atox

User Manual

Magnus Lie Hetland

Version 0.5 **April 23, 2004**

Contents

Atox is a framework for automated markup. With it one can quite easily write custom scripts for converting plain text into XML in any way one wishes. Atox is normally used as a command-line script, using a simple XML language to specify the desired transformation from text to markup, but it is also possible to build custom parsers using the Atox library. The name (short for ASCII-to-XML) is inspired by such UNIX tools and system functions as `atops` and `atoi`.

1 Introduction

Documents have structure. Simple, plain text without any kind of structure is often not good enough to represent what you want to put into your documents. This is why we have markup, which comes in several flavors. One is, of course, the closed, proprietary form of "invisible" markup you get in a binary document from a word processing program such as Microsoft Word. Another is the form of markup that is actually represented as some form of tagging of the text. The most standardized and popular markup language at the moment is probably XML, and using various tools (such as an XSLT processor) XML markup can be transformed into other markup formats (such as TEX or RTF, for example).

However, in many cases writing XML manually can be impractical. The reasons may be many — you may simply be tired of typing the tags (even though a good editor will help here), you may be working with people who don't want to learn about markup, or you may be working with documentation inside comments in source code. There are editors out there that can make the XML part almost invisible to you in the cases where this is feasible, but working with plain text still has its advantages. It is easily sent over (and read in) email, it interacts favorably with version control systems such as CVS, and in some cases you may simply have existing text without markup that needs to be structured automatically. Enter automatic markup.

There are many solutions for adding markup automatically to plain text; here are some (but by no means all):

- Wiki Text Formatting[1]: Some formatting rules that are used in Wikis.
- Setext[2]: A precursor to reStructuredText.
- reStructuredText[3]: A markup language with lots of functionality, based on StructuredText and its ancestors.
- txt2html[4]: A tool for converting text to HTML, with quite flexible interpretation of the input. There is also a version supporting TEX output.
- David Mertz's Smart ASCII[5]: Mertz's used this in writing his book "Text Processing in Python."
- Markdown[6]: A text-to-HTML converter based on lessons learned from several other similar tools (but still with a fixed input format).
- Jeni Tennison's XSLT Markup Utility[7]: A simple utility to mark up words in plain text using XSLT directly.

[1] `http://www.c2.com/cgi/wiki?TextFormattingRules`
[2] `http://docutils.sourceforge.net/mirror/setext.html`
[3] `http://docutils.sf.net/rst.html`
[4] `http://www.aigeek.com/txt2html`
[5] `http://gnosis.cx/TPiP`
[6] `http://daringfireball.net/projects/markdown`
[7] `http://www.jenitennison.com/xslt/utilities`

- And many, many more (such as atx[8], textile[9], Grutatxt[10], EtText[11], Latte[12], Confluence[13]...).

There are, as you can see, many formats out there. (Even more, including a proposal for standardization, are described on Bill Seitz's Smart ASCII page[14]). I have too, for several years, pondered various possible ways of making such implicit plain-text markup as "invisible" as possible (and implemented quite a few scripts for this sort of thing along the way).

As I am sure you have guessed, I'm not completely satisfied with any of the existing solutions that I've seen so far. But I have also realized that if I simply introduced *another* format, I would only solve the problem for myself, because others probably wouldn't be satisfied with that. Also, the solution most likely would be temporary, because I'm likely to change my mind about how I want the format to be, and new applications would drive me to introduce new features, leading to a bloat similar to that found in some other formats.

The problem, in my opinion, with most of the existing tools, is that the input format is fixed, and so is the existing output formats. My idea for a solution was twofold:

1. Let the user specify the input format.
2. Use user-specified XML as the output format, and encourage the use of XSLT to further create other forms of output.

In many ways, Atox the becomes for plain text what XSLT is for XML: A way of describing a transformation. Although Atox is quite easily usable as a library for plain text parsing (producing a DOM tree based on the grammar supplied by the user) this API is not described in the current documentation, mainly because it isn't completely stable yet. The XML format used to describe the input format, however, and the command-line tool used to apply it, are described in the following sections.

1.1 How to Get It

Atox is currently hosted at SourceForge, at `http://atox.sf.net`. The current development sources are available through CVS, and the most recent stable version may be downloaded from the SourceForge project page[15].

[8] `http://www.aaronsw.com/2002/atx`
[9] `http://textism.com/tools/textile`
[10] `http://www.triptico.com/software/grutatxt.html`
[11] `http://ettext.taint.org/doc/ettext.html`
[12] `http://www.latte.org`
[13] `http://confluence.atlassian.com/renderer/notationhelp.action`
[14] `http://webseitz.fluxent.com/wiki/SmartAscii`
[15] `http://sf.net/projects/atox`

1.2 Usage

Using the `atox` program is very easy:

```
$ atox -f markup.xml input.txt > output.xml
```

The file `markup.xml` describes the structure of the input text, as well as how to mark it up, using the format language described later in this document. The file `input.txt` contains the text to which markup should be added, and `output.xml` is the resulting XML file.

