A User's Guide to *txt2XML* v0.98

Brian Thomas

Astronomical Data Center
Goddard Space Flight Center/NASA

May 17, 2000

Contents

List of Figures

List of Tables

PART 1

Introduction

1.1. What is *txt2XML*?

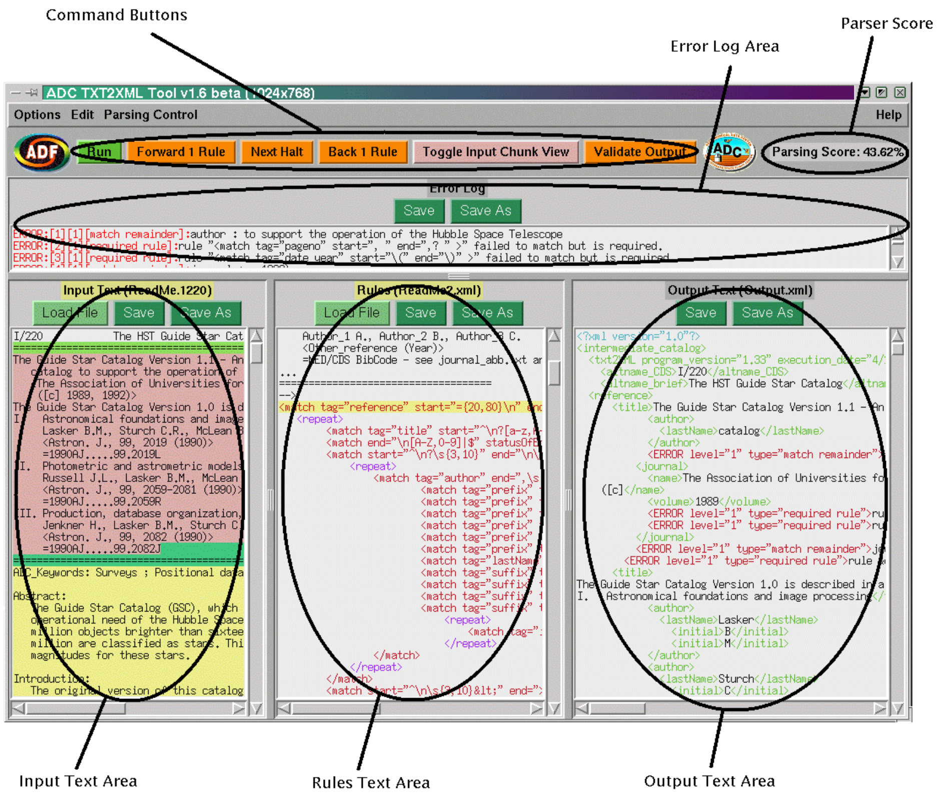
txt2XML is a *Perl*/Tk program designed to transform semi-formatted text files into XML according to a set of user-specified rules. By semi-formatted we mean recognizable character patterns that mark the start and end of the text that is to be tagged.

txt2XML uses *Perl*'s powerful regular expression pattern matching to identify the interesting text. The rules (which are written in XML) can be nested to limit the scope of the pattern matching and include constructs to allow looping and control flow similar to a programming language. The result of applying these rules to an input document is a well-formed XML document that contains some or all of the text of the original input file, parsed into meaningful XML elements.

This program has been tested on both *UNIX* and Microsoft *Windows*^{tm} 95/98 (*W9x*) computers but actually has a fair chance of running on any computer that has *Perl* installed and a C compiler (C compiler NOT needed for *Windows* versions to run; see section 1.3 for software requirements).

txt2XML may be run as a command line call or with its graphical user interface (GUI, figure 1.1). The GUI features a WYSIWYG interface designed to ease the writing and debugging of parsing rules. In the graphical mode the user may interactively run, debug, and edit the input or rule files, and then re-run the parser any number of times. Keyboard and mouse shortcuts allow the user to jump to any point in the parsing run to examine exactly what chunk of text was fed to a rule, and how the rule reacted to it. From the GUI, the user may save/load new output text, input and rules text at any point.

Figure 1.1: *txt2XML* in action.



```
ERROR:[1][1][match remainder]:author : to support the operation of the Hubble Space Telescope
ERROR:[2][1][required rule]:rule "<match tag="pageno" start=", " end=",? " >" failed to match but is required.
ERROR:[3][1][required rule]:rule "<match tag="date year" start="\(" end="\)" >" failed to match but is required
```

```
I/220                The HST Guide Star Cat
==========================================
The Guide Star Catalog Version 1.1 - An
   catalog to support the operation of
   <The Association of Universities for
    ([c] 1989, 1992)>
The Guide Star Catalog Version 1.0 is d
I.    Astronomical foundations and image
      Lasker B.M., Sturch C.R., McLean B
      <Astron. J., 99, 2019 (1990)>
      =1990AJ.....99.2019L
II.   Photometric and astrometric models
      Russell J.L., Lasker B.M., McLean
      <Astron. J., 99, 2059-2081 (1990)>
      =1990AJ.....99.2059R
III. Production, database organization,
      Jenkner H., Lasker B.M., Sturch C.
      <Astron. J., 99, 2082 (1990)>
      =1990AJ.....99.2082J
==========================================
ADC_Keywords: Surveys ; Positional data

Abstract:
   The Guide Star Catalog (GSC), which
   operational need of the Hubble Space
   million objects brighter than sixtee
   million are classified as stars. Thi
   magnitudes for these stars.

Introduction:
   The original version of this catalog
```

```
    Author_1 A., Author_2 B., Author_3 C.
    <Other_reference (Year)>
    =MED/CDS BibCode - see journal_abb.txt ar
...
=================================
-->
<match tag="reference" start="={20,80}\n" end
   <repeat>
      <match tag="title" start="^\n?[a-z,A-
      <match end="\n[A-Z,0-9]|$" statusOfE
      <match start="^\n?\s{3,10}" end="\n\
         <repeat>
            <match tag="author" end=",\s
                  <match tag="prefix"
                  <match tag="prefix"
                  <match tag="prefix"
                  <match tag="prefix"
                  <match tag="prefix"
                  <match tag="prefix"
                  <match tag="lastName"
                  <match tag="suffix"
                  <match tag="suffix"
                  <match tag="suffix"
                  <match tag="suffix"
                     <repeat>
                        <match tag=":
                     </repeat>
            </match>
         </repeat>
      </match>
      <match start="^\n\s{3,10}&lt;" end=">
```

```
<?xml version="1.0"?>
<intermediate_catalog>
  <txt2XML program_version="1.33" execution_date="4/
    <altname_CDS>I/220</altname_CDS>
    <altname_brief>The HST Guide Star Catalog</altnam
  <reference>
    <title>The Guide Star Catalog Version 1.1 - An
          <author>
            <lastName>catalog</lastName>
          </author>
          <ERROR level="1" type="match remainder">
        <journal>
          <name>The Association of Universities fo
    ([c]</name>
          <volume>1989</volume>
          <ERROR level="1" type="required rule">ru
          <ERROR level="1" type="required rule">ru
        </journal>
        <ERROR level="1" type="match remainder">jo
      <ERROR level="1" type="required rule">rule
    <title>
The Guide Star Catalog Version 1.0 is described in a
I.   Astronomical foundations and image processing</
          <author>
            <lastName>Lasker</lastName>
              <initial>B</initial>
              <initial>M</initial>
          </author>
          <author>
            <lastName>Sturch</lastName>
              <initial>C</initial>
```

In this figure *txt2XML* is shown running in the sliding frames mode (the default display mode when the *Perl Tk::SplitFrame* module is installed).

1.2. Where and When to Use *txt2XML*

Like any piece of software, *txt2XML* has its proper operational space. *txt2XML* is designed to be useful for conversion of *many* ASCII documents which share a similar, but not rigidly common, format. *txt2XML* can do an excellent job of *tagging* ASCII text but definitely does a terrible job of *arranging* the tagged text into the format you desire. *txt2XML* will give you a consistent output format, but tags in the output document are rarely arranged/nested in the manner that you may desire (i.e. to the DTD).

With a lot of effort you can make *txt2XML* create documents with an acceptable output format[1], however, this is tantamount to eating Chinese food with a pair of screwdrivers. Its just not a rational thing to do. The arrangement/re-nesting of XML nodes is really better left to software like XSLT so, in general, you will want to have a two-step process for converting ASCII text to your specified DTD:

1) Capture tagged content (using *txt2XML*)

2) Rearrange tagged content to meet your DTD (using XSLT or the like).

1.3. Installing *txt2XML*

1.4. Software Requirements

You will first need a copy of the *txt2XML* distribution archive. These are obtainable from our anonymous FTP server `xml.gsfc.nasa.gov` in `/pub/XML/txt2XML` or off of the WWW from our page at `http://adc.gsfc.nasa.gov/adc/adc_software.html`.

You will need an installation of *Perl*, version 5.00503 or better. On many *UNIX* systems *Perl* is probably already installed. You can check its version using:

```
> perl -v
```

[1]In fact, its quite complicated to do even the simplest rearrangement of tags with *txt2XML*, so we won't bother to describe how.

Table 1.1: *Perl* Modules Required to Run *txt2XML*.

Perl	Required	Associated Files by FTP site	
Module	Version	CPAN (*UNIX*/*W9x*)[1]	*ActiveState* (*W9x*)[2]
XML::Parser	2.27+	`XML-Parser-2.27.tar.gz`	Included in *ActivePerl*
XML::DOM	1.25+	`XML-DOM-1.25.tar.gz`	`XML-DOM.zip`
Tk	8.00.013+	`Tk800.013.tar.gz`	`Tk.zip`
Tk::SplitFrame[3]	0.01+	`Tk-DKW-0.01.tar.gz`	—
	1.04+	`Tk-GBARR-1.0401.tar.gz`	
	0.07+	`Tk-Contrib-0.07.tar.gz`	

[1]Location: `ftp://ftp.cpan.org/pub/CPAN/modules/by-module`
[2]Location: `http://www.activestate.com/packages/zips`
[3]Module is optional. Can be difficult to install on *W9x* platforms.

or from within a MS-DOS command shell:

```
> perl.exe -V
```

If you find that you need to upgrade *Perl* you may obtain a copy from `ftp.cpan.org`. RedHat Linux users may wish to just obtain the binary rpm from `ftp.redhat.com` (don't forget to check the signature of the rpm file!). *Windows* users can obtain a free copy of *ActivePerl* from *ActiveState* at `http://www.activestate.com/` (obtain build 521 or better).

