGI standard defines a **WSGI application** to be a callable that takes two arguments: *environ, callback.* The value of *environ* is a dict representing environmental variables. Seal is sensitive to the following entries:

`PATH_INFO` The value is the path part of the URL, e.g., `/foo/bar`.

`REQUEST_METHOD` The value is `GET` or `POST`.

`QUERY_STRING` If the method is `GET`, this contains the portion of the URL following `?`.

`wsgi.input` If the method is `PUT`, this is an open file containing the data portion of the request.

`USER` The user name, when the server is running locally.

`REMOTE_USER` The user name, in the case of a secure connection with an authenticated user.

`SCRIPT_NAME` When the app resides within a script file, this is the script filename. All URL pathnames will begin with the script filename, but the WSGI server invokes the app with pathnames that are relative to the script filename.

The value of *callback* is a function that expects two arguments: a status string (e.g., `'200 OK'`) and a list of pairs associating HTTP header names with their values. For example:

```
[('Content-Type', 'text/plain'),
 ('Content-Length', '26')]
```

The header `Content-Length` must be present, and its value must match the total number of characters in the return value.

The return value from the application must be an iterable containing strings, providing the body of the HTTP response. The total number of characters in the strings must match the value of `Content-Length`.

33.1.2 Seal application

A **Seal application** is a callable that takes a `Request` and returns a `Response`. The class `WsgiApp` wraps a Seal application and turns it into a WSGI application. When it receives an HTML request from the WSGI server, the wrapper packages up the request as a `Request` object, and passes it to the Seal application. The Seal application returns a `Response` object, which the wrapper then delivers to the WSGI server.

The class `HtmlDirectory`, discussed in the chapter on `seal.ui`, provides an implementation of a Seal application. The user aspects of `Request` and `Response` objects are also discussed there.

In a little more detail, the `WsgiApp` receives a call with two arguments: *environ* and *callback,* in accordance with the WSGI specification. It constructs a request from the call. The request contains the following members (among others):

`pathname` The pathname of the requested item, obtained from the environment variable `PATH_INFO` (in *environ*). It is guaranteed to begin with `/`.

`user` The name of the user, from `REMOTE_USER` or `USER`, with a preference for the former, otherwise `None`.

`root_filename` The value of `SCRIPT_NAME`, otherwise `'/'`. Either it begins with a `/` followed by additional characters, or else it is the empty string.

`form` A dict representing form data obtained from `QUERY_STRING`, if `REQUEST_METHOD` is `GET`, and from `wsgi.input` if `REQUEST_METHOD` is `POST`.

The form data is digested and represented as a dict. Raw form data consists of (name, value) pairs. A name beginning with star (`*`) is deemed to be list-valued, and a name *not* beginning with star is deemed to be string-valued. The dict **key** is defined to be the name excluding the star. (Key and name are identical, if there is no star.) It is permissible to have multiple pairs with the same name, if the name begins with star, but not otherwise. In the normalized dict, keys corresponding to starred names have lists of strings as value (even

if there is only one string), and keys corresponding to unstarred names have a string as value. The normalized dict is stored as `r.form`, where `r` is the `Request`.

The Seal application is called, with the request as sole argument. The return must be of class `Response`. Subclasses of `Response` include `HtmlPage`, `RawFile`, `Text`, and `Redirect`. It is also permissible for the directory to raise an `HttpException`. (`HttpException` is in fact a subclass of `Response`.) `PageNotFound` is currently the only specialization of `HttpException`.

33.2 Providing an application to a server

33.2.1 Apache

WSGI stands for Web Server Gateway Interface. It is a Python standard, but it is implemented by Apache and other web servers. On our local web server, an application "foo" is expected to reside in the **script file**

`~/public_html/cgi-bin/foo.wsgi`

The script file must define a callable called `application`. Here is an example:

```
1  import site
2  site.addsitedir('/home/clling/cl/lib/python2.6/site-packages')
3
4  from seal.examples.ui import TestDirectory
5  from seal.wsgi import WsgiApp
6
7  application = WsgiApp(TestDirectory())
```

33.2.2 Test server

One can also run a WSGI test server on the local host. It will not serve requests from other machines, but it can be accessed by pointing a web browser at `http://localhost:8000/`. To start the server, call `run()`. It takes a Seal application as input:

```
1  >>> from seal.examples.ui import TestDirectory
2  >>> from seal.wsgi import run
3  >>> run(TestDirectory())
```

Then visit `http://localhost:8000/`.

The server runs in the main thread. To stop it, type control-C.

33.2.3 Calling an application in python

For programmatic testing, the function `call()` will run the server in a subordinate thread, and send it a pathname as if one had typed it into a web browser. The return value is the contents of the resulting web page. The following example also illustrates implementation of a minimal Seal application.

```
>>> from seal.wsgi import Text, call
>>> def hello (req):
...     return Text('Hello, world!')
...
>>> call(hello)
'Hello, world!'
```

The function `call()` generally takes two arguments, the second being a string representing the pathname portion of a URL. When omitted, as here, it defaults to `''` (the root).

Although it might seem that `call()` does nothing but call the function and print out the result, in fact it launches a WSGI server in a separate thread, and passes a URL to it just as if one had typed it into a web browser. It then processes the HTTP response returned by the server.

This example illustrates that a Seal application is simply a callable that takes a `Request` and returns a `Response`. The class `Text` is a subclass of `Response` that wraps a string or list of strings. The strings are interpreted as plaintext, not HTML.

33.2.4 Calling from CGI

Chapter 34

Persistent objects: `seal.db`

This chapter documents the module `seal.db`. The examples assume that one has done:

```
>>> from seal.db import *
>>> from seal.io import ex
```

The `seal.db` module provides functionality for maintaining a flat-file database of persistent objects.

34.1 Examples

34.1.1 Creating tables and records

A table is a collection of **records**. The table can be thought of as a data matrix whose rows are records and whose columns are **fields**. The table is (usually) associated with either one or two files. If there are two files, one has the suffix `.tab` and contains the tabular contents, and the other has the suffix `.hdr` and contains field names and other information constituting the table **schema**. If there is only one file, it is a simple tabular file, and the field names must be supplied by the caller, or else default to "F1," "F2," etc.