In general, you can specify zero or more input files. If none is specified, standard input is used.

Atox can be customized by setting several options. You can set these by creating a config file called `.atoxrc` in your home directory, using the fairly standard configuration syntax (without variable interpolation) described in the Python Library Reference section on ConfigParser[16], with the exception that the config file should not have any sections. Two alternative ways of setting options exist: Using environment variables or command-line switches.

If you want to set an Atox option using environment variables, use the `ATOX_` prefix, and put the name in uppercase. In other words, to modify the option `input_encoding`, use an environment variable called `ATOX_INPUT_ENCODING`. To set options through command-line switches, use the name of the option as a long option, with underscores changed to dashes. For example, the option `format_file` can be set using the switch `--format-file`. In some cases, you can also use the first character of the option as a single-character switch (for example `-f` for `format_file`).

The following lists all the available options, along with the short command-line switches, where available.

- `format_file` (`-f`) is the name of the format file, optionally without the `.xml` suffix. The default is `markup.xml`.
- `output_file` (`-o`) is the output file, with the default being standard output.
- `input_encoding` sets the encoding to use for plain-text input, with the default being `iso8859-1`.
- `output_encoding` sets the encoding to use for XML output, with the default being `utf-8`. Note that if you use XSLT fragments in your format file, your XML output file will be fed through an XSLT processor. The `output_encoding` variable will then only determine the encoding used for the file that is fed *to* the XSLT processor, not the file that comes *out* of it. In order to affect the encoding of that file, you should use an `xsl:output` declaration (see any XSLT referenc for more information).

[16] `http://www.python.org/doc/lib/module-ConfigParser.html`

- `xslt_command` is the command used to process XSLT internally. This is only used when you put XSLT fragments in your format file. (See *Enlisting* XSLT, later in this document.) The strings `${in}`, `${out}` and `${xsl}` are used as placeholders for the input file, output file and XSLT stylesheet. The default is `xsltproc -o ${out} ${xsl} ${in}`. (Note: If you're setting this command as an environment variable, make sure you backslash-escape the dollar signs.)
- `input_ext` is the file extension of input files. If this is left out in input file names, it is automatically added *unless* the file name/path already contains a dot after the last path separator (and therefore, probably, another extension). The default is `.txt`.
- `format_ext` is the file extension of format files. If this is left out in format file names, it is automatically added *unless* the file name already contains a dot (and therefore, probably, another extension). The default is `.xml`.

So, for example, the following will will have the same effect as the previous command line:

```
$ cat atox.txt | ATOX_OUTPUT_ENCODING=utf-8 atox -o foo.xml
```

If, for example, you wanted to use `utf-8` as your default input encoding, you could put the following in your `.atoxrc` file:

```
input_encoding = utf-8
```

It is possible to produce (subsets of) some useful formats such as DocBook or XHTML directly with Atox, but in most cases you'll want to transform the XML into something else, using, for example, an XSLT processor such as xsltproc[17], Xalan[18] or Saxon[19]. For more information on XSLT, see the W3C Web site[20] or one of the many online tutorials (such as the one from W3Schools[21]). See also the `doc` directory of the Atox distribution for an example of how XSLT can be used to transform Atox output into HTML and LaTeX. For information on how to use XSLT directly in your Atox format files, see Enlisting XSLT, below.

1.3 The Small Print

Atox comes with no warranty of any kind. Also, even though the current version works well, and the project is currently (as per early 2004) being actively developed, there is no guarantee of continued support. What you see is what you get.

[17] `http://xmlsoft.org/XSLT`
[18] `http://xml.apache.org`
[19] `http://saxon.sf.net`
[20] `http://www.w3.org/TR/xslt`
[21] `http://www.w3schools.com/xsl`

2 The Atox Format Language

The Atox parser is a deterministic top-down left-to-right parser, somewhat similar to an LL(1) parser, but with a somewhat non-standard tokenization. Each token is represented as a regular expression (actually, you may implement any kind of parser and use it as a token). The text is then tokenized as it is parsed: When the parser decides which rule to apply, it searches for all the applicable tokens, and applies the rule that occurs earliest in the input text. For example, using the `alt` tag (with the Atox namespace) you can make the parser choose between two alternatives, like this:

```
<ax:alt>
  <f>foo</f>
  <b>bar</b>
</ax:alt>
```

When the parser tries to parse this pair of alternatives at any point in the text, it will look for `foo` and `bar` and choose either the `f` or `b` element, depending on which token appears first.