Depending on the distribution of *Perl* you installed, you will need to obtain and install the *Perl* modules in table 1.1: *Tk::SplitFrame* is an optional module; if it is installed the user can select to display the tool using a sliding frames mode (this is the mode used in figure 1.1). Otherwise, *txt2XML* will only display with all of the text areas split into separate windows.

For *Windows* users it is definitely far easier to go the *ActivePerl* route and install the PPD (module) files they provide. You can obtain the *ActiveState* PPD files (archived in zip format) from `www.activestate.com/packages/zips`.

To use the *CPAN* modules (archived as *gzip*'d tarballs) *UNIX* and *Windows* users will need a decent C/C++ compiler (*gcc* 2.8.1 works for us) to build the *XML::Parser* module, and a compatible *make* utility (*NMAKE* works for us on *W9x* platforms). CPAN modules are available at `ftp.cpan.org/pub/CPAN/modules/by-module`.

1.4.1. Installation on *UNIX* systems

You will first need to insure that you have the correct version of *Perl* and all of the needed *Perl* modules installed.

To install a *Perl* module, unpack the tarball:

```
> gunzip -c file.tar.gz | tar xvf -
```

Change into the new sub-directory and then type:

```
> perl Makefile.PL; make ; make test; make install
```

You will need to do this for each module in turn. Installation of the modules will go easier if you install them in the order:

```
XML-Parser-2.27.tar.gz
XML-DOM-1.25.tar.gz
Tk800.013.tar.gz
Tk-GBARR-1.0401.tar.gz
Tk-Contrib-0.07.tar.gz
Tk-DKW-0.01.tar.gz
```

Now unpack the *txt2XML* tarball `txt2XML_latest.tar.gz`:

```
> gunzip -c txt2XML_latest.tar.gz | tar xvf -
```

You should be able to use *txt2XML* right from the new sub-directory. There is a simple install script, *install.sh*, which you may use to install *txt2XML* in a favorite directory. If you can get super-user access (if you don't know what this is, you don't have it) you can install the program in a system directory using the install script.

Finally, a quick troubleshooting note: the path to *Perl* may not be set properly on your machine and this can cause the program to fail to execute. Type this to determine the location of *Perl*:

```
> which perl
```

or, if you have a Linux/GNU system:

```
> locate perl | grep bin
```

or

```
> slocate perl | grep bin
```

IF *Perl* is in a different spot than /usr/local/bin/perl you will have to edit the first line in the `txt2XML.pl` file.

1.4.2. Installation for users with *W9x* platforms

Installation on *Windows* platforms is even easier than for *UNIX*, however, just as in the *UNIX* install you will first need to insure that you have the correct version of *Perl* and all of the needed *Perl* modules installed. If you need to install/upgrade *Perl*, download the zip file from the *ActiveState* web site `http://www.activestate.com/` and run the installer. The modules are similarly easy to install. Download the zip files, unpack them, and type (within MS-DOS shell):

```
> ppm install XML-DOM.ppd
```

to install the XML-DOM module. Just substitute the appropriate PPD file name to install the other modules.

Now unpack the latest *txt2XML* zip file. You can now invoke the program from the spot you unpacked it. If you don't want to have to type the full path to the *Perl* executable you can check the PATH variable for your account. Make sure it includes the directory that perl.exe resides in (use the *Find Utility* in *Windows* to locate the *Perl* executable).

1.5. License

Please read the License file which is included in the archive distribution.

[**Will** *txt2XML* **be released as open software? Is GPL or BSD license appropriate? Does NASA have its own license for software?**]

1.6. Support/Contact/Bug Reports

There is no official user support for *txt2XML*. We are, however, sympathetic to the first time user who has failed to get the program running even after having *read the documentation*. Please send requests for help to:

adc@adc.gsfc.nasa.gov

Remember we are busy on another project or two, so our response may take some time.

Comments on how we may work to improve this program, its documentation and bug reports (especially patches!) are greatly appreciated. Please send this email to:

brian.thomas.1@gsfc.nasa.gov

1.7. Credits

Concept:	Ed Shaya, Brian Thomas, Brian Holmes.
Parser Design:	Brian Thomas, Ed Shaya, and Jim Blackwell.
GUI Design:	Brian Thomas, Ed Shaya and Brian Holmes.
Code Written by:	Brian Thomas and Ed Shaya.

PART 2

Getting Started Using *txt2XML*

2.1. Parsing Philosophy

txt2XML does not think in terms of lexical structures (e.g. lines, paragraphs, pages, etc.), instead, the document is conceived of as a "chunk" of text which may be subdivided into a number of sub-chunks (which may themselves be further subdivided, and so on). Any amount of text may be a chunk (1 or more characters) and *txt2XML* just views a chunk as comprising a string of characters (which may be interspersed with formatting characters like \n and \r).

txt2XML proceeds to execute each parsing rule starting from the top of the rules document. Traversal of the rules can involve looping operations which means that the parser can execute any particular rule more than once. Therefore we call each execution of a parser rule a "rule event". Each rule event will have with it an associated portion of the input text.

The "working chunk" is that portion of the text which is under consideration by the "current rule event". A parsing rule may either divide the current "working chunk" and pass it on to child rules, or work with the current working chunk by matching part or all of it. The matched part of the text is refered to as "Match Text" (see section 3.4) and the rules can specify that the Match Text either be ignored or inserted in the output document within a specified tag. After use, the Match Text is subtracted from the working chunk. If any text remains, it becomes the new working chunk which is passed to a sibling rule. If no siblings exist then, under normal circumstances, that portion of the text chunk is tagged with an error tag in the output XML document.

There are different levels of errors in *txt2XML* and these are presented in order of "seriousness" in table 2.1. It can sometimes be helpful to run a tight ship and

Table 2.1: Defined Error Levels in *txt2XML*.

Error Level	Error Name	Description	Example Error
0	Trivial	Unlikely to be missed information.	Unmatched Whitespace
1	Normal	Likely to be missed information.	Unmatched Character String
2	Serious	Missing/malformed Parser rule.	non-tag <match> rule missing child rules.
99	Don't Halt	No errors will stop this run.	—

catch all errors when you are trying to debug your rules. On a production run, where input file formats may vary from file to file, upping the error threshold is more useful so you can get the job finished and treat the errors for each file individually later.

2.2. Running *txt2XML* from the Command Line

If you have already developed the rules you need, then most of the time you don't want the GUI. Instead, you'll want to quietly parse a pile of files without any of the fireworks. Command line mode is for you.

To run the program in command line mode depends on the system that you are running *txt2XML* on:

From a *UNIX* shell type:

```
> txt2XML.pl -f <rules_file> <file_to_be_converted>
```

and from a MSDOS command window:

```
> perl.exe txt2XML.pl -f <rules_file> <file_to_be_converted>
```

2.3. Running the Program in GUI Mode

The GUI mode presents the user with an interactive interface with which to see the operation of the parser as it goes through the input text. This mode is used to help the user write and debug their rules. There are many short cut mouse and key bindings that will help to accelerate the work of the user (see tables A.3 and A.2 for a full listing).

To fire up *txt2XML* in the GUI mode, it depends on the platform you are using it on:

From a *UNIX* shell type:

```
> txt2XML.pl -g
```

From *Windows* the easiest way to run in GUI mode is to click on the icon called "gtxt2XML". Optionally you can open up an MSDOS command window, switch to the directory in which the `txt2XML.pl` file resides and type:

```
> perl.exe txt2XML.pl -g
```

2.3.1. GUI Orientation

Now that you have the GUI up and running you will notice that there are several text areas and a menu/command button bar (refer to figure 1.1). We'll give a quick orientation on each of these below, you may wish to jump over this section to the short tutorial which follows it (section 2.3.2).

Command Buttons

The command buttons available are the "Run Button", "Forward 1 Rule Button ", "Back 1 Rule Button ", "Next Halt Button", "Toggle Input Chunk View Button", and "Validate Output Button". There is a "Parsing Score" display on the far right of the command button bar. The score is defined as:

$$score = \frac{\text{Numb. of characters text that were successfully parsed}}{\text{Numb. of characters in the input text}} \tag{2.1}$$

Pressing the Run Button causes the application to re-run the parser on the text which is currently in the Input Text Area using the parsing rules which are currently in the Rules Text Area. Output from the parser run is then shown in the Output Text Area.

Pressing the Forward 1 Rule Button will make the application advance forward 1 rule if it can. You will see the current rule highlighted in yellow in the Rules Text Area shift down 1 rule and the input chunk highlighted in yellow mark the chunk of text that was fed into that rule.

Pressing the Back 1 Rule Button causes the application to return to the previous rule, if one exists. As for the Forward 1 Rule Button the highlighting in the Input Text Area will change to indicate the working chunk for the current rule.

Pressing the Toggle Input Chunk View Button will toggle the colorization of the working chunk. The default is to colorize the working chunk with a single color: yellow. In this mode the color of the button will be yellow. The other mode is to colorize the Start Match, Match Text, and End Match portions of the working chunk (see section 3.4 for a definition). The button will appear pink in this mode.

The Validate Output Button can be used to check if the output text conforms to the XML specification (ie. check to see that the output is well formed XML document). In the future, it may be possible to validate the output document against a specified DTD, but this currently is not provided, and given that *txt2XML* is unlikely to be used to create the final XML formatted document, probably not needed. IF a given output document is found to be *not* well formed XML, the first node at which the document is found non-conforming will be colorized bright green.

Options Menu

The Options menu contains the following options: "Change Tool Display Size", "Change Text Font Size", "Change Text Fg/Bg Color", and the "Quit" entries.

Selection of the Change Tool Display Size entry will allow the user to pick from a limited menu of other display sizes. Numbers in parenthesis next to the entries are guides to the user of the general screen size in pixels of the given display. This selection will change the point size of the font too.