To load a simple tabular file (with no accompanying header file), one calls the `DataTable` constructor with the `filename` argument:

```
>>> tab1 = DataTable(filename=ex.tab1.tab)
>>> print(tab1)
foo      42
bar      15
```

In this case, the field names are automatically provided:

```
>>> print(tab1.schema())
[0] F1 w=8 sz=20
[1] F2 w=8 sz=20
```

One can provide more descriptive field names using the `schema` argument:

```
>>> tab1 = DataTable(filename=ex.tab1.tab, schema=['name', 'age'])
>>> print(tab1.schema())
[0] name w=8 sz=20
[1] age w=8 sz=20
```

Schemas are discussed in more detail below.

A table with a persistent schema—that is, a table stored in paired `.tab` and `.hdr` files—must be created using the `create_table()` function. One provides a filename and a schema. The schema is a list whose elements are either field names (strings) or `DataField` objects.

```
>>> schema = [DataField('id', id=True), 'name', 'age']
>>> create_table('/tmp/foo', schema)
```

This creates the files `/tmp/foo.hdr` and `/tmp/foo.tab`. If the `.hdr` file already exists, it is overwritten. If the `.tab` file already exists, it is left alone.

To load the table, use the `DataTable` constructor with the `basename` argument (which is the first argument):

```
>>> table = DataTable('/tmp/foo')
```

The `DataTable` constructor takes one more argument: `rectype`. This allows one to define specialized record types; the value should be either `Record` or a specialization of `Record`.

The length of the table is the number if items it contains.

```
>>> len(table)
0
```

One creates a new item using the method `new()`.

```
>>> item = table.new()
>>> type(item)
<class 'seal.db.Record'>
>>> len(table)
1
```

34.1.2 Accessing and setting values

When an item is created, values are set for its fields:

```
>>> item['id']
'1'
>>> item['name']
''
>>> item['age']
''
```

The value for `'id'` is automatically generated, because it was declared "`id=True`." The other values default to the empty string.

One can modify the table by changing the values in an item. Values must always be strings.

```
>>> item['name'] = 'Alice'
>>> item['age'] = '20'
>>> print(table)
1        Alice    20
```

The table is automatically saved each time an item is created, deleted, or modified.

```
>>> del table
>>> table = DataTable('/tmp/foo')
>>> print(table)
1        Alice    20
```

One can pass initial property values to `new()`. Use `None` for ID fields.

```
>>> item = table.new(None, 'Bob', '6')
>>> print(item)
id   2
name Bob
age  6
```

One can also specify the properties in keyword format.

```
>>> item = table.new(age='80', name='Carl')
>>> print(item)
id   3
name Carl
age  80
```

34.1.3 Accessing items

Items can be accessed by ID. (If there is more than one ID field, the first one is used.)

```
>>> item = table['1']
>>> print(item)
id   1
name Alice
age  20
```

One can also iterate over the items in the table.

```
>>> for item in table: print(item['id'], item['name'])
...
1 Alice
2 Bob
3 Carl
```

There are also methods for fetching items according to their values for attributes: see the section on Indexing and Searching below.

34.1.4 Other information

One can test for the presence of an ID:

```
>>> '2' in table
True
```

One can get the list of IDs:

```
>>> sorted(table.keys())
['1', '2', '3']
```

(The first one is a Unicode string because it was read from file. The other two were subsequently created.)

One can get the list of field names:

```
>>> [f.name for f in table.schema()]
['id', 'name', 'age']
```

Or one can test whether something is a field name:

```
>>> table.has_field('age')
True
```

34.1.5 Deletion

Items may be deleted by ID:

```
>>> del table['3']
>>> print(table)
1        Alice    20
2        Bob      6
```

The table will not re-use the ID of the deleted item.

```
>>> print(table.new(None, 'Diane', '45'))
id   4
name Diane
age  45
```

An item may also be deleted by calling its `delete()` method.

```
>>> item = table['1']
>>> item.delete()
>>> print(table)
2          Bob        6
4          Diane      45
```

The length of the table also changes when an item is deleted:

```
>>> len(table)
2
```

An item has no values after it is deleted. One can test whether an item has been deleted using the method `is_deleted()`.

One can delete an entire table by calling the table's `delete()` method.

```
>>> table.delete()
```

34.2 Refinements

For illustration, let us create a table that exercises more of the facilities that are available.

```
>>> fs = [DataField('id', width=4, id=True),
...       DataField('name', immutable=True),
...       DataField('boss', indexed=True)]
>>> create_table('/tmp/foo', fs)
>>> table = DataTable('/tmp/foo')
>>> rec1 = table.new(None, 'Abby', '')
>>> rec2 = table.new(None, 'Beth', '1')
>>> rec3 = table.new(None, 'Charley', '1')
>>> rec4 = table.new(None, 'David', '2')
```

34.2.1 Indexing

If a field is specified as being indexed, it maintains an internal index. Given a value, the index returns the list of objects that have that value in the given field. A field that is specified as an ID field is automatically indexed.

In the data table just created, the field `'boss'` is indexed. One accesses the index using the method `where()`.

```
>>> table.where('boss', '1')
[<Record 2>, <Record 3>]
>>> table.where('boss', '2')
[<Record 4>]
```

These are the items that have the value `'1'` for `boss`. One can find out what values are present using the method `values()`.

```
>>> table.values('boss')
['1', '2']
```

New items are added when created:

```
>>> i5 = table.new(None, 'Erin', '2')
>>> table.where('boss', '2')
[<Record 4>, <Record 5>]
```

Changing a value may cause the item to be reindexed:

```
>>> i5['boss'] = '3'
>>> table.where('boss', '2')
[<Record 4>]
>>> table.where('boss', '3')
[<Record 5>]
```

Deleting an item removes it from the index.

```
>>> del table['2']
>>> table.where('boss', '1')
[<Record 3>]
```