In some cases, two alternative tokens may occur at the same position in the text; in this case, the parser chooses the one that is listed first inside the `alt` tag. So, for example, in the following (contrived) example, the `a` element will be parsed:

```
<ax:alt>
  <a>foo</a>
  <b>foo</b>
</ax:alt>
```

The moral of this last mechanism is that you should place your most specific alternatives first. For example, the first token of a list element and that of a paragraph may occur at the same position in the input; however, because the list item is more specific (you want it to *override* the paragraph) it should be put first.

There are many excellent parser generator toolkits available for Python, some of them using more fancy parsing strategies than the LL-like one used here. I have, in fact, tried some other parsing strategies (and existing parsing packages) in earlier versions, but ended up with this one, partly because of the need to support the somewhat unusual tokenization method, and partly because a deterministic top-down parsing strategy seems to work well with plain-text documents, and also leads to significant speedups over some other methods. (The current Atox implementation isn't exactly lightning fast, though.)

There is one thing you should be aware of, though: The parser *only* considers the next token when choosing what to parse. This means that if you want it to recognize a text-block as a header (rather than a paragraph, for example) because it is underlined in some way, you must give it some form of lookahead. There is currently no support for this directly in Atox, but, luckily, the regular

expression language used supports lookahead quite nicely. The resulting regular expressions aren't always pretty, but it works. (This might be improved in future versions.)

2.1 Format Description Structure

The main document element of a format description is `format`. It should contain a definition of the Atox namespace, called `ax` by convention:

```
<?xml version="1.0"?>
<ax:format xmlns:ax="http://hetland.org/atox">
  <!-- Put the rest of the format description here -->
</ax:format>
```

The format descriptions are similar to XSLT stylesheets in that they mimic the output by using the desired output tags, and at the same time they are similar to XML schemas, which describe the structure of their input:

```
<?xml version="1.0"?>
<ax:format xmlns:ax="http://hetland.org/atox">
  <article>
    <!-- Content structure here -->
  </article>
</ax:format>
```

This will add a document tag called `article` to the output.

The top-level document element is special — it will always enclose the entire document. All other output tags (those that do not use the Atox namespace) only enclose the occurrences tokens they contain. An example might clarify this (I'm dropping the `format` tag here; it is implied in this and the following examples):

```
<salutation>
  <em>Hello</em>
</salutation>
```

This format will enclose the entire input in the `salutation` tag, but only the first occurrence of the string `Hello` will be enclosed in an `em` tag. In other words, `Hello, world!` will be transformed into

```
<salutation><em>Hello</em>, world!</salutation>
```

If an output tag contains several tokens (or other structures, which themselves contain tokens) it will enclose them all in the output.

The following sections describe the various special format tags (those that use the Atox namespace) you can use to specify the structure of the input.

2.2 Patterns

Any plain text will be interpreted as a regular expression, or *pattern*, currently the only form of built-in token in Atox. A series of patterns may only be broken up by tags — if you try to separate them with whitespace, for example, they will be treated as a single pattern. Whitespace at the beginning and end of the pattern is stripped, but internal whitespace is kept. The regular expression language is described in the section about the `re` module in the Python Library Reference[22].

Patterns are used without any flags (such as multiline, verbose and so forth), but those can be specified by using the `(?...)` syntax (also described in the Library Reference). Note that line breaks are normalized by Atox, so that all line break combinations (that is, `\r`, `\r\n` and `\n`) are treated as single newline characters, and may be matched as such in patterns.

In addition to the implicit syntax for patterns, you can use the `pat` tag to be explicit. For example,

```
<ax:pat>F.*r</ax:pat>
```

will match (among other things) `Foobar`. Note the use of the Atox namespace prefix — without it, this would be interpreted as an output tag named `pat`.

There is a built-in set of patterns that is quite useful because they occur frequently in practice, but are a bit tiring to type. They are the *start-of-block* and *end-of-block* patterns. They represent the start and end of a block (a lump of text, separated from other blocks by empty lines) such as paragraphs, block quotes or list items.

The start-of-block pattern is available through the `ax:sob` tag, and is equivalent to `(?=\S)` which means that the next character is a non-whitespace character. The end-of-block pattern is available through the `ax:eob` tag, and is equivalent to the somewhat more cumbersome `(?=\s*\Z|\n\s*\n)`, which means that the following text is either an optional run of whitespace, ending at the end of the text, *or* two newlines, possibly separated with some whitespace. Both of these should always be used as empty tags. While they can be useful on their own (for example, the `ax:eob` tag can be useful to end blocks that start with some specific pattern, such as a list item), they are often used as a pair. You can then use the `ax:block` element instead. It is a form of sequence (see a description later in this document section), so it can contain other contents (or not) but it will always implicitly begin with an `ax:sob` and end with an `ax:eob`.