Selection Change Text Font Size entry will allow the user to change the point size of the font of the tool without changing the overall display.

Selection of the Change Text Fg/Bg Color entry allows the user to change the display colors of the text areas within the tool.

Selection of the Quit entry exits the program. You will be NOT asked if you want to save any edited files (you have been warned!).

Edit Menu

The Edit menu contains the following options: "Allow Edit Input Text" and "Allow Edit Output Text". By selecting "Yes" for either option you can enable editing of that text area. When editing is enabled in a text area, the name of the file will be shown with a yellow color. Uneditable files have a dull gray background color.

Parsing Control Menu

The Parsing Control menu contains the following options: "Halt Parser on Error Level", "INCLUDE Tagged Errors in Output XML Text", "Entify Chars in Output XML Text", and the "Allow Null Matches in Output XML Text" entries.

Selection of the Halt Parser on Error Level allows the user to change the level of error (or higher) that will stop the program. If "normal" error level is selected then "trivial" errors will be noted, but passed by while any "normal" or "serious" error the parser encounters will halt the program. Setting to "Don't Halt" prevents the program from halting on any error once it starts its parsing run. Refer to table 2.1 to see the defined error levels.

Selection of the INCLUDE Tagged Errors in Output XML Text entry allows the user to select whether or not errors encountered in the parsing run will show up in the output text (the errors are tagged with the string "ERROR"). Regardless of this setting all errors still show up in the Error Log Area.

Selection of the Entify Chars in Output XML Text changes whether or not characters within tags in the output text are entified. There are a number of reasons for toggling the entification. We have found cases where either turning entification on/off improves readability of the output text.

Selection of the Allow Null Matches in Output XML Text entry is a special use option. It allows one to tag 'empty' text. A case of this is when one might want to record the existence of an empty tag in SGML. Normally, it is safe (and desirable) to set this option to "No".

Help Menu

The Help menu contains the following options: "About", "Parsing Text", and the "Known Bugs" entries. Selecting each of these entries will bring up a short help message on that topic. The Known Bugs is notoriously out of date. If you find a bug, please be sure to let us know about it![1];

Error Log Area

The Error Log Area is where all of the accumulated errors from a parsing run are displayed. Every time the Run Button is pressed this area is cleared out for a fresh batch of errors (fun!).

Error messages are only loosely formated in the log. A typical message would appear as:

```
ERROR:[0][1][remainder]: A chunk of text that errored out.
```

where the first bracketed number records the order in which the error occurred, the second bracketed number provides the numerical representation of the error level (see table 2.1), the last bracket encloses the name of the error and the remaining text is the chunk of text which was errored out.

You can find use the Error Log Area to quickly navigate to the problem spots within your rules. Click on any given error entry in the Error Log Area with the lefthand mouse button. Doing so resets the Rules Text Area to display the rule at which the error occurred and highlights the chunk of text in the Input Text Area.

[1]send a bug report to *brian.thomas.1@gsfc.nasa.gov* please!

Input Text Area

The input ASCII text is shown here. The text shown will be highlighted depending on the setting of the Toggle Input Chunk View Button. In the default highlighting scheme the working chunk appears highlighted in yellow and the rest of the text plain. In the second mode, the working chunk highlighting is more complex: the Start Match, Match Text, and End Match portions of the working chunk are all colored differently.

You can edit the text in the Input Text Area by setting the Edit menu option Allow Edit Input Text to "Yes" (the name of the file will be shown with a yellow background). The editing process is simple, position the mouse over a section of text and click the left mouse button to relocate the text cursor. Alternatively, you can use the arrow keys to move the cursor. Now, just type in new characters on the keyboard to add text. You can "swipe" regions of text with the mouse by holding down the left mouse button. Swiped text will appear with a grey background. If you hit the `backspace` or `delete` keys the swiped text will be deleted from the Input Text Area. Now, you can relocate text by swiping it first then clicking the middle (*UNIX*) or right (*W9x*) mouse buttons. Swiped text will be duplicated starting from the spot over which you have positioned the mouse. Unfortunately, there is no buffer history. One day we'll replace this simple editor with something more heavy-weight.

You can set a break point by positioning the mouse over the point in the input text you wish to stop feeding the parser text and pressing the <`Ctrl`> key and then left-clicking the mouse. The breakpoint character will appear as red on a blue cell. To clear this breakpoint, press <`Alt`>−`c`.

There are three green buttons: "Load File Button", "Save Button", and "Save As Button" which appear at the top of the Input Text Area. Use these buttons to bring in new input files, overwrite your edited text to a file (defaults to the name of the original file), or save your buffer to a new file.

Rules Text Area

The Rules Text Area displays the parsing rules which are an XML document. The various rules within body of the document are colorized to ease reading of the rules. The "current rule" is displayed in this area and it will be highlighted in yellow. At the end of a parsing run, the *last* rule successfully parsed with will be the current rule. You can advance/go back to the next/prior rule by clicking on the Forward 1 Rule Button / Back 1 Rule Button or by using the `<Alt>−f`/`<Alt>−b` key presses.

You can hop to any rule of interest that was successfully parsed by pressing down the `<Ctrl>` key and left clicking the mouse on the rule. The current rule will become the rule you clicked on and will be highlighted in orange. Note that if the new current rule contains children, than for this particular function both the leading element and the closing element will be highlighted to improve readability.

The Rules Text Area has all of the same editing abilities as the Input Text Area. When you type in new rules the text will be normal. When you run the parser, if the newly added rules are valid XML, they will be colorized. IF the edit you made to the rules does not result in a well-formed XML document, the first rule at which the document becomes non-conforming will be colorized bright green. Usually this means that the error was made in a preceding nearby rule. Note that the processor will not run until you have a well-formed XML rules document. The content and use of the rules are discussed in section 3.1.

As for the Input Text Area there are three green buttons at the top of the widget: "Load File Button", "Save Button", and "Save As Button". These have the same meanings and abilities.

Output Text Area

The Output Text Area displays output XML text in a colorized format. Elements are colored green and error elements are colored red. Tagged text is left normal.

The Output Text Area is (by default) not editable and only has two of the three green file function buttons that the Input Text Area and Rules Text Area have. You can save / save as your output file. As for the other two widgets, the "Save Button" and "Save As Button" buttons have the same meanings. To make the Output Text Area editable, set the Edit menu option Allow Edit Output Text to "Yes" (the name of the output file will be shown with a yellow background then).

2.3.2. Short Tutorial Using the GUI

Alright, lets take this baby out for a spin, shall we? In this part we'll focus on some selected features of the GUI and how they may be used to debug an existing set of rules. We'll discuss some example problems in our test rule set but, for now, we'll save the greater details on how the rules function for part 3.

As netzitens, many of us belong to email lists. All sorts of useful information is traded on these lists, but often the site that serves the list does not provide any archival searching. Lets examine how we can XML'ize some email from a listserver that some of you may be familiar with (the *ActiveState* Perl–XML mailing list) so that it can be made part of an XML database we want to host[2].

You should have several example files which where packed within the tarball distribution. Change to the directory where these reside and then fire up the GUI with the sample input ASCII text file `sample.txt`, and its accompanying sample rules `sample_rules.xml` (the content of these files is shown in appendix E). To do all of this without having to resort to loading these files after the GUI comes up, type the following:

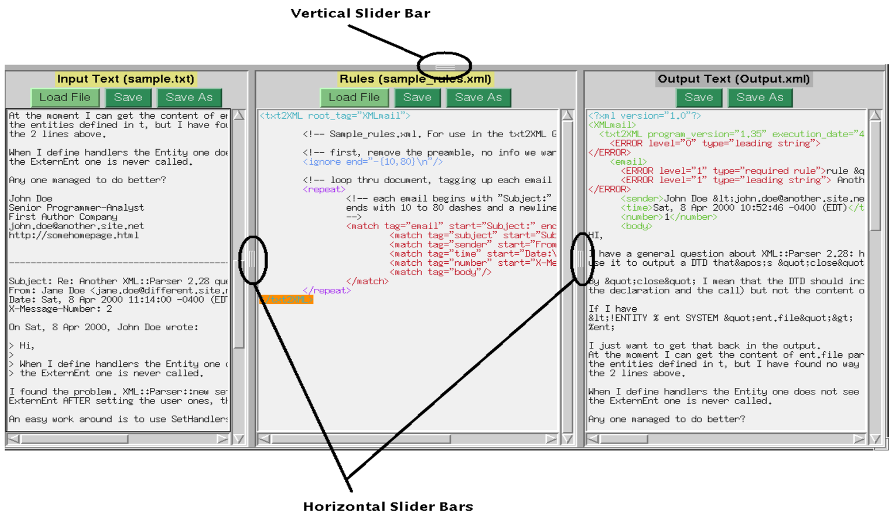
```
> txt2XML.pl -g -f sample_rules.xml sample.txt  (UNIX)
```

or

```
> perl.exe txt2XML.pl -g -f sample_rules.xml sample.txt (W9x )
```

[2]Yes, I know the *ActiveState* people provide archival searching capability, so for *this* mailing list this is a redundant exercise. Its just a tutorial, Ok?