However, the deleted item's ID may still persist in fields of other items.

```
>>> print(table['4'])
id   4
name David
boss 2
>>> table.where('boss', '2')
[<Record 4>]
```

As far as the table is concerned, the value for the "boss" attribute is an arbitrary string; we are the ones who interpret it as an item ID. If we wish to eliminate these "dangling references," we must do it manually:

```
>>> for item in table.where('boss', '2'):
...     item['boss'] = ''
...
>>> table.where('boss', '2')
[]
```

34.2.2 Searching

One can search for items by description, using the method `items()`.

```
>>> table.items(name='Charley')
[<Record 3>]
>>> table.items(boss=['1', '3'])
[<Record 3>, <Record 5>]
```

The method `items()` will use indices if it can. To be precise, it constructs a candidate list, which it then filters using the function `matches()` from `seal.misc`. If `id` is one of the attributes in the description, then the candidate list contains only one item: the item with the given ID. Alternatively, if any of the attributes in the description are indexed, the candidate list consists of the items with the given value, taken from the index. If more than one attribute is indexed, the shortest candidate list is used. Only as a last resort, if none of the attributes in

the description are `id` or indexed attributes, does the candidate list consist of all items in the table.

The method `item()` returns a single item instead of a list. It signals an error if the description does not specify a unique, existing item.

```
>>> table.item(name='Charley')
<Record 3>
```

To determine all the attested values for a given property, one can use `values()`. For indexed fields, it simply returns the value list in the index. For non-indexed fields, it passes through the table and constructs a list of unique values.

```
>>> table.values('boss')
['1', '3']
```

34.3 Classes

34.3.1 Record

An item belongs to the class `Record`. A `Record` behaves like a dict, mapping field names to values; the values are effectively the contents of cells in the data table. Setting a value changes both the record in memory and on disk.

The following is a summary of the attributes and methods of the class `Record`.

- `rec.table`: The table that the record belongs to.
- `rec(key)`: Returns a value, given a field name.
- `rec(key) = val`: Changes the value, in memory and on disk.
- `key in rec`: Whether a given string is a field name.
- `rec.keys()`: Returns the list of field names.
- `rec.is_immutable(key)`: Whether the given field is immutable.
- `rec.values()`: Returns a tuple containing the value for each field.
- `rec.setvalues(vals)`: takes a list of values, and changes the values of all fields at once. Changes the values both in memory and on disk.
- `rec.delete()`: Delete the record. It is deleted from the table both in memory and on disk. After deletion, the record still contains its values, but `rec.table` is `None`, so most methods other than `rec.values()` will fail.
- `rec.iteritems()`: Returns field, value pairs.

```
>>> for field, value in rec1.iteritems():
...     print(field, field.immutable, repr(value))
...
id True '1'
name True 'Abby'
boss False ''
```

- `rec.id()`: Returns the value for the ID field, if the table has an ID field.
- `rec.matches(desc)`: Whether or not the record matches the given description.

34.3.2 Data field

A schema consists of a list of fields. The constructor is `DataField`, though the class name is actually `_Field`; it is a specialization of `str`.

The following is the complete list of arguments that the `DataField` constructor takes.

- `name` (string): the name for the field (a string).
- `immutable` (boolean): if `True`, the value of the field cannot be changed once set. (The empty string counts as not set.) Default: `False`.
- `indexed` (boolean): whether to create an Index that allows quick access to records by their value for this field. Default: `False`.
- `unique` (boolean): if `True`, then there may be only one record with any given value for this field. Automatically sets `indexed=True`. Default: `False`.
- `width` (int): controls the printing when one does "`print(table)`." The value for this field is truncated at `width` characters. A width of -1 means infinity. A width of 0 means that the field is not included in the summary at all. Default: 8.
- `size` (int[,int]): the size of the text box or text area to use when displaying this field in an HTML form. A single int means to use a text box. Two ints are rows and columns for a text area. Default: 20.
- `values` (comma-separated strings): A list of valid values for this field. An error is signalled if one attempts to set the value to something else. Causes a dropdown list to be used in an HTML form. Default: unconstrained.
- `id` (boolean): whether this field is an ID field or not. If `True`, then values are automatically generated (though it is possible to choose a specific value for a given record, if desired). Specifying `id=True` automatically sets `immutable=True`, `indexed=True`, `unique=True`.

One can obtain a schema from a data table; the schema behaves like a list of fields.

```
>>> schema = table.schema()
>>> f = schema[0]
>>> f.name
'id'
>>> f.immutable
True
```

Data field objects have the following attributes:

- `f.name`: a string.
- `f.i`: the position of the field in the list of values, from 0.
- `f.immutable`: whether the field is immutable.
- `f.index`: an `Index` mapping values to lists of records, or `None`.
- `f.unique`: whether or not values must be unique (at most one record with any given value).
- `f.width`: the column width when printing out the table.
- `f.size`: the size of the text box or text area when constructing an HTML form.
- `f.values`: the valid values for the field. `None` or `''` mean unrestricted.
- `f.maxid`: the maximum numeric value in this field for any record that has yet been created.

Incidentally, a field object is a specialization of a string. The tuple returned by the method `schema()` consists of field objects.

34.3.3 Schema

The function `as_schema()` accepts a list of strings or data fields (possibly a mixture of both), and returns a `Schema` object. A string is converted to a data field by doing `DataField(s)`. The function `as_schema()` also accepts a `Schema`, returning it unchanged.

Alternatively, a schema can be loaded from a file using the `Schema` constructor:

```
>>> foo = Schema('/foo/bar.hdr')
```

A `Schema` object has the following attributes and methods:

- `schema.fields`: the list of fields.
- `schema.filename`: the filename this schema is associated with, if any.

- `schema.id_field`: the ID field in the schema. If there is more than one, this is the first one. If there are none, this is `None`.
- `len(schema)`: the number of fields.
- `schema[i]`: the i-th field.
- `for f in schema`: iterating over fields.
- `f.save(fn)`: save the schema to a file. `Fn` is optional. If provided, it becomes the new value of `schema.filename`. If not provided, `schema.filename` must already be set.
- `print(schema)`: prints out information about each field.