As an example, the following is a natural way of implementing paragraphs:

```
<p>
  <ax:block/>
</p>
```

If you have inline elements, such as emphasis (examples of this may be found later in this document), they can be put inside the `ax:block` element.

[22] `http://www.python.org/doc/lib`

2.3 Deleted Patterns

Sometimes you want to match a pattern in the input text, but keep drop it in the output. For example, a `note` might start with the word `Note:` followed by some whitespace, and it might end with a double newline. Since you're using a `note` tag in the output, there is no need to keep the `Note:` string:

```
<document>
  <note>
    <ax:del>Note:\s+</ax:del>
    <ax:eob/>
  </note>
</document>
```

In order to get the `note` element to span the entire note, I've used an `ax:eob` (end-of-block) to mark the end of the contents.

2.4 Sequences

Any two tags that follow each other will implicitly be interpreted as a sequence. It is possible, however, to be explicit about this too, using the `seq` tag:

```
<ax:seq>
  <ax:pat>foo</ax:pat>
  <ax:del>bar</ax:pat>
</ax:seq>
```

Note that sequences (including all non-`atox` elements, including the main document element) can be empty. (Empty sequences can be matched anywhere by Atox, because they match any empty slice of text, just like the empty pattern.)

2.5 Alternatives

Sometimes you want the parser to choose between several alternatives. This basically means that the input may contain one of several possible structures. It is important to note that the parser *must* be able to choose among these alternatives by only looking at the *first* token (pattern) in each of them.

Alternatives are represented with the `alt` tag:

```
<ax:alt>
  <ax:pat>foo</ax:pat>
  <ax:pat>bar</ax:pat>
</ax:alt>
```

2.6 Repetition

Normally, each element occurs once and only once. You can, however, adjust this by setting the `minOccur` and `maxOccur` attributes. For example, the following will match anything from zero to five occurrences of `foo`:

```
<ax:pat minOccur="0" maxOccur="5">foo</ax:pat>
```

As you may have surmised, the defaults for both of these attributes are 1. When using them on structure tags (those with the Atox namespace) there is no need to give an explicit namespace. However, you may also use the attributes on output tags — in this case, you should be explicit about the namespace. Here is an (incomplete) example of how `minOccur` and `maxOccur` might be used on an output tag:

```
<list>
  <item ax:minOccur="1" ax:maxOccur="inf">
    <!-- Add the content structure of list items here -->
  </item>
</list>
```

Note the use of `inf` (short for "infinite") in `maxOccur` here — that means that there is no upper limit to the number of occurrences.

There is one additional quirk about repeated elements that's worth mentioning — in fact, it can be quite important in many cases: By default, the parsing of repeated elements is *greedy*. Like with regular expressions, this means that Atox will match as many occurrences of the repeated elements as possible (and allowed by `ax:maxOccur`). Sometimes what you want is *non-greedy* matching, that is, you want to match only as many occurences as can be found *before* the first occurrence of the following element. Here is an example where greedy matching will foul things up:

```
<letter>
  <p ax:minOccur="0" ax:maxOccur="inf">
    <ax:block/>
  </p>
  <ps>
    <ax:del>PS:\s+</ax:del>
    <ax:eob/>
  </ps>
</letter>
```

This shows the structure of a letter, which consists of a sequence of paragraphs, followed by a postscript, that is, a paragraph that begins with `PS:`. You could structure this differently (for example, by a repeated `ax:alt` element, where the `ps` element came before the `p` element), but if you try to use it the way it is, you will invariably get an error message — the `PS:` pattern wont't be found. Why? Because Atox has already parsed all paragraphs as `p` elements, using

greedy matching, so there is nothing left to be parsed as a `ps` element. What we need is to turn on non-greedy parsing, so that once a `ps`-compatible paragraph comes along, the stream of `p` elements will come to a halt. That's easily doable by setting the `ax:greedy` attribute to a false value (either `false`, `no`, `off`, or `0`, as opposed to `true`, `yes`, `on`, or `1`, with all values being case-insensitive). The default value is, of course, `true`, but you can set that explicitly too, if you want to. In other words, adding `ax:greedy="false"` to the `p` element in the previous example will make it behave the way we want.

One thing is worth noting about non-greedy repetition: It won't be interrupted by *empty matches*. For example, an empty sequence or an element with `ax:minOccur` set to zero can match an empty piece of the input at any point. If Atox were to let such matches interrupt a non-greedy repetition, the repetition wouldn't get anywhere. So empty sequences won't interrupt it at all, and only non-empty matches of optional elements (with `ax:minOccur="0"`) will interrupt the repetition. The element occurring *after* one (or more) such empty match *can*, however, interrupt the repetition. (If you find this stuff confusing, rest assured that you can probably just ignore it. Atox is designed to do the "right thing" here.)