Figure 2.1: *txt2XML* GUI tutorial text areas



```
At the moment I can get the content of en
the entities defined in t, but I have fou
the 2 lines above.

When I define handlers the Entity one doe
the ExternEnt one is never called.

Any one managed to do better?

John Doe
Senior Programmer-Analyst
First Author Company
john.doe@another.site.net
http://somehomepage.html


----------------------------------------

Subject: Re: Another XML::Parser 2.28 que
From: Jane Doe <jane.doe@different.site.n
Date: Sat, 8 Apr 2000 11:14:00 -0400 (EDT
X-Message-Number: 2

On Sat, 8 Apr 2000, John Doe wrote:

> Hi,
>
> When I define handlers the Entity one d
> the ExternEnt one is never called.

I found the problem. XML::Parser::new se
ExternEnt AFTER setting the user ones, th

An easy work around is to use SetHandlers
```

```
<txt2XML root_tag="XMLmail">

        <!-- Sample_rules.xml. For use in the txt2XML G

        <!-- first, remove the preamble, no info we war
        <ignore end="-{10,80}\n"/>

        <!-- loop thru document, tagging up each email
        <repeat>
                <!-- each email begins with "Subject:"
                ends with 10 to 80 dashes and a newline
                -->
                <match tag="email" start="Subject:" enc
                        <match tag="subject" start="Sub
                        <match tag="sender" start="From
                        <match tag="time" start="Date:\
                        <match tag="number" start="X-Me
                        <match tag="body"/>
                </match>
        </repeat>
</txt2XML>
```

```
<?xml version="1.0"?>
<XMLmail>
  <txt2XML program_version="1.35" execution_date="4
    <ERROR level="0" type="leading string">
</ERROR>
    <email>
      <ERROR level="1" type="required rule">rule &q
      <ERROR level="1" type="leading string"> Anoth
</ERROR>
      <sender>John Doe &lt;john.doe@another.site.ne
      <time>Sat, 8 Apr 2000 10:52:46 -0400 (EDT)</t
      <number>1</number>
      <body>
HI,

I have a general question about XML::Parser 2.28: h
use it to output a DTD that&apos;s &quot;close&quot

By &quot;close&quot; I mean that the DTD should inc
the declaration and the call) but not the content o

If I have
&lt;!ENTITY % ent SYSTEM &quot;ent.file&quot;&gt;
%ent;

I just want to get that back in the output.
At the moment I can get the content of ent.file par
the entities defined in t, but I have found no way
the 2 lines above.

When I define handlers the Entity one does not see
the ExternEnt one is never called.

Any one managed to do better?
```

In this figure the text areas of *txt2XML* are shown after the GUI comes up.

Ajustment of GUI Appearance

You should see the GUI come up after a brief wait (don't forget to use the '-g' switch!!). As a first step, let's get comfortable adjusting aspects of the GUI appearance. Figure 2.1 shows how the text areas within the GUI should look and highlights the slider bars which can be used to resize the various text areas within the GUI as desired. The overall size of the GUI can be adjusted by pulling on its edges.

If you don't like the size of the font (either too small or too large), go to the Options Menu and select a new size font from the Change Text Font Size submenu. In the Options Menu you'll notice other changes you can make to the GUI apperance.

Basic Interface and Controls

There is some text in all 3 text areas as well as in the Error Log Area. The output text is populated because the GUI will run the parser (when it is first invoked) IF you specify both an input and rules file on the command line (which you were instructed to do!). To run the parser the next time, just press the Run Button or `<Alt>-p`.

At the top of the GUI (figure 1.1) you can see the command buttons arrayed horizontally. Press the Validate Output Button; doing so will cause *txt2XML* to check the well-formedness (but not strong validation) of the output document. One of the tags in the output document is highlighted with a bright green color:

```
<ERROR level="1" type="remainder">
```

This is because some of the error statements that *txt2XML* can insert into the output document will violate the XML standard. Lets turn these error statements off and try again to see how well we do. Turning off errors in the output is easy, go to the Parsing Control Menu and select “No” from the INCLUDE Tagged Errors in Output XML Text. Now run the parser then press the Validate Output Button again. You should now see that there is no bright green color highlighting any of the tags in the output text (thereby indicating the output document is well-formed).

You will notice that the tags within the Rules Text Area and Output Text Area are colorized to improve readability. The need to follow the XML standard requires that we entify certain characters and this can hurt the readability of the tagged output text. Lets turn off the entification by selecting “No” from the Entify Chars in Output XML Text submenu within the Parsing Control Menu. Run the parser again; now we see that some of the tagged content gets colorized but is possibily more readable. Not suprisingly, if you check you will find the output document is not well formed. Select “Yes” from the Entify Chars in Output XML Text submenu within the Parsing Control Menu and rerun the parser to return the output text back to its original state.

The most important information the GUI can provide is in the identification of what portion of the input text (or current "working chunk") is being seen by a particular rule at any one time. This information may be gleaned from the Input Text Area and Rules Text Area. You might have already noticed that there will sometimes be background highlighting of text in both text areas. In the Input Text Area the text that is highlighted (yellow) corresponds to the current working chunk. In the Rules Text Area the text that is highlighted (yellow or orange) corresponds to the current rule event. You can change the current rule event by either selecting a particular rule using the mouse (place the pointer over the rule, hold down the `<Ctrl>` key and left click) or by pressing the Forward 1 Rule Button or Back 1 Rule Button . Try hopping to a rule using the mouse. Select the following rule with the `<Ctrl>` key held down and left click:

```
<match tag="time" start="Date:\s?" end="\n"
statusOfStart="accept" />
```

You should see the working chunk change to encompass the following text:

```
Date: Sat, 8 Apr 2000 10:52:46 -0400 (EDT)
X-Message-Number: 1

HI,

I have a general question about XML::Parser 2.28: has anyone managed to
use it to output a DTD that's "close" to the one in the input document?

By "close" I mean that the DTD should include the external entities (both
the declaration and the call) but not the content of those entities.

If I have
<!ENTITY % ent SYSTEM "ent.file">
%ent;

I just want to get that back in the output.
At the moment I can get the content of ent.file parsed and I can output
the entities defined in t, but I have found no way to get ANYTHING about
the 2 lines above.

When I define handlers the Entity one does not see the first line and
the ExternEnt one is never called.

Any one managed to do better?
```

John Doe
Senior Programmer-Analyst
First Author Company
john.doe@another.site.net
http://somehomepage.html

Now click on the Forward 1 Rule Button. You should see the working chunk change slightly. The first line will disappear (this is because the last rule used that part of the chunk). Now click on the Back 1 Rule Button twice. You should see the working chunk change slightly again. This time lines were added at the beginning of the working chunk, showing the content being 'undigested' by the rules as you go backward through the rule events. Notice that when you use the mouse to select the current rule event the color is different from when you use the buttons; orange when you use the mouse and yellow when you use the buttons. This difference is to indicate the slight difference in meaning between the two actions. Clicking with the mouse will highlight the whole node of the rule (the opening and ending elements) whereas the button navigation will only highlight the opening node of the rule. To illustrate this compare the difference in appearance between clicking on these rule nodes:

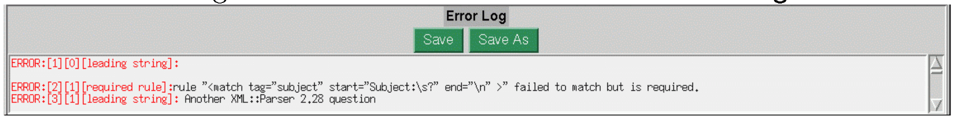
```
<ignore end="-{10,80}\n"/>
```

and

```
<match tag="email" start="Subject:" end="-{10,80}\n"
statusOfStart="accept" >
```

There is more information that can be gleaned about the way in which the rule is using the working chunk. To see how the working chunk was 'digested' by the rule event click on the Toggle Input Chunk View Button. This will now change the highlighting of the working chunk in the Input Text Area to show what portions of the working chunk correspond to the Start Match, End Match, and Match Text portions (see section 3.4 for definition of these concepts).

Figure 2.2: *txt2XML* GUI tutorial Error Log Area



```
ERROR:[1][0][leading string]:

ERROR:[2][1][required rule]:rule "<match tag="subject" start="Subject:\s?" end="\n" >" failed to match but is required.
ERROR:[3][1][leading string]: Another XML::Parser 2.28 question
```

Figure 2.2. In this figure the Error Log Area of *txt2XML* are shown after the GUI comes up.

We have left one of the most useful aspects of the GUI interface for last. You can use the errors shown in the Error Log Area to hop to problem spots in your rules. Figure 2.2 shows the Error Log Area for this tutorial. Take the mouse and left click on the first error, you will see the working chunk and current rule change to reflect the point where the error occured.

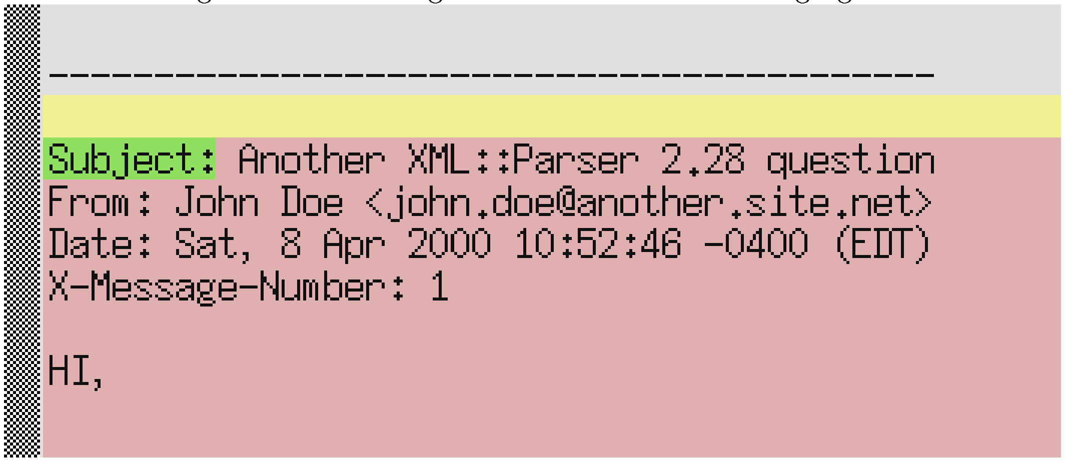
Fixing Errors and Editing Rules

Now that we have all of the basic pieces in place lets try to understand the tutorial sample rules and text errors, fixing them as we go. As you may have already deduced, the `match` rules specify starting and ending points at which text is grabbed from the working chunk and tagged in the output document. The way we specify these starting and ending points is through *Perl* regular expressions. In this tutorial you don't need to know a whole lot about *Perl* regular expressions or the inner workings of the rules Don't worry too much about the reasons for the edits we will later make (below) in the rules set. Remember that our goal in this section is to see how the various GUI features work together to help us identify problems in the rules. You can follow up this section with a read of part 3 and/or appendix B to bring yourself up to speed on the inner workings of the rules.