34.3.4 Table

A table behaves like a dict in which the keys are IDs (that is, ID field values). If the table has no ID field, then the position of the record in the table (numbering from 0) is used as ID. When positional IDs are used, one may use either ints (e.g., `6`) or strings (e.g., `"6"`) interchangeably.

A caution is in order: deleting records can leave gaps. The position of a deleted record is *not* a valid ID. Deleting a record does not cause the position of any other record to change, but if one reloads the table from disk, the gaps disappear, and the positions of records may change. That is, positional IDs are valid only for a particular instance of the table in memory, not across in-memory instances of the table.

A `DataTable` object has the following attributes and methods.

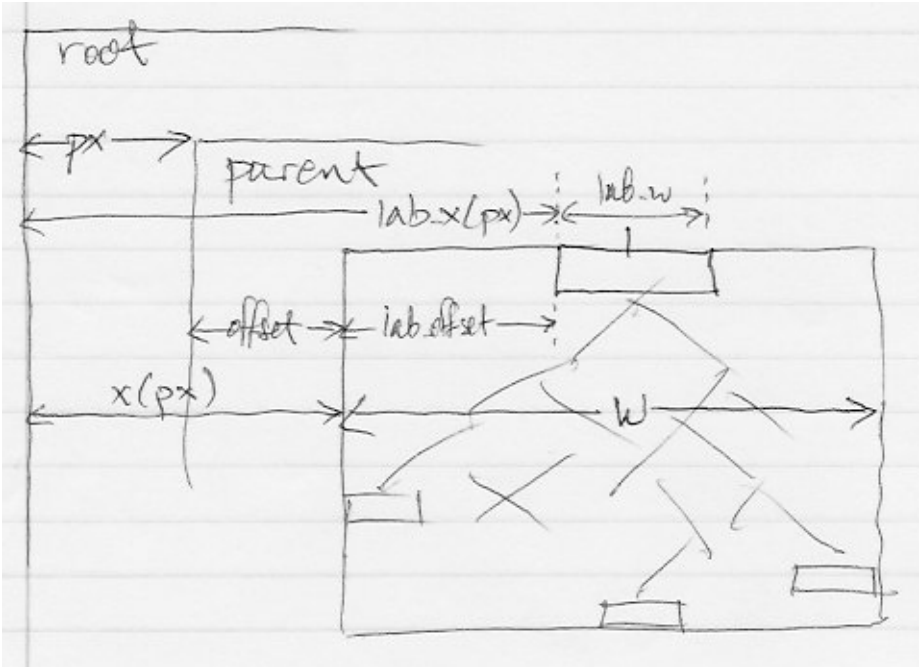
- `t.schema()`: returns the schema.
- `t.field(key)`: convenience function that gets a field by name, from the schema.
- `t.field_index(key)`: convenience function that returns the position (column number, from 0) of a named field.
- `t.has_field(key)`: whether or not the schema contains a field named `key`.
- `t.id_field()`: the ID field object, or `None`.
- `t.indexed_fields()`: the list of indexed fields.
- `t.is_immutable(key)`: whether the named field is immutable.
- `t.filename()`: the filename of the data file (not the schema file). Conventionally has the suffix `.tab`.
- `len(t)`: the number of records in the table. Deleted records are not included in the count.

- `for rec in t`: iterates over `Record` objects.
- `id in t`: whether the given ID is present in the table. IDs for deleted records do not count as present. Valid IDs are never less than 0, but they may be `len(t)` or greater, because of gaps left by deleted records.
- `t[id]`: returns the record with the given ID.
- `t.keys()`: returns all IDs. If the table uses positional IDs, the return value is a list of ints.
- `print(t)`: prints all records in the table.
- `t.where(key,val)`: returns a list of records that have the given value for the given field.
- `t.values(key)`: returns the list of distinct values for the given field.
- `t.items(...)`: returns the list of records matching the given description.
- `t.item(...)`: returns the single record matching the given description. Signals an error if the matching record does not exist or is not unique.
- `t.new(...)`: creates a new record. One can optionally provide initial values, giving them either as positional arguments, or as keyword arguments.
- `del t[id]`: delete the record with the given ID.
- `t.delete()`: delete the entire table, on disk. Also deletes the schema on disk. Does nothing if there is no filename. Does not change the table in memory. Note that if one subsequently modifies any records, the table will automatically save itself, and the files will reappear.

Chapter 35

Javascript

35.1 Tree drawing



A tree does not record its absolute position, but its position relative to its parent. The rigid information we have is:

- `w`: width of bounding box
- `offset`: left edge of bounding box relative to parent
- `labOffset`: left edge of label relative to left edge of bounding box

- `labW`: width of label

From that we can compute:

- `x = px + offset`

We do not set `x` permanently, because we may need to move the tree repeatedly, and each move would require a walk over all the nodes of the tree. Instead, we set `x` temporarily, in the context of a particular recursive descent of the tree. When `x` is set, we can compute:

- `r = x + w`
- `labX = x + labOffset`
- `labR = labX + labW`
- `labCtr = labX + labW/2`

When we first create a tree, the `offset` is set to 0. The `offset` is set to something meaningful when the tree becomes a child in a larger tree.

To assemble a tree from a given `label` and a list of `children`, we must set the `offset` of each child, and we must compute `labOffset` and `w` for the new (parent) tree. Assume that `spacer` is also given: this is the minimum space we want between a pair of adjacent labels.

The first child's offset is 0. Iterate through the remaining children, and position them. Then `w` is `r` of the last child. The desired center position is `w`/2, so:

```
labOffset = w/2 - labW/2
```

To position a `child`, let `prev` be the preceding child. We do a recursive walk of the right edge of `prev` and the left edge of `child`. At each step in the walk, we are given the current descendant of `prev` (`pnode`), the current descendant of `child` (`cnode`), and the current estimate of `offset`. Both `pnode` and `cnode` have their `x` values temporarily set. (The setting of `cnode.x` assumes that `offset` is correct.)

The minimum x position for `cnode`'s label is

```
minleft = pnode.labR + spacer
```

If `cnode.labL` is less than `minleft`, then add the difference to `offset` and also to `cnode.x`. Then recurse.