2.7 Indentation

A powerful mechanism for indicating structure in plain text is indentation. In Atox you may parse indentation using the `ax:indent` and `ax:dedent` tags, indicating that the current line has ha higher or lower indentation level, respectively, than the current level. Usually, an easier way of handling indentation, though is to use the `ax:indented` tag, which, in effect, combines one `ax:indent` and one `ax:dedent` (the indent at the beginning and the dedent at the end of the tag), and makes sure the dedent goes back to the level before the indent. That way, you don't have to keep track of any additional indentation inside the indented block to make sure the indent and dedent match up. The `ax:indented` element can be used without any contents, just like the `ax:block` element.

An example of using indentation to parse nested lists as well as one for parsing indented code blocks may be found in the `demo` directory of the Atox distribution. The rest of this section gives a more thorough description of the indentation rules used.

The indentation rules used are quite similar to those defined in the Python language reference[23]. The rules may be summed up as follows:

- A (logical) line is simply a physical line (terminated by a newline)
- Each line containing non-whitespace characters has an indentation level, determined by the whitespace at the beginning of the line.
- A whitespace-only line does not have an indentation level and does not affect the indentation structure.

[23] `http://www.python.org/doc/current/ref/indentation.html`

- Each space character adds 1 to the indentation level. A tab character is interpreted as the number of spaces needed to reach the next multiple of eight. No other whitespace characters are interpreted as part of the indentation.
- If a line has a higher indentation level than the highest one on the indentation stack (initially only contains zero), an indent event occurs (before the whitespace indentation). Only one indentation event may occur in one line. When an indent event occurs, the corresponding indentation level is pushed onto the indentation stack. The indent element skips over the whitespace that causes the indent, in the same way that a pattern skips over the matched text.
- If a line has a lower indentation level than the previous one, a dedent event *may* occur. A dedent event occurs whenever the indentation level is lower than or equal to an indentation level which is alredy found on the indentation stack, except for the most recent one (one event occurs per such passed). In other words, each time the indentation level returns to a previous indentation, a dedent event occurs and the corresponding level is popped off the stack (in addition to the top-level indent). If any dedents occurred (but not otherwise), the new level is pushed onto the stack. Dedent events occur at the end of the last non-whitespace line before the line which has a lower indentation level, and the dedent tag does not skip over any input text.
- End-of-File causes all current indentation levels to be dedented. (In other words, the end-of-file token is considered to occur at indentation level 0.)

2.8 Definitions

In many cases the structure of a format description may be made more readable by introducing *definitions*. You can define a sub-structure in one place and use it in another. This also allows a structure to contain itself (recursion) which can be useful in many cases (for example, if you want to allow emphasis within emphasis or quotes within quotes, and the like).

Definitions are made with the `def` tag:

```
<ax:def name="foo">foobarbaz</ax:def>
```

You can place any structure (that is, a combination of sequences, alternatives, repetitions, output tags and patterns) inside the definition. You can then later refer to it with the `match` tag:

```
<ax:match name="foo"/>
```

The parser will treat the description as if the sub-structure actually occurred at the location of the `match` tag.

For a more complete example of how all these tags are used, check out the file `markup.xml` in the `doc` directory of the distribution — it describes the plain-text

structure and XML output of this manual. For several smaller examples, see the `demo` directory.

2.9 Glue Patterns

Usually the standard Atox parsing mode is quite convenient — it searches for the first occurrence of a valid element and starts parsing that. However, sometimes you want to make sure that the parsed elements are contiguous. (This is what happens in ordinary LL(1) parsers, for example.) This is only relevant when the parser can make choice, in other words, for `ax:alt` elements and for repeated elements (those with `minOccur` and `maxOccur` set to different values).

Consider, for example, that you want to represent emphasis by enclosing words in asterisks, like `*this*`. Perhaps you also want this to represent links, when they are followed directly by URLs in parentheses. A simple way of implementing this would be the following:

```
<em>
  <ax:del>\*</ax:del>
  <ax:del>\*</ax:del>
  <url ax:minOccur="0">
    <ax:del>\(</ax:del>
    <ax:del>\)</ax:del>
  </url>
</em>
```

Note that the asterisks and the parentheses are escaped with backslashes — that is because asterisks and parentheses have special meanings as parts of regular expressions.