Looking at our GUI afresh, we note that the Parsing Score tells us how well we did in tagging the content of the input document. In our case the Parsing Score is 68.28%, a fairly mediocre score. Lets try to do better by understanding the errors. Click on the first error, e.g.

ERROR:[1][0][leading string]:

Figure 2.3: Working Chunk of the Rule Belonging to Error



```
------------------------------------------

Subject: Another XML::Parser 2.28 question
From: John Doe <john.doe@another.site.net>
Date: Sat, 8 Apr 2000 10:52:46 -0400 (EDT)
X-Message-Number: 1

HI,
```

Figure 2.3. Colors indicate Start Match (light green) and Match Text (pink) portions of the working chunk (yellow).

Doing so will highlight the rule:

```
<match tag="email" start="Subject:" end="-{10,80}\n"
statusOfStart="accept" >
```

and will highlight part of the text in the Input Text Area. If the Toggle Input Chunk View Button is not pink click on it once to colorize the working chunk. We now see that the begining part of the input text (the working chunk for the rule indicated) looks like that shown in figure 2.3; it has a yellow band followed by a light green portion (indicating the Start Match) and a longer pink portion (indicating the Match Text). The yellow band indicates that the leading portion of the working chunk was *unused* hence the error we see. Since this unused portion of the chunk is merely whitespace, the level of the error is low (the error level is 0), so we could choose to discount this problem, however, just for the fun of it, lets change the offending rule to grab the leading whitespace.

Edit the rule by clicking on the rule in the Rules Text Area (at the `start=` part of the rule). You will see a blinking cursor and can edit the rules text (refer to section 2.3.1 if you need a few pointers on how to use the editing functions). After editing the rule should now look something like this:

```
<match tag="email" start="^\s*?Subject:\" end="-{10,80}\n"
statusOfStart="accept" >
```

Re-running the parser will show that we still have the *same* first error (!?!):

```
ERROR:[1][0][leading string]:
```

However, left clicking on the error shows the following (different!) rule is highlighted:

```
<match tag="subject" start="Subject:\s?" end="\n"/>
```

So, mercifully, a different problem is occurring: we forgot to propagate our prior changes down to the appropriate child rule. We simply need to change this child rule to reflect the change in its parent rule. After editing it now looks like this:

```
<match tag="subject" start="^\s*?Subject:\s?" end="\n"/>
```

Re-running the parser should now give us the following (*different!*) leading error:

```
ERROR:[1][1][required rule]:rule "<match tag="email" start=
"^\s*?Subject:" end="\s*?-{3,80}\n" >" failed to match but is required.
```

It is related to the other following error which is a *match remainder error*. The "email" block of rules is receiving too much text, a combination of both the email portion of the list (which we want) AND the trailing block of text that occurs at the end of the email (which we don't want). There are a number of ways which we could use to fix the problem. For starters we could edit the input text to remove the offending trailing text. This is only a half-solution though because we can expect all the mail which is sent out by the listserver to have this trailing section (and we definitely *don't* want to have to edit every incoming message!). Lets simply change the parent email rule to NOT be required, e.g. after editing it looks like:

```
<match tag="email" start="^\s*?Subject:\" end="-{10,80}\n"
statusOfStart="accept" required="no">
```

and we can insert a "*trash match*"[3] rule outside of the `repeat` block to tag up the trailing text. Re-running the parser (for the last time, whew!) shows NO errors (yeah!). The final, editted rules set would look like this:

```
<txt2XML root_tag="XMLMail">


    <!-- Sample_rules.xml.  For use in the txt2XML
GUI tutorial.  These rules are designed to
ingest mailing list digests into XML (database).
-->


      <!-- first, remove the preamble, no info we want there -->
<ignore end="-{10,80}\"/>


      <!-- loop thru document, tagging up each email -->
<repeat>
<!-- each email begins with "Subject:" line and
ends with 10 to 80 dashes and a newline -->
<match tag="email" start="^\*?Subject:" end="\*?-{3,80}\" statusOfStart="accept"
required="no">
<match tag="subject" start="^\*?Subject:\?" end="\"/>
<match tag="sender" start="From:\?" end="\"/>
<match tag="time" start="Date:\?" end="\"/>
<match tag="number" start="X-Message-Number:\?" end="\"/>
<match tag="body"/>
</match>
</repeat>


    <!-- remove last stuff w/ trash match -->
<match tag="TRASH"/>


</txt2XML>
```

[3]Yes, this is a technical term, see section 3.5.3

PART 3

txt2XML Parsing Rules

3.1. Introduction

In this section we'll get into some details of the components and use of the *txt2XML* parsing rules. Before we proceed further, a few definitions are needed:

1) A *txt2XML* "rules document" is an XML document which controls how/what content the *txt2XML* parser tags from the input document.

2) Each rules document comprises an ordered set of "rules" (see table 3.1 and appendix B) which occur as elements (nodes) within the rules document.

To help illustrate these points, here is an example of a simple rules document:

```
<?xml version="1.0" ?>
<txt2XML root_tag="OutputFileRootTag" >
<!--

    A simple rules document which has the root tag
    ``OutputFileRootTag'' and tags the entire input
    document content under one element
    ``entireBodyOfText''.

-->

    <match tag="entireBodyOfText" />

</txt2XML>
```

Using these rules, if the input document where to look like this:

```
This is a single line document.
```

then the output document would look like this:

```
<?xml version="1.0" ?>
<OutputFileRootTag>

    <entireBodyOfText>
        This is a single line document.
    </entireBodyOfText>

</OutputFileRootTag>
```

An analogy to programming code will serve as a useful paradigm for understanding the relationships of these components to each other. *txt2XML* acts as the compiler and runtime interpreter for the rules document, the rules document serves as the programming code and the rules serve as elements of the "parser language". The input and output documents are the input and output data.

Indeed, we can extend this analogy further to the process of creating the rules document. Skill needed to write a rules document is very much the same as that needed to write program code; the user formulates the rules, tests output, then corrects the rules based on their testing until the desired output is achieved.

In the rest of this chapter we will describe the content of the rules document, how that document is used by *txt2XML* to tag the content of the input document, how matching by the rules works, and examples of rules structures to achieve common tagging tasks.

3.2. About the Rules Document

In line with a programming analogy a couple of points concerning the rules document can be made. First, just as program code obeys the formalism of its target language, there is a definite formalism that all rules documents (and declarations of various rules within the rules document) must obey. This is described in the rules document DTD (appendix D) which we can distill the gist of down to a few important points:

1) Some of the rules have attributes which control their behavior. There are default values for some attributes.

Table 3.1: Summary of Available Parsing Rules and Attributes.

Parsing Rule	Allowed Attributes	Rule Description
`txt2XML`	`root_tag`[1], `script_version`, `accept_null_matches`, `entities_to_encode`, `include_errors_in_output`, `error_tag`	Root element in XML rules document. Doesn't have to have child rules, but without them not much of anything will get parsed!
`match`	`start`, `end`, `tag` `statusOfStart`, `statusOfEnd`, `test`, `remainder` `required`	Tries to match text passed to it from the current working chunk. If successful the matched text is either tagged and inserted into the output document or passed to child rules (if they exist).
`repeat`	—	This rule will cause to loop over its child rules until *none* of the child rules is successful in extracting something from the current chunk. Repeat rules must have child rules.
`choose`	—	Allows choice between child rules. Must have child rules. *Use of this rule is deprecated.*
`print`	`what`	Print current chunk to STDOUT. Useful for debugging rules without the GUI. *Use of this rule is deprecated.*
`halt`	—	Halts the parser run when encountered. In the GUI, it is possible to proceed from one halt rule to the next. Similar in functionality to break point functionality in C debuggers.
`ignore`	`start`, `end`, `tag` `statusOfStart`, `statusOfEnd`, `test`, `remainder` `required`	Tries to match text passed to it from the current working chunk. If successful it will ignore this text (so it won't error out or appear in output) Ignore rules may not have children.

[1]This attribute is required.

2) Some rules are allowed to take other rules as child nodes.

3) The `txt2XML` must always be the root node of any rules document.

4) The "`root_tag`" attribute of the `txt2XML` must be explicitly defined in any set of rules.

Second, it is appropriate to reiterate that a good "programming style" is important when writing the rules document. Good programming style for a rules document should include:

1) Use generous and consistent indentation to promote readability of the rules document.

2) Use generous commenting of various rules to promote understandability of the rules document.

3.3. How the *txt2XML* Uses the Rules to Tag Text

The rules document describes the procedure whereby the *txt2XML* parser will tag the input text. As the procedure is executed, the input text is broken up into smaller "working chunks" of text which are are either tagged and inserted in the output document or passed onto the next part of the procedure. There is no limit (other than the number of characters within the text) to how many times a working chunk may be sub-divided before it is actually tagged. Each time the parser evaluates a working text chunk, it is using a rule within the rules document to do so. Each evaluation is an "rules event" for which there is an associated rule and "current working chunk".

The parser finishes its run when the procedure described by the rules document ends, is stopped specifically by a halt rule or the parser runs out of input text to process. The parser will record an error in the output document IF it runs out of rules to execute while input text remains OR if input text remains but it has executed all of its instructions in the rules document.

3.4. Matching Text with Rules

Directing the *txt2XML* parser in which text to tag or pass on is the most critical functionality for any rules writer to understand. Both the ignore and match rules may be used to grab parts of the working chunk. The mechanism for determining which text will be matched and then tagged (or passed onto child nodes) is controlled by the "`start`" and "`end`" attributes of these rules. Both `start` and `end` attributes take a regular expression as their value. The text that the expression in either the `start` or `end` attributes matches is respectively known as the "Start Match" and "End Match". The text which lies between the Start Match and End Match is known as the "Match Text". Figure 3.1 shows several examples how this matching model works.