When the recursion bottoms out, we will have the needed `offset` for `child`.

After positioning all children, recall that we will need `r` for the last child. That can be computed by setting its `x` value to its `offset`. The last child's parent is the node we are creating, so we can safely define `px` = 0.

Chapter 36

Browser as user interface: `seal.ui`

This chapter documents the modules `seal.ui` and `seal.html`. The examples assume that one has done:

```
>>> from seal.ui import *
```

36.1 Overview

36.1.1 Creating a web page

One creates a web page by instantiating `HtmlPage`. A commonly-used optional parameter is `title`:

```
>>> p = HtmlPage(title='Test Page')
```

One then adds UI components to the page. For example:

```
>>> p.add(P('This is a ', B('test'), '.'))
```

Printing the page shows what will be sent to the client:

```
>>> print(p)
HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Content-Length: 286

<html>
<head>
<title>Test Page</title>
<link rel="stylesheet" type="text/css" href="/.lib/default.css" />
<script src="/.lib/default.js" type="text/javascript"></script>
```

```
</head>
<body>
<p>This is a <b>test</b>.</p>
<script type="text/javascript">sealSetup();</script>
</body>
</html>

```

36.1.2 Html directories

An `HtmlDirectory` is, conceptually, nothing more than a map from filenames to web pages. The web pages are generated on demand, so they are represented not as instances but as methods. Hence defining a directory consists in defining a subclass of `HtmlDirectory`. The mapping from filename to method is placed in the distinguished static member `pages`.

Here is an example, from `seal.examples.ui`:

```
class TestDirectory (HtmlDirectory):

    pages = {'': 'home_page',
             'foo': 'foo',
             'bar': 'recurse'}

    def __init__ (self, depth=0):
        HtmlDirectory.__init__(self)
        self.depth = depth

    def home_page (self):
        page = HtmlPage(title='Hello')
        page.add(P('Depth %d' % self.depth))
        page.add(UL(Link('foo 42', 'foo.42'),
                    Link('bar', 'bar/')))
        return page

    def foo (self, id):
        page = HtmlPage(title='Foo %s, depth %d' % (id, self.depth))
        page.add(P('The id is ' + id))
        page.add(P('The depth is %d' % self.depth))
        return page

    def recurse (self):
        return TestDirectory(self.depth + 1)
```

Instead of typing this in, you can do:

```
>>> from seal.examples.ui import TestDirectory
>>> d = TestDirectory()
```

The method for accessing a page is `__getpage__()`. It takes three arguments: the page name (one of the keys in `pages`), a list of positional arguments for the corresponding method, and a dict of keyword arguments. The positional arguments default to `[]` and the keyword arguments default to `{}`. Remember that all arguments are strings. For example:

```
>>> d.__getpage__('')
<HtmlPage Hello>
>>> d.__getpage__('foo', ['2'])
<HtmlPage Foo 2, depth 0>
>>> d.__getpage__('bar')
<TestDirectory>
```

There should be little reason to use `__getpage__()` apart from debugging. It is meant to be called by `__call__()`, which implements the `seal.wsgi` application protocoll.

36.1.3 Request

One `HtmlDirectory` serves as the root directory for the application. It receives HTML requests, and handles them by walking recursively down the directory hierarchy with a sequence of `__getpage__()` calls.

A digested HTML request is represented by the class `Request` (from `seal.wsgi`). The `Request` constructor is not intended for general use; one should instead use the function `parse_request` (from `seal.ui`):

```
>>> r = parse_request('/bar/bar/foo.3')
>>> r
<Request /bar/bar/foo.3>
```

The request can then be used as a key to access the root of the directory hierarchy:

```
>>> d(r)
<HtmlPage Foo 3, depth 2>
```

The request translates the pathname into a sequence of `__getpage__()` calls. The call sequence is available in the member `callseq`:

```
>>> r.callseq
[('bar', [], {}), ('bar', [], {}), ('foo', ['3'], {})]
```

The elements in a call are (*name, args, kwargs*), where *name* is the page name, *args* is the list of positional arguments that the corresponding method receives, and *kwargs* is a dict giving values for keyword arguments. Note that the keys in `pages` are *page names;* the values are *method names.*

36.1.4 Running an application

`HtmlDirectory` implements the `seal.wsgi` application protocoll. One runs a web application as discussed in the chapter on `seal.wsgi`. For example:

```
>>> from seal.wsgi import run
>>> run(d)
```

Then visit `http://localhost:8000`.

36.2 More on HTML Directories

As we have seen, there are two ways to access an HTML directory. In **local access**, the directory is called as a function, the first argument being the page name and the remaining arguments being whatever the corresponding method requires. In **recursive access**, the directory represents the hierarchy of which it is root. The directory is accessed like a dict, using a `Request` as key, and the return value is a `Response` object.

36.2.1 Pathnames, forms, and `Request`

A `Request` encapsulates the information contained in an HTML request. To be precise, it has the following members:

`root_filename` URLs contain pathnames that begin with the WSGI script name. The WSGI server strips the script name before passing the pathname to the app. `Root_filename` is the stripped prefix. It is guaranteed either to consist of `/` followed by additional characters, or to be the empty string.

`pathname` The pathname of the requested item, relative to `root_filename`. Despite being a relative pathname, it is guaranteed to begin with `/`.

`cpts` The components of `pathname`. Remove the leading slash and split `pathname` on `/` to obtain the components.

`form` A dict representing form data. Keys prefixed by star in the original form have lists of strings as value, and keys not prefixed by star have single strings as value.

`callseq` The sequence of calls to be made to walk down the directory hierarchy. There are as many calls as there are pathname components.

`user` The name of the user at the client end.