There is one problem with the scheme above. When applied to a piece of text where some text in parentheses occur after an emphasized piece of text, the parentheses and *all* the text in-between will be included. For example,

```
This is a *test*. (Fee fie foe.)
```

will be turned into:

```
This is a <em>test. <url>Fee fie foe.</url></em>
```

What we want is to require that the URL element occurs immediately after the emphasis, possibly with some whitespace in-between. We do this by *gluing* the first (deleted) pattern in place (and adding the proper whitespace to it):

```
<em>
  <ax:del>\*</ax:del>
  <ax:del>\*</ax:del>
  <url ax:minOccur="0">
    <ax:del glue="">\s*\(</ax:del>
    <ax:del>\)</ax:del>
  </url>
```

```
</em>
```

The attribute used is called `glue` (and as with `minOccur` and `maxOccur` you should add the `ax` namespace to it when using it on non-Atox elements). It is a regular expression, just like the body of `pat` elements, and places a restriction on the text that may be skipped before finding the element. In this example an empty piece of glue is used, meaning that the `del` element must be located immediately, without skipping anything.

Note that I glued the `ax:del` element and not the `url` element. That is because glue is applied *outside* repetition, so even if the `url` element were repeated zero times (as allowed by the `minOccur` attribute) the `glued` would first match the repeated element (which would match some parentheses later in the document) and *then* check whether they occurred at the current location (which they wouldn't).

In other words: The choice (whether to actually match a URL or not) is made before the glue check is made. However, the glue is only useful if it is applied before the choice is made, so it can influence that choice.

Glue can be especially useful in conjunction with backtracking (described later in this document). See the `indent` example in the `demo` directory of the distribution for an example where (non-empty) glue is necessary to discriminate between quotes and definition list entries.

In previous versions of Atox, there was Boolean attribute called `glued`. This is no longer available, but it is equivalent to using `glue` with an empty glue pattern.

2.10 Backtracking

With the kind of left-to-right parsing used in Atox, you sometimes run into a very specific kind of trouble: You have two different things you want to parse, and they start out just the same. In other words, as Atox encounters the text feature, there is no way of telling which way to parse it. It's a bit like dealing with statements like "Ooh, how useful. Not." Unless you can hear the tone of voice, you don't know that the statement is actually a negative until you see the "not."

A simple example of this would be the previous glue example, where a piece of emphasized text can be turned into a link by putting a URL after it. (Glue is often useful in conjunction with backtracking.) What you would really want is to parse it as a link, not as emphasized text. But you don't know about the URL until you reach it...

There are several ways of dealing with this problem, most of which aren't easy to implement with Atox's special tokenization strategy. Backtracking is the simplest and most general strategy (although it can be quite inefficient). In Atox you use the `ax:try` tag to do backtracking. It will look for potential matches for its contents, using the standard search rules (basically looking for its first descendant in the text); however, when it finds a potential match it

won't commit to it completely, as the normal parsing strategy does. Instead, it will try to parse its contents at that position, but if it fails, it will keep looking.

Here is an example of how this works, again using the emphasis-as-link example (the following should be put inside an `ax:alt` tag):

```
<ax:try>
  <link>
    <ax:del>\*</ax:del>
    <ax:del>\*</ax:del>
    <url>
      <ax:del glue="">\s*\(</ax:del>
      <ax:del>\)</ax:del>
    </url>
  </link>
</ax:try>
<em>
  <ax:del>\*</ax:del>
  <ax:del>\*</ax:del>
</em>
```

Consider how the following is parsed:

```
*This* is emphasis. *This* (atox.sf.net) is a link.
```

When Atox starts looking for a `link` element, it first finds the first star (`\*`). It happily starts parsing but when it comes to the left parenthesis, there is none — at least not glued right after the star-enclosed text. Therefore this potential `link` occurrence fails. Normally, this would cause Atox to bail out, but because we have used an `ax:try` tag, it keeps looking. Luckily, the next star is luckier (no pun intended), and is used as the next `link` occurrence. So when Atox decides what do do at the beginning of this string, it starts parsing an `em` occurrence, because that is found right at the beginning.

Note that although backtracking is very powerful, it can be very inefficient if used wrong. If you rely on it all over the place, and nest several `ax:try` tags inside each other, so that Atox has to try out several alternatives at each level, the number of combinations to try out grows exponentially with the number of levels. Even for such simple applications as the previous example, it can slow things down a bit. For example, if there are lots of emphasized pieces of text but quite few links in your document, the backtracking approach will still try to parse all emphasis as links.

3 Enlisting XSLT

Atox has been designed from the beginning as a companion to XSLT — you use Atox to mark up a plain text file and some XSLT tool to convert the resulting XML file further. This workflow works well when the XML file is in some common interchange format and you have XSLT stylesheets that convert it to various output formats, such as XHTML, TEX, RTF, `xsl:fo`, or whatever.