Before we move on, we need to briefly discuss the regular expression matching used by *txt2XML*. As you may expect, *txt2XML* uses the *Perl* regular expression matching conventions with one exception−−you can not use the parenthesis structure. The reason for this boils down to the fact that the *txt2XML* parser internals make special use of parenthesis already so that use of parenthesis in a rule regular expression screws up *txt2XML* parser engine. For example, the following regular expression is OK:

```
^[^a][A|b][A-z]\d.*?\n-{1,100}$
```

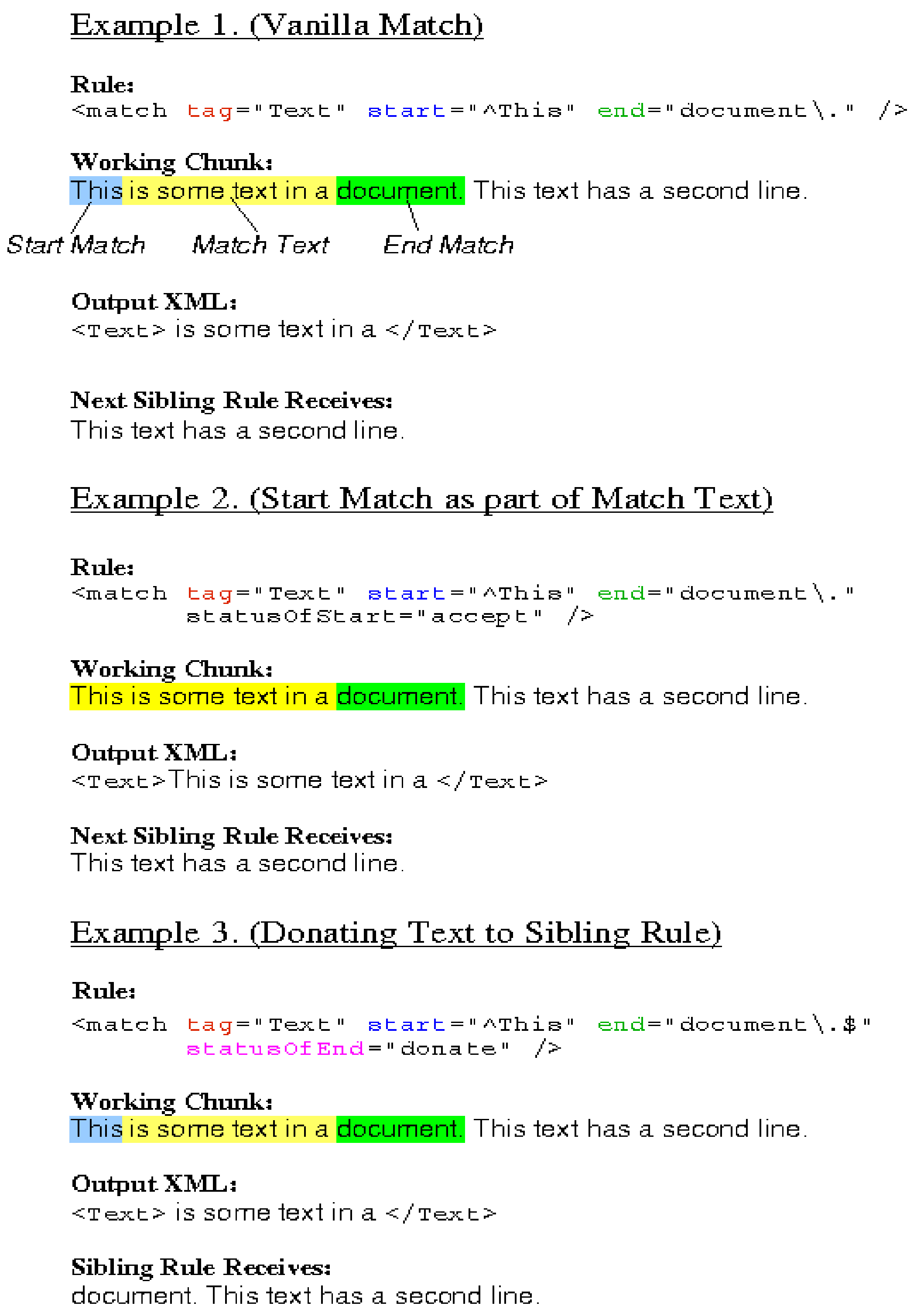
while this regular expression is NOT:

```
(^word|dude)
```

For more information on *Perl* regular expressions, check out these references:

Wall, L., Christensen, T., & Schwartz, R.L., "*Programming Perl*", 2nd ed., O'Reilly & Associates, Inc., Sebastopol, CA, 1996

Friedl, J.E.F., "*Mastering Regular Expressions*", 1st ed., O'Reilly & Associates, Inc., Sebastopol, CA, 1997



Example 1. (Vanilla Match)

Rule:

```
<match tag="Text" start="^This" end="document\." />
```

Working Chunk:

This is some text in a document. This text has a second line.

Start Match
Match Text
End Match

Output XML:

`<Text>` is some text in a `</Text>`

Next Sibling Rule Receives:

This text has a second line.

Example 2. (Start Match as part of Match Text)

Rule:

```
<match tag="Text" start="^This" end="document\."
       statusOfStart="accept" />
```

Working Chunk:

This is some text in a document. This text has a second line.

Output XML:

`<Text>`This is some text in a `</Text>`

Next Sibling Rule Receives:

This text has a second line.

Example 3. (Donating Text to Sibling Rule)

Rule:

```
<match tag="Text" start="^This" end="document\.$"
       statusOfEnd="donate" />
```

Working Chunk:

This is some text in a document. This text has a second line.

Output XML:

`<Text>` is some text in a `</Text>`

Sibling Rule Receives:

document. This text has a second line.

Figure 3.1. This figure shows the Match Text, Start Match, and End Match portions of the working chunk for various values of the `start`, `end`, `statusOfStart`, and `statusOfEnd` attributes.

3.5. Examples of Rules Use

Here are some examples that illustrate several common problems often faced by the person writing the rules.

3.5.1. Controlling the Working Chunk that Child/Sibling Rules See

Match rules can be used to control what part of the working chunk which is passed on to child and sibling rules. Control is obtained by using the `statusOfStart` and `statusOfEnd` attributes.

Consider the following set of rules:

```
<match tag="firstRule" start="^" end="text\."
statusOfStart="drop" statusOfEnd="drop" >

    <match tag="childRule" />

</match>
<match tag="siblingRule" />
```

operating on the following working chunk:

```
This is a chunk of text. This is more text.
```

will produce the following block of output XML:

```
<firstRule>

    <childRule>This is a chunk of </childRule>

</firstRule>
<siblingRule> This is more text.</siblingRule>
```

There are a couple of things are going on here. First, notice that the unmatched portion of the working chunk is automatically passed to the next *sibling* rule (which is tagged by it). Second, notice that only the Match Text portion of the working chunk is passed to that match rules children. In our example above you can see that the child rule received only a part of the first sentence (the end of the sentence was part of the End Match). This is because in the first rule the values of its `statusOfStart` and `statusOfEnd` attributes is set to “drop” (the default value). This means that the portion of the working chunk that is part of the Start Match and End Match will be eliminated (not passed on to the children).

If we make a small change the rules block by setting the `statusOfEnd` attribute in the first rule to “accept”, e.g. using the following rules:

```
<match tag="firstRule" start="^" end="text\."
statusOfEnd="accept" >

    <match tag="childRule" />

</match>
<match tag="siblingRule" />
```

the following block of output XML is obtained:

```
<firstRule>

    <childRule>This is a chunk of text.</childRule>

</firstRule>
<siblingRule> This is more text.</siblingRule>
```

The `statusOfStart` works similarly to the `statusOfEnd` attribute. If statusOf-Start is set to “accept” then the portion of the working chunk belonging to the Start Match will be passed onto the child rules. Unlike `statusOfStart`, the `statusOfEnd` attribute can also take the value of “donate”. When the value of “donate” is chosen, then the End Match portion of the working chunk will be passed onto the sibling rule. Consider the following changed rules block where we have set statusOfEnd to “donate”:

Consider the following set of rules:

```
<match tag="firstRule" start="^" end="text\."
statusOfEnd="donate" >

    <match tag="childRule" />
```

```
</match>
<match tag="siblingRule" />
```

then the following block of output XML is obtained:

```
<firstRule>

    <childRule>This is a chunk of </childRule>

</firstRule>
<siblingRule>text.  This is more text.</siblingRule>
```

3.5.2. Passing the Working Chunk Back to the Parent Rule

The `remainder` attribute makes it possible to pass the working chunk that remains (e.g. after a match/ignore rule is finished) back up to the parent rule level. Normally the `remainder` attribute is set to "error" meaning that any remaining chunk left, after all rules at that level have been exhausted, is errored out. However, if the child rule has remainder set to "data", then the remainder chunk is passed back up to the parent rule. If the parent rule also has the `remainder` attribute set to "data", the remainder chunk is passed up to the next parent rule level.

Consider the following example set of rules:

```
<match tag="parentRule" >

    <match tag="childRule" end="This" remainder="data"
    />

</match>
```

operating on the following working chunk of text:

```
This is a chunk of text. This is more text.
```

will produce the following output text:

```
<parentRule>
```

```
    <childRule>This is a chunk of text.</childRule>
is more text.  </parentRule>
```

Note if we set the `statusOfEnd` attribute (see sections 3.5.1, B.2.2) in the child rule to "donate" we will pass back up its End Match text as well as the remainder text. The output XML document would then look like the following:

```
<parentRule>
    <childRule>This is a chunk of text.</childRule>
This is more text.  </parentRule>
```

3.5.3. Removing Extraneous Text

Often there will occur some text which is undesirable to tag in the output document. The rules writer may choose to use either a "*trash match*" or ignore rules to eliminate this unwanted text.

A *trash match* is a match statement where a common tag such as "GARBAGE" is used. This makes it easy for a post-processing script (such as one based on XSL) to run on the output document to rip out this text. The ignore rule simply deletes the Match Text portion and may be the favorable option to trash matching in the cases that you won't post-process your output or are confident that the rule is getting exactly the text you want it to. It is certainly desireable to use trash matching over ignore rules when you are trying to debug your output.