Usually the `Request` will be created by the WSGI server, but for debugging purposes we can use `parse_request`. Here is a more complete example:[1]

```
>>> r2 = parse_request('/bar/foo.2',
...                    [('age', '42'),
...                     ('*pets', 'Snoopy'),
...                     ('*pets', 'Garfield')],
...                    user='pat',
...                    root_filename='/foo.wsgi')
```

We will use `TestDirectory2` as an example. It is just like `TestDirectory` except for the method `foo()`:

```
    def foo (self, id, age='0', pets=[]):
        page = HtmlPage(title='Foo %s, depth %d' % (id, self.depth))
        page.add(UL('ID: %s' % id,
                    'Depth: %d' % self.depth,
                    'Age: %s' % age,
                    'Pets: %s' % ', '.join(pets)))
        return page
```

In the digested form, the key `'pets'` is list-valued and `'age'` is string-valued:

```
>>> sorted(r2.form)
['age', 'pets']
>>> r2.form['age']
'42'
>>> r2.form['pets']
['Snoopy', 'Garfield']
```

Here is the result of accessing the directory with the request:

```
>>> from seal.examples.ui import TestDirectory2
>>> d2 = TestDirectory2()
>>> p2 = d2(r2)
>>> print(p2)
HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Content-Length: 359

<html>
<head>
<title>Foo 2, depth 1</title>
<link rel="stylesheet" type="text/css" href="/.lib/default.css" />
<script src="/.lib/default.js" type="text/javascript"></script>
```

[1] Recall from the chapter on `seal.wsgi` that the star indicates that the key is list-valued rather than string-valued. Incidentally, this is a low-level issue. The element-creating functions described in §36.5 are smart about which form elements are list-valued and which are string-valued. One does not add stars when using them; they add the stars themselves as needed.

```
</head>
<body>
<ul>
<li>ID: 2</li>
<li>Depth: 1</li>
<li>Age: 42</li>
<li>Pets: Snoopy, Garfield</li>
</ul>
<script type="text/javascript">sealSetup();</script>
</body>
</html>

```

Let us consider the call sequence:

```
>>> r2.callseq[0]
('bar', [], {})
>>> r2.callseq[1]
('foo', ['2'], {'age': '42', 'pets': ['Snoopy', 'Garfield']})
```

There are two things to note. First, the filename `'foo.2'` is split at dots. The first component is the page name, and the remaining components are arguments. Second, the form is passed to the last call in the sequence as a kwargs dict, to be picked up by the keyword parameters of the method `foo()`. (All calls before the last have an empty kwargs dict.) An error is signalled if the form data contains keys that the page method does not accept. However, no error is signalled if the form data is incomplete: the method's keyword parameters have default values.

As for the `user` and `root_name`, they are simply stored as members:

```
>>> r2.user
'pat'
>>> r2.root_filename
'/foo.wsgi'
```

36.2.2 `__parent__`, `__name__`, `__filename__`

When a root directory is accessed with a `Request`, the member `__filename__` of the root directory is set to the root filename, and `__name__` is the same without the leading slash. The member `__parent__` is set to `None`. Then as each local directory is accessed while recursing down the hierarchy, its value for `__parent__` is set to the directory from which it was locally retrieved, `__name__` is set to the pathname component, and `__filename__` is set to the parent's filename plus the child's name.

```
>>> d2.__parent__
>>> d2.__filename__
'/foo.wsgi'
>>> p2.__name__
```

```
    'foo.2'
    >>> p2.__filename__
    '/foo.wsgi/bar/foo.2'
    >>> p2.__parent__.__filename__
    '/foo.wsgi/bar'
```

36.2.3 Trailing slashes

The addition of a trailing slash merits a bit of commentary. Consider the URLs:

```
http://localhost:8000/foo
http://localhost:8000/foo/
```

Both identify the same item, which we may take to be a directory. Suppose that we generate the same web page to represent the directory, and that that web page contains a link to `bar.html`. Depending on which URL the browser accessed the page under, it will interpret the link differently:

```
http://localhost:8000/bar.html
http://localhost:8000/foo/bar.html
```

The pathname with a trailing slash (`/foo/`) parses differently than the one without (`/foo`). The former yields the parsed path `['foo']`, whereas the latter yields `['foo', '']`. Redirecting to the path with a trailing slash may help, because if `['foo']` leads to a directory, `['foo', '']` will access it with the empty string, which may produce a renderable page.

36.2.4 Library requests

The Seal `MainFunction` intercepts certain paths and handles them itself, rather than using the directory. In particular, a path that begins with "`/.lib/`" is taken to name a file in the directory `/cl/data/seal`.

One can set "`.lib`" to a different value by passing a value for `libkey` to the `MainFunction` constructor, or by setting the attribute `libkey`. Setting it to `None` disables the behavior.

36.3 Web pages

These are various specializations of `Response`. All of them can be rendered to an HTTP connection, and are appropriate values for a main function.

36.3.1 HtmlPage

An `HtmlPage` object represents a regular HTML page. The head and the start and end tags for the body are generated automatically. One need provide only the contents of the body. For example:

```
>>> page = HtmlPage(title='Hello')
>>> page.add(H1('Hello'))
>>> page.add(P('Hello, world!'))
```

This renders as:

```
>>> print(page)
HTTP/1.1 200 OK
Content-Type: text/html;charset=utf-8
Content-Length: 289

<html>
<head>
<title>Hello</title>
<link rel="stylesheet" type="text/css" href="/.lib/default.css" />
<script src="/.lib/default.js" type="text/javascript"></script>
</head>
<body>
<h1>Hello</h1>
<p>Hello, world!</p>
<script type="text/javascript">sealSetup();</script>
</body>
</html>

```

One may also provide initial contents as an argument to the constructor:

```
>>> page = HtmlPage(H1('Hello'), P('Hello, world!'))
```

Any subsequent writes will append to the initial contents.

A third alternative is to specify a filename containing the contents.

```
page = HtmlPage(src='/foo/bar.html')
```

Note that the named file should contain only the contents of the body, not a complete HTML page.

An additional optional argument for `HtmlPage` is `default_stylesheet`. If not `None`, it is placed at the head of the list of stylesheets that are included in the header. The default value is `/.lib/default.css`.