But XSLT has many uses, and is a very powerful tool. So powerful, in fact, that some functionality was intentionally left out of Atox because you can easily use XSLT to achieve the same things. Consider, for example, the problem of getting attributes in your output. Atox won't let you do this — it will only let you mark up the attribute values as elements. For example, the following could be a marked-up link:

```
<link>The Python website<url>http://www.python.org</url></link>
```

This is as far as you get with the built-in Atox functionality. You could then use the following XSLT templates to transform the URL to an attribute:

```
<xsl:template match="link">
  <link url="{url}">
    <xsl:apply-templates/>
  </link>
</xsl:template>
<xsl:template match="link/url"/>
```

The first of these adds a `url` attribute consisting of the contents of the `url` element, while the second template silently matches (and eliminates) the `url` element.

It may seem odd to delegate such a basic operation to XSLT, but the point is that there are many similar "basic" operations (such as dealing with namespaces or automatic numbering or what-have-you). Instead of implementing many *ad hoc* pieces of functionality, it is better to let XSLT do the job, because it does the job so well. (Note that any XSLT element can be included, not just `xsl:template` elements. Another useful candidate is `xsl:output`, which can be used to control the details of the output format from your XSLT processor.)

In general, the rule is: Let Atox do the parsing and XSLT the transformations. (Some oft-used, simple transformations are available, though for convenience. One example is `ax:del`, which could also have been performed by XSLT.)

If you're going to do a specific transformation (such as adding attributes) regardless of output format, you'll need to apply *two* XSLT stylesheets to your file (or, worse, duplicate the functionality in all your output-specific stylesheets). This can be a bit impractical, and it could lead to you using only the Atox format file without the accompanying XSLT stylesheet, leading to an incorrect interchange format. Wouldn't it be neat if you could put XSLT templates right into your Atox format files? Guess what — you can.

Any top-level element (that is, child elements of `ax:format`) that use the the XSLT namespace (see the example below) will be filtered out of your format and collected into an "invisible" XSLT stylesheet. In addition to any templates you define there, a template which copies everything by default is added, so you only have to add templates for things you want to *change*. This stylesheet will be used automatically by Atox, using the option `xslt_command` as a shell-out. (In future versions Atox may also use Python XSLT libraries to avoid the overhead of using temprary files for the automatic XSLT processing.)

If you don't use any XSLT templates, no such processing occurs.

Here is an example stylesheet that some sloppy HTML with links:

```
<?xml version="1.0"?>
<ax:format xmlns:ax="http://hetland.org/atox"
           xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <html>
    <a ax:maxOccur="inf">
      <ax:del>\*</ax:del>
      <ax:del>\*</ax:del>
      <url>
        <ax:del>\s*\(</ax:del>
        <ax:pat>[a-zA-Z]+:\S+[a-zA-Z0-9/]</ax:pat>
        <ax:del>\)</ax:del>
      </url>
    </a>
  </html>
  <xsl:template match="a">
    <a href="{url}">
      <xsl:apply-templates/>
    </a>
  </xsl:template>
  <xsl:template match="a/url"/>
</ax:format>
```

It may seem a bit overwhelming (and *certainly* not beautiful) but it's not all that complex, really. When applied to the text

```
This is a *test* (http://test.com).
```

it will produce the following output:

```
<?xml version="1.0" encoding="utf-8"?>
<html>This is a <a href="http://test.com">test</a>.
</html>
```

4 Known Problems

Quite a bit of the code is still rather experimental, and could do with some refactoring for transparency and readability. If you find bugs that aren't mentioned (and you're using the most recent version), please report them to the Atox mailing list[24].

- There seem to be some problems with using `ax:try` recursively.
- There is quite a bit of redundant processing going on, so Atox is slower than it has to be. I will attack this in time, but, as the saying goes: *Make it*

[24] `mailto:atox-user@lists.sourceforge.net`

work, make it better, make it faster. It more or less works now, so... (There are many options for making it really fast, eventually, such as using C extension modules. Because Atox is a batch-mode program, this is not a top priority — nag me about it if you need more speed.)

5 Plans, To-Do List

The following are some possible features for future releases (in a rather arbitrary order):

- Add `ax:line`, `ax:sol`, and `ax:eol` (similar to the block elements, but for lines).
- Add an `ax:la` (lookahead) element for pattern-lookahead? (Just as as syntactic sugar, to avoid the need for `(?=...)`.)
- Perhaps rework the tokenizing strategy to avoid some unnecessary work. Encapsulating the discovery of occurrences of atomic parsers (tokens) might make it possible to use hashing in `ax:alt` tags, to speed up the selection process there considerably. It would require quite a bit of refactoring, though.
- Improve the command-line interface (with more switches and the like). Support searching for Atox formats in an `ATOXPATH` path variable, or something like it. Support automatic running of an XSLT processor, perhaps also with automatic searching for XSLT stylesheets? The distribution could then have some built-in Atox formats with corresponding XSLT schemas for various output formats, lowering the threshold for using Atox.
- Use the indentation features in the documentation.
- Extend the simple API (for using the XML format and for parsing input files) and document it.
- Make it possible to delete the whitespace of an indent.
- Add support for absolute indentation (e.g. "indented 2 spaces" or "indented more than 4 spaces")? Explicit relative indentation too? Simply setting absolute indentation greater than zero could be quite useful.
- Add an `ax:fill` attribute (with possible values `start`, `end`, and `both`) to make an element "fill out" the text segment available to it (up to the next/previous element).
- Add `ax:fail` and `ax:not` or the like, for controlling backtracking.
- Make `ax:ref` work on other (empty) tags than `ax:match`, as a shortcut that creates a single `ax:match` child element.
- Simplify and document the extension API, and allow references to custom parser elements in Atox format files.