Consider the following example set of rules that incorportate the ignore rule:

```
<match tag="parentRule" >
    <ignore end="This is" remainder="data" />
</match>
```

operating on the following working chunk of text:

```
This is a chunk of text. This is the remaining text.
```

will produce the following output text:

```
<parentRule> the remaining text.</parentRule>
```

3.5.4. Flexible Tagging

Sometimes it is desirable to allow some flexibility in whether a rule can match text or not. Normally, match and ignore rules are required, and when they fail to match text within the working chunk an error is recorded in the output text. By setting the `required` attribute in these rules to “no” it is possible to quietly fail to match text in the working chunk.

Consider the following example set of rules:

```
<match tag="parentRule" >

    <ignore end="Strange String" required="no" />
    <!-- Now the child match rule gets
    remaining working chunk -->
    <match tag="childRule" required="no" />

</match>
```

operating on the following working chunk of text:

```
This is a chunk of text.
```

will produce the following output text:

```
<parentRule>

    <childRule>This is a chunk of text.</childRule>

</parentRule>
```

3.5.5. Creating Control Statements

The `test` attribute allows the rules writer some limited possibility to create control statements within the rules document. The `test` attribute, if set, takes a *Perl* regular expression. If a match is made, then the expression within the `test` attribute is then evaluated on the Match Text.

Consider the following example set of rules:

```
<match tag="parentRule1" test="\d{1,3}" required="no">
    <match tag="childRule1"/>
</match>
<match tag="parentRule2" test="chunk" required="no">
    <match tag="childRule2"/>
```

operating on the following working chunk of text:

```
This is a chunk of text.
```

will produce the following output text:

```
<parentRule2>
    <childRule2>This is a chunk of text.</childRule2>
</parentRule2>
```

You can see that only the above rules work similarly to a case switch. Note that the use of the `required` attribute is also needed to make this rules block work smoothly (see sections 3.5.4 and B.2.2 for more information on using the `required` attribute).

3.5.6. Tagging Repeated Text Structures

Often the same lexical structure will occur repeatedly within a document. It is then convenient to use the repeat rule to create shorthand blocks of rules.

Consider the following example set of rules:

```
<repeat>

    <!-- match from the beginning to the
    first newline char -->
    <match tag="childRule" end="\n" />

</repeat>
```

operating on the following working chunk of text:

```
This is the first line.
This is the second line.
This is the third line.
```

will produce the following output text:

```
<childRule>This is the first line.</childRule>
<childRule>This is the second line.</childRule>
<childRule>This is the third line.</childRule>
```

3.5.7. Tagging Null Text Content

It can sometimes happen that the rules writer wants to match null content (in other words the Match Text portion of the working chunk is equal to null content). Set the `txt2XML` attribute `accept_null_matches` to "Yes" to achieve this goal.

Consider the following example set of rules:

```
<txt2XML root_tag="Info" accept_null_matches="yes" >

    <match tag="Name" start="^" end=":" />
    <match tag="Address" start="^" end="\n" />
```

```
</txt2XML>
```

operating on the following working chunk of text:

```
:150 Moon Orbit Dr.
```

will produce the following output text:

```
<Info>

    <Name></Name>
    <Address>150 Moon Orbit Dr.</Address>

</Info>
```

3.5.8. Arranging Output Tag Order

There is no sensible way to get *txt2XML* to arrange the output tag order. We purposely have left this functionality out of *txt2XML* because this is something that can be accomplished well by other software, namely XSL.

Appendix A

Summary of Options

Table A.1: Summary of General Options in *txt2XML*.

Option	GUI menu	Command line
Help	Help->About Help->Parsing Text	`-h` —
GUI mode	—	`-g`
Split Windows Display	—	`-split`
Tiny Display (800x600)	—	`-tiny`[1]
Small Display (1024x768)	—	`-small`[1]
Normal Display (1280x1024)	—	`-normal`[1]
Large Display (1600x1200)	—	`-large`[1]
Print Score to STDERR	—	`-s`
Change Halt on Error Level	Parsing Control->Halt on Error Level	`-halt_on_error` [value]
Turn OFF Tagged Errors in Output	Parsing Control->Tagged Errors in Output	`-e`
Turn OFF Encoding of Output Text	Parsing Control->Entify Chars in Output	`-n`
Allow Null Matches in Output Text	Parsing Control->Allow Null Matches	`-null_match`
Change Display Size	Option->Change Tool Display Size	
Change Font Size	Option->Change Text Font Size	
Change Text Color	Option->Change Text Fg/Bg Color	

[1]Turns on GUI automatically (no need to supply `-g` switch).

Table A.2: Summary of Mouse Bindings in *txt2XML*.

Key Binding	Function
Input Text Bindings	
right mouse button click	relocate editor cursor
`<Ctrl>` + right mouse button click	mark text break point
left mouse button click	paste swiped text at cursor location
Rules Text Bindings	
left mouse button click	relocate editor cursor
`<Ctrl>` + left mouse button click	make rule current event
right mouse button click	paste swiped text at cursor location

Table A.3: Summary of GUI Key Bindings in *txt2XML*.

Key Binding	Function
`<Alt>−b`	Back 1 Rule.
`<Alt>−B`	Back 10 Rules.
`<Alt>−c`	Clear Input Text Break Point.
`<Alt>−f`	Forward 1 Rule.
`<Alt>−F`	Forward 10 Rules.
`<Alt>−n`	Skip to Next <halt> Rule.
`<Alt>−p`	Parse Document.
`<Alt>−q`	Exit Program
`<Alt>−v`	Toggle Input Text Chunk View.

Appendix B

Rules Glossary/API

A glossary of the *txt2XML* rules.

B.1. `txt2XML` Rule

B.1.1. Description

For any rules document, there is always one `txt2XML` rule and it must be the root node. This rule is used to specify special processing instructions and the version of the rules script.

B.1.2. Attributes

Attribute	Default Value	Description
`root_tag`	—	Specifies the root element of output document. Can take any string as a value.
`script_version`	—	Specifies which version this rules document is. The version indicated here will be stamped in the output document. Can take any string as a value.
`accept_null_matches`	no	Allow match and ignore rules to "grab" null text. Can take either "yes" or "no" as a value.
`entities_to_encode`	"&<'	Specify which entities should be encoded by the parser. Can take any string as a value.
`include_errors_in_output`	no	Include errors as tagged text in the output document. Can take either "yes" or "no" as a value.
`error_tag`	ERROR	Value of the tag to use when including errors in the output document. Can take any string as a value.

B.1.3. Usage

You must specify a value for the `root_tag` attribute. All other attributes are optional. It is a good idea to use the `script_version` attribute to indicate in the output document which set of your rules created it.

B.2. `match` and `ignore` Rules

B.2.1. Description

These two rules are the heart and soul of tagging input text.

B.2.2. Attributes

Attribute	Default Value	Description
`start`	^	Indicates the beginning of the match. Takes any *Perl* regular expression[1] as a value.
`end`	$	Indicates the end of the match. Takes any *Perl* regular expression[1] as a value.
`tag`[2]	—	Indicates the name of the tag for matched text in the output document. Can take any string as a value. This attribute is not used by the `ignore`.
`statusOfStart`	drop	When a match is successfull, this rule indicates what should be done with the text that was matched by the regular expression in the `start` attribute of this rule. Can take a value of “drop” or “accept”.
`statusOfEnd`	drop	When a match is successfull, this rule indicates what should be done with the text that was matched by the regular expression in the `end` attribute of this rule. Can take a value of “drop”, “accept” or “donate”.
`test`	—	If a successfull start and end match occur the matched text is subjected to this further check. If the check indicated by the test attribute is passed the match or ignore rule is carried out as normal, otherwise the rule fails to match. May take any *Perl* regular expression or null (don't check) as a value.
`remainder`	error	Indicates how remaining text should be treated.
`required`	yes	Whether or not this rule is required to match. If it fails to match and is required, an ERROR tag will be created in the output text at the point the rule failed. May take either “error” or “data” as a value.

1. With the exception that you can not use the parenthesis. See section 3.4 for an explanation.

2. The `tag` attribute is only available for the `match` rule.

B.2.3. Usage

The `match` and `ignore` rules share most of the same attributes and encompass overlapping usage. Both rules act on the working chunk (see section **??**) by examining it and taking part of it away for their own purposes. The text which is *not grabbed* is passed onto the next *sibling* rule. What happens to the grabbed (or "matched") tex depends on the rule and the context in which it is used. The simplest rule to explain is the `ignore` rule. Text grabbed from the working chunk by this rule is dropped from consideration by the parser. The text grabbed by a `match` can be used in several different ways. If the given `match` has a `tag` attribute, then it is considered a "*tagged match*". This type of `match` will always insert a tag in the output text IF it grabs text from the working chunk. The name of the inserted tag is specified by the `tag` attribute and the the grabbed text (in this case, the Match Text) will be the text that is tagged under the specified tag. If the tagged match rule has child rules, then a tag is opened for the tagged match and the text it grabbed from the working chunk is passed onto its child rules (and the text that is passed on is considered to be the working chunk by the children). If any child rules are tagged `match` rules, then they may open tags nested within the parent match rule. Consider the following rules:

```
<match tag="parentTaggedMatch" start="^" end="$" >

    <match tag="child1" start="^" end="\n" /> <match
    tag="child2" start="^" end="\n" />

<match>
```

operating on the following working chunk of text:

```
This is the first line. This is the second line.
```

will produce the following output text:

```
<parentTaggedMatch>
```

Table B.1: Differences in usage of Ignore and Match Rules

Rule Type	Child Nodes
`match` with the `tag` attribute ("tagged match")	May have child rules.
`match` without the `tag` attribute ("*non-tag matching*")	Must have child rules.
`ignore`	Never has child rules.

```
    <child1>This is the first line.</child1>
    <child2>This is the second line.</child2>

</parentTaggedMatch>
```

If a `match` lacks the `tag` attribute, it is being used to selectively grab a portion of the working chunk to pass onto its child rules. In this case the `match` *must* have at least 1 tagged `match` rule OR `ignore` rule as a child rule. These cases are summarized in table B.1.

You might wish to refer to section 3.4 for a more thorough description of how the parser matches text. There are other helpful examples of the usage of both `match` and `ignore` rules in section 3.5 and sprinkled throughout this appendix.

B.3. `repeat` Rule

B.3.1. Description

The `repeat` rule is used to create looping operations in the rules.