If control over the head is desired, the following facilities are provided. One may set the title:

```
page.title = "Some Stuff"
```

One may add (a link to) a stylesheet or javascript file:

```
page.add_stylesheet("foo.css")
page.add_javascript("foo.js")
```

Or one may add arbitrary lines at the end of the head:

```
page.head_write(Script('foo'))
```

36.3.2 Raw html page

An `HtmlPage` seeks to make it easy to do common tasks, but it somewhat limits one's flexibility. If you wish to specify the complete contents of the entire HTML page, you may create a `RawHtmlPage` instead. It has no useful methods; one initializes it with the page contents as one large string:

```
page = RawHtmlPage(contents)
```

In this case, the contents should include both the head and the body. Nothing will be added.

36.3.3 Raw file

One can also return the contents of a file in response to a request:

```
page = RawFile(filename)
```

The content-type is determined from the filename suffix. The suffixes currently recognized are: `.css`, `.html`, `.js`, `.pdf`, `.txt`, `.wav`.

36.3.4 Redirect

A handler may return a `Redirect` object to indicate that the browser should send a new request with a different URI. This may be appropriate not only when a resource has moved, but also in response to a request that creates a new page.

```
def myhandler (request):
    return Redirect('foo')
```

36.3.5 Exceptions

A main function may either return a `Response` or an `HttpException`. The latter causes an error response to be generated. The class `MainFunction` catches any `HttpException` that is raised, and returns it. Also, if a directory `__getitem__()` returns `None`, the main function returns `PageNotFound`. (The class `PageNotFound` is defined in `seal.html`.)

36.3.6 Utility functions

The function `escape()` replaces ampersand, less than, and greater than characters with the corresponding HTML character entities. (It is essentially the same as `cgi.escape()`.)

```
>>> escape('x<y & y>z')
'x&lt;y &amp; y&gt;z'
```

36.4 Elements

A variety of functions are available for building a web page as a hierarchical structure of HTML elements.

36.4.1 Element

The class `Element` is the base class for the following. Every element has `contents`, which is a list of strings or subelements. `Element` also provides two methods: `add()` is used to add additional items to the contents, and `render()` writes the contents as an HTTP response.

36.4.2 Spans

There are elements corresponding to the font-changing tags `B`, `I`, and `TT`, as well as the headers `H1` to `H6`.

```
>>> e = B('test 1 2 3')
>>> print(e)
<b>test 1 2 3</b>
```

They accept multiple arguments.

```
>>> e = H1('The ', I('Titanic'))
>>> print(e)
<h1>The <i>Titanic</i></h1>
```

36.4.3 Spacers

`BR` is not a function but a variable.

```
>>> print(BR)
<br />
```

`NBSP` is a tag, but it also can be called as a function taking an optional argument that indicates the number of spaces.

```
>>> print(NBSP)
 
>>> e = NBSP(2)
>>> print(e)
  
```

36.4.4 Blocks

`P` produces a paragraph. It accepts any number of arguments. `Pre` produces a pre-formatted block. It accepts a single argument, and an optional `width` parameter. Note that the restriction to a single item is not limiting: one can pass in a list.

```
>>> e = Pre(['hi there\r\n', 'foo bar\r\n'])
>>> print(e)
<pre class="source">
hi there
foo bar
</pre>

```

36.4.5 Lists

`UL` takes multiple arguments. Each is rendered as a list item. One may also create a `UL` and add items one at a time.

```
>>> e = UL('Lather', 'Rinse', 'Repeat')
>>> print(e)
<ul>
<li>Lather</li>
<li>Rinse</li>
<li>Repeat</li>
</ul>
```

Additional items can be added using the `add()` method.

`Stack` is not a standard HTML element. It takes multiple arguments, and connects them with `BR`'s.

```
>>> e = Stack('Hi there', 'A test')
>>> print(e)
Hi there<br />
A test
```

36.4.6 Table

One can create a table all at once, or add a row at a time, or mix the two modes.

```
>>> e = Table(Row('hi', 'there'), Row('foo', 'bar'))
>>> print(e)
<table class="display">
<tr><td>hi</td><td>there</td></tr>
<tr><td>foo</td><td>bar</td></tr>
</table>
```

One may also use `Header` instead of `Row`. A `Header` is a row in which each cell is wrapped in `th` instead of `td`. Additional rows can be added using the `add()` method.

To change the `rowspan` or the `colspan` of a cell, one must create an explicit `Cell` object. For example:

```
>>> t = Table('hi', 'there'), Row(Cell('boo', colspan=2)))
>>> print(t)
<table class="display">
<tr><td>hi</td><td>there</td></tr>
<tr><td rowspan=1 colspan=2>boo</td></tr>
</table>
```

36.4.7 Navigation

Link. A `Link` represents an HTML anchor. It takes two arguments: the text and the URL.

```
>>> e = Link('go there', '/foo')
>>> print(e)
<a href="/foo">go there</a>
```

An optional third argument is the `target`. Typical values are `_top` or `_blank`.

Button. The `Button` constructor takes two arguments: the text that appears on the button, and the URL to be visited if the button is clicked on. If the URL is `None`, the button is disabled. An optional argument is `target`, which specifies the window that the URL should be opened in.

Path. A `Path` is a sequence of links representing the path to the current directory. It takes an `HtmlDirectory` as argument. For example:

```
>>> from seal.examples.ui import RootDirectory
>>> root = RootDirectory()
>>> text = root(parse_request('doc.10/page.3/text'))
>>> page = text.__parent__
>>> print(Path(page))
<div class="path">
<a href="/">(Root)</a> &gt;
<a href="/doc.10/">doc.10</a> &gt;
<a href="/doc.10/page.3/">page.3</a> &gt;
</div>
```

Menubar. A `Menubar` is a `div` created from a list of buttons.

36.5 Forms

Forms comprise a number of different elements, so we put them in a section of their own.

36.5.1 Form element

The `Form` constructor takes a single argument, which is the callback URL. The information in the form will be `POST`ed to the callback URL when the form is submitted.

```
>>> e = Form('do_it')
>>> e.add(Submit('Go!'))
>>> print(e)
<form enctype="multipart/form-data" action="do_it" method="post">
<input type="submit" name="action" value="Go!"/>
</form>

```

Each form element generates a key-value pair in the `POST` data. Each of the following constructors takes a key as its first argument.