- Create an element for inserting text into the constructed XML document (including group substitution from the previous pattern).
- Improve the error messages. Make Atox ignore elements that cause errors? Possibly with a command-line switch? (I.e. become more robust to malformed text files.)
- Support for escaping patterns. For example, it would be nice to be able to use dollar signs (possibly with an escape code) in the text even if dollar signs were used to indicate math mode (as in TEX).
- Possibly a different way of supporting top-level elements (that is, the document element). The current method is a bit odd if you use, for example, a pattern on its own. (Affects only the API, not the XML format.)
- Support for groups and named groups (from all previously matched regular expressions) in the output ... and the regular expressions? (The latter would require more refactoring to implement.)
- Better support for re-use, including using imported structures with renamed tags. This can include built-in support for (e.g.) paragraphs and symbols (with the corresponding escape mechanisms for the symbols).
- Command-line shortcuts, obviating the need for an XML format file, possibly using one of several default grammars (such as "any element may occur anywhere," or "the document consists of blocks, and all blocks except code blocks consists of inlines") with proper specification of the various elements as regular expressions.
- Parametrized use of definitions?
- Pie-in-the-sky: Redesign Atox as a set of small command-line tool filters that can be used to refine the mark-up in an XML file? This would be similar to the pipelines used with The Documenter's Workbench (*troff* and friends). A first step would be to make Atox XML-aware, so that it can be used several times in a pipeline, with different stylesheets.

6 Revision History

0.5.0: Added support for XSLT fragments in Atox format files. Added support for non-greedy repetition. Added several new options to the configuration system. Split input and output encoding and made UTF-8 the default output encoding.

0.4.0: Made the error handling slightly more user-friendly. Added some basic improvements to the command-line interface (the `-e`, `-f` and `-o` switches, as well the ability to use multiple input files or standard input). Note that the new calling convention is incompatible with the previous version, in that the format file is no longer supplied as an argument. Normalized newline-handling. Added the utility tags `ax:block`, `ax:sob` (start-of-block) and `ax:eob`

(end-of-block). Fixed an important bug in the indentation code, which affected `ax:indented`. Made empty sequences legal. Added support for config files.

0.3.0: Added some examples to the `demo` directory. Added the `ax:indented` tag. Refactored the codebase to remove the use of side-effects in parsing. Added backtracking in the form of the `ax:try` tag. Removed the `glued` attribute and added the (more powerful) `glue` attribute.

0.2.0: Added the `demo` directory with some examples. Added a test suite that uses the examples. Implemented the glue mechanism, for contiguous parsing. Added the `ax:indent` and `ax:dedent` tags, for more powerful parsing of indentation.

0.1.0: Completely rewritten, using a deterministic top-down parsing algorithm (related to LL(1), but with a markup-friendly tokenizing strategy). Redesigned XML format for format description. Simple (currently undocumented) API for parser construction. Changes from release candidate 1: Made latin 1 the default encoding, to allow non-ASCII characters in the input text. Fixed some problems with the `minOccur` and `maxOccur` attributes.

0.0.4: Unreleased prototype. Same architecture as 0.0.3 but with a new API, based on a decorator design. The architecture was a bit too general, and in some cases not very efficient. No XML for format description.

0.0.3: Unreleased prototype. Rewritten to use a metaclass-based API and a very general parser architecture. The API was quite convenient, but way too magical. No XML for format description.

0.0.2: Unreleased prototype. Completely rewritten using a generic pushdown automaton with regular expressions initiating state transitions. Efficient, but the API a bit unwieldy. No XML for format description.

0.0.1: Unreleased prototype. Used the Earley parsing algorithm, taken from Spark[25]. Was quite slow (mainly, I'm sure, because of my flawed design) and not quite user-friendly. An XML format was used for format description.

Early sketches: Various attempts at using hard-coded formats but with an easily extendable design. See, for example, chapter 14, "Instant Markup," of my book *Practical Python* (Apress, 2002).

7 License

[25] `http://pages.cpsc.ucalgary.ca/~aycock/spark`