36.5.2 Check boxes

The `CheckBoxes` constructor takes two arguments: a key and a list of values. One checkbox is generated for each value. An optional argument is `selected`, which may be a value or a list of values that should initially be checked. By default, no boxes are checked. Another optional argument is `separator`, which specifies what should be placed between each pair of adjacent check boxes. By default, it is a single space.

36.5.3 Dropdown

The `Dropdown` class represents a dropdown list. The constructor takes two arguments: key and values. The key identifies this piece of information in the form. Values is a list of possible values. The initially selected value is the first in the list. An optional argument `selected` allows one to specify one of the other values as the initially selected value.

36.5.4 File upload

A `File` element supports file upload. In the form, it takes the form of a browse button that allows a user to select a file. In the `POST` information, the entire contents of the file, as a string, is the value of key associated with the `File` element.

Here is an example:

```
class FileTest (HtmlDirectory):

    def getitem (self, name, args, kargs):
        if name == '': return self.test()
        elif name == 'upload': return self.upload(**kargs)

```

```
    def test (self):
        form = Form('upload')
        form.add(Table(Row('File:', File('file')),
                       Row(Cell(Submit('Submit'), colspan=2))))
        p = HtmlPage(title='File Test')
        p.add(form)
        return p

    def upload (self, file='', submit=''):
        p = HtmlPage(title='File Contents')
        p.add(Pre(file))
        return p
```

To run it:

```
>>> from seal.wsgi import App, run
>>> from seal.examples.ui import FileTest
>>> run(App(FileTest()))
```

Then visit `http://localhost:8000/`.

36.5.5 Hidden

A `Hidden` element can be used to pass information from the code that creates the form to the code that receives the resulting `POST`. The constructor takes two arguments: key and value.

36.5.6 Not editable

The `NotEditable` constructor takes two arguments, key and value. Like a hidden element, the key-value pair is included in the `POST`. But unlike a hidden element, the value is displayed—though it is not editable.

36.5.7 Radio buttons

The `RadioButtons` constructor takes two arguments: a key and a list of values. Each value generates a radio button. An optional argument is `selected`, which contains one of the values. By default, none of the boxes is initially selected. Another optional argument is `separator`, which specifies what should be between each pair of adjacent radio buttons. By default, it is a single space.

36.5.8 Submit

A `Submit` button constructor takes a single argument: the text to display on the button. It generates a key-value pair in which the key is "submit" and the value is the text.

36.5.9 Text box

The `Textbox` constructor takes two arguments: key and value. The value provides the initial text in the box. If omitted, it defaults to the empty string. An optional argument is `size`, whose value is an integer representing the width of the text box in characters.

36.5.10 Text area

The `Textarea` constructor is just like `Textbox`, except that the `size` parameter expects a pair of numbers, representing the number of rows and columns in the box.

36.5.11 Example

The class `FormTest` illustrates the use of a form. It is defined as follows:

```
class FormTest (HtmlDirectory):

    def getitem (self, name, args, kargs):
        if name == '': return Redirect('form.42')
        elif name == 'form': return self.form(*args)
        elif name == 'update': return self.update(**kargs)

    def form (self, id):
        t = Table(Row('Name:', Textbox('name')),
                  Row('Password:', Password('passwd')),
                  Row('Sex:', RadioButtons('sex',
                                           ['Female', 'Male'])),
                  Row('Income:', Dropdown('inc',
                                          ['', 'Some', 'Lots'])),
                  Row('Pets:', CheckBoxes('pets',
                                          ['Dog', 'Cat', 'Python'])))
        form = Form('update')
        form.add(t)
        form.add(Hidden('id', id))
        form.add([Submit('Submit'), NBSP(), Submit('Cancel')])

        p = HtmlPage(title='Form Example')
        p.add(H1('Form'))
        p.add(form)
        return p

    def update (self, id='', name='', passwd='', sex='',
                inc='', pets=[], submit=''):
        p = HtmlPage(title='Update')
        p.add(Table(Row('Id:', id),
```

```
                    Row('Name:', name),
                    Row('Password:', passwd),
                    Row('Sex:', sex),
                    Row('Income:', inc),
                    Row('Pets:', ', '.join(pets)),
                    Row('Submit:', submit)))
        return p
```

Note the line `getlist('pets')` in `update()`. With check boxes, multiple boxes may be checked, yielding multiple values for "pets."

To run the test:

```
>>> d = FormTest()
>>> d.run()
```

Visit `http://localhost:8000/`. The browser will redirect to `form.42`. Fill in some information and click either "Submit" or "Cancel." You should get a web page showing what you entered.

36.6 Editors

36.6.1 Datum editor

A `DatumEditor` is a specialization of `HtmlDirectory` that provides functionality for viewing and editing a `Datum`. Here is an example:

```
>>> from seal.db import *
>>> from seal.ui import DatumEditor, run
>>> fs = [Field('id', width=4),
...       Field('name', immutable=True),
...       Field('boss', indexed=True)]
...
>>> table = Table(tmpfile(), schema=fs)
>>> table.new('Abby', '')
>>> table.new('Beth', '1')
>>> table.new('Charley', '1')
>>> table.new('David', '2')
>>> run(DatumEditor('Item', table['2']))
```

It provides facilities for viewing, editing, and deleting the given datum.

One can use the editor to create a new item by giving it a table instead of an existing item:

```
>>> run(DatumEditor(table=table))
```

36.6.2 Data table editor

A `DataTableEditor` is used to edit a `DataTable`. The constructor takes a data table as input. It also takes two optional arguments: `name` and `action`. The `name` argument allows one to provide a name for the table, which is used for the HTML page title. The opening display is a listing of the items in the table. The entries in the id column are links, and the `action` argument lets one specify what is to be done when one clicks on one of the links. The two legal values are `view` and `edit`.

36.7 Convenience module: seal.html

The module seal.html imports the HTML pages, elements, and editors, but no additional symbols. This allows one to do:

```
>>> from seal.html import *
```

without importing unexpected symbols.