B.3.2. Attributes

The `repeat` rule does not have any attributes.

B.3.3. Usage

The `repeat` rule must have child rules. You can have any rule as a child (including other repeat rules). For example:

```
<repeat>

    <match tag="Line" start="^" end="\n" />

</repeat>
```

will tag each line within a given chunk with separate "Line" tags.

B.4. `halt` Rule

B.4.1. Description

The `halt` is used to stop the parser from proceeding further.

B.4.2. Attributes

The `halt` rule does not have any attributes.

B.4.3. Usage

This rule is useful for debugging the rules. In the GUI you may select to proceed to the next halt statement. In this sense the `halt` rule is similar to a "stop" statement in a C debugger. For example, consider the following snipett of parser rules:

```
<matchtag="Tag" start="^" end="\n" />
<halt/>
<ignore/><! − − ignore the rest of the chunk −− >
```

In the above rules the halt rule will stop the parser just after the match rule is executed but before the repeat rule is run.

If a halt rule exists within a repeat rule, eg

```
<repeat>

    <halt/>
```

```
        <match tag="Line" start="^" end="\n" />
    </repeat>
```

then the parser is halted on the first pass through the repeat loop (and, in the snipett above, *before* the match rule is reached). By using the GUI Next Halt Button it is possible to skip forward. Each press of the Next Halt Button would advance the parser run one pass further along in the repeat loop (in the example rules above the user would see the parser capture one more line of text from the working chunk for each press of the Next Halt Button).

B.5. `choose` Rule

B.5.1. Description

The `choose` may be used to flexibly group together several different `match` or `ignore` rules. Only one of the child rules is allowed to match text in the working chunk and the children are considered in the order in which they appear. Once a match is obtained, the remaining child rules are not considered and the parser moves to the next sibling rule (of the `choose`). As an exception to their normal usage, the `required` attribute of the child `match` and `ignore` rules defaults to “no” instead of “yes”.

B.5.2. Attributes

The `choose` does not have any attributes.

B.5.3. Usage

The use of this rule is deprecated. It must have child rules. Note that GUI interface will not work properly and parser is not extensively tested (so it might knuckle under and give erroneous results).

It is possible to use match rules within a repeat rule to simulate the functionality of the choose rule. Consider the following rules:

```
<choose>

    <match tag="number" start="^" test="\d" end="\n"
    />
    <match tag="noNumber" start="^" end="\n" />

</choose>
```

operating on the following working chunk of text:

```
This is the line has no numbers in it.
This line does have numbers. It is the number 5.
```

will produce the following output text:

```
<number>This line does have numbers.  It is the number
5.</number>
```

These rules would then pass on to the next sibling rule the following chunk:

```
This is the line has no numbers in it.
```

Here is a replacement set of rules that does almost the same thing but without the choose rule:

```
<repeat>

    <match remainder="data" >
        <match  tag="number"  start="^"  test="\d"
        end="\n" required="no" remainder="data" />
        <match  tag="noNumber"  start="^"  end="\n"
        required="no" remainder="data"/>
    <match>

</repeat>
```

The main difference between using the `choose` rule example and the `match/repeat` rule example being that the later will try to get as much of the working chunk as possible. This would also be the case if we wrapped the `choose` rule example rules in a `repeat` rule too. The advantage of using something like the above `match/repeat` rules are that the output is not out of order in which it appeared. The disadvantages are that the `match/repeat` rules are harder to formulate and precise `choose` functionality can not be replicated.

B.6. `print` Rule

B.6.1. Description

The `print` rule causes the parser to print out information on the proceeding rule when it is encountered.

B.6.2. Attributes

`what`	What to report in the print. Takes either "rule" or "chunk".

B.6.3. Usage

The use of this rule is deprecated (use the *txt2XML* GUI mode instead!). Here is an example usage:

```
<match tag="Tag" start="^" end="\n" />
<print what="chunk" />
```

will print out the working chunk for the preceeding `match` to standard error. The `print` will *not* halt the parser (use a `halt` after the `print` to do that).

Appendix C

TODO/BUGS

Although we feel that *txt2XML* has all of the basic functionally to tackle real problems there remain some interesting possibilities for advanced features. Here is a brief list of features that we are thinking of adding (when time permits!) and bugs that we are aware of that need to be fixed.

i) Validation of rules (versus stock rules DTD).

ii) Click on output to find input chunk and rule which produced that output.

iii) Allow a batch mode to run on multiple files.

iv) Allow user to configure the keyboard/mouse short cuts to taste.

v) Allow user to browse/use all of the fonts available on their machine.

vi) Allow user to save configuration of GUI options between sessions.

vii) Allow validation of output document if a DTD is specified.

viii) Reorganize the content of this document for clarity and then convert it into XML!![1]

ix) BUG: Fix the memory leak seen in the GUI program.

[1] Isn't it ironic we wrote this in LaTeX?!?

Appendix D

txt2XML Rules DTD

This is the DTD that may be used to validate any *txt2XML* set of rules.

```
<!-- txt2XML.dtd - Text-to-XML Rules Language
                Date April 13, 2000
                XML Language Document Type Definition (DTD)
                for rules used by the txt2XML processor
                (txt2XML.pl) to translate semi-structured
                text data into XML. Essentially, a way to
                pop XML tags into documents that do not have
                XML tags.
-->
<!ELEMENT txt2XML (match | repeat | choose | print | halt | ignore)+ >
  <!ATTLIST txt2XML
                  root_tag CDATA #REQUIRED
                script_version CDATA #IMPLIED
                accept_null_matches (yes|no) "no"
                include_errors_in_output (yes|no) "no"
                entities_to_encode CDATA #IMPLIED
                error_tag CDATA "ERROR"
>
<!ELEMENT match (match | repeat | choose | print | halt | ignore)* >
  <!ATTLIST match
                  start CDATA "^"
                end CDATA "$"
                tag CDATA #IMPLIED
                statusOfStart (drop|accept) "drop"
                statusOfEnd (drop|accept|donate) "drop"
                test CDATA #IMPLIED
                required (yes|no) "yes"
                remainder (data|error) "error"
>
<!ELEMENT repeat (match | repeat | choose | print | halt | ignore)+ >
<!ELEMENT choose (match | repeat | choose | print | halt | ignore)+ >
<!ELEMENT print EMPTY >
<!ATTLIST print
```

```
                what (rule|chunk) "chunk"
>
<!ELEMENT halt EMPTY >
<!ELEMENT ignore EMPTY >
  <!ATTLIST ignore
                  start CDATA "^"
                end CDATA "$"
                statusOfStart (drop|accept) "drop"
                statusOfEnd (drop|accept|donate) "drop"
                test CDATA #IMPLIED
                required (yes|no) "yes"
                remainder (data|error) "error"
>
```

Appendix E

Sample Text

Here are the sample rules used in the GUI tutorial.

```
<?xml version="1.0"?>
<txt2XML root_tag="XMLmail">

    <!-- Sample_rules.xml.  For use in the txt2XML GUI tutorial -->

    <!-- first, remove the preamble, no info we want there -->
    <ignore end="-{10,80}\"/>

    <!-- loop thru document, tagging up each email -->
    <repeat>
          <!-- each email begins with "Subject:" line and
                 ends with 10 to 80 dashes and a newline
            -->
          <match tag="email" start="Subject:" end="\*?-{3,80}\"
statusOfStart="accept">
                <match tag="subject" start="Subject:\?" end="\"/>
                <match tag="sender" start="From:\?" end="\"/>
                <match tag="time" start="Date:\?" end="\"/>
                <match tag="number" start="X-Message-Number:\?" end="\"/>
                <match tag="body"/>
          </match>

    </repeat>

</txt2XML>
```

Here is the sample text used in the GUI tutorial.

```
From ???@??? 00:00:00 1997 +0000
Return-Path: <bounce-perl-xml-108857@listserv.activestate.com>
Received: from some.site.net (some.site.net [127.100.100.0])
                by computer.some.site.net (8.9.3/8.9.3) with ESMTP id DAA30309
                for <user@some.site.net>; Sun, 9 Apr 2000 03:11:12 -0400
Received: from some.site.net (some.site.net [127.100.99.5])
                by some.site.net (8.9.3/8.9.3) with ESMTP id DAA13116
                for <user@some.site.net>; Sun, 9 Apr 2000 03:12:11 -0400 (EDT)
Received: from listserv.activestate.com (listserv.activestate.com [199.60.48.6])
                by some.site.net (8.9.3/8.9.3) with SMTP id DAA20147
                for <user@some.site.net>; Sun, 9 Apr 2000 03:12:08 -0400 (EDT)
Date: Sun, 09 Apr 2000 00:00:49 -0700
Subject: perl-xml digest: April 08, 2000
To: "perl-xml digest recipients" <perl-xml@listserv.activestate.com>
From: "Perl-XML Mailing List digest" <perl-xml@listserv.activestate.com>
Reply-To: "Perl-XML Mailing List" <perl-xml@listserv.activestate.com>
```

```
Precedence: bulk
Message-Id:
<LYR108857-53058-2000.04.09-00.00.49--user#some.site.net@lyris.ActiveState.com>
List-Unsubscribe: <mailto:leave-perl-xml-108857H@lyris.ActiveState.com>
List-Software: Lyris Server version 3.0
List-Subscribe: <mailto:subscribe-perl-xml@lyris.ActiveState.com>
List-Owner: <mailto:owner-perl-xml@lyris.ActiveState.com>
X-List-Host: ActiveState <http://www.ActiveState.com>
Sender: bounce-perl-xml-108857@listserv.activestate.com
X-listname: perl-xml
X-ListMember: [user@some.site.net]

Perl-XML Mailing List Digest for Saturday, April 08, 2000.

1. Another XML::Parser 2.28 question
2. Re: Another XML::Parser 2.28 question

----------------------------------------------------------------------

Subject: Another XML::Parser 2.28 question
From: John Doe <john.doe@another.site.net>
Date: Sat, 8 Apr 2000 10:52:46 -0400 (EDT)
X-Message-Number: 1

HI,

I have a general question about XML::Parser 2.28: has anyone managed to
use it to output a DTD that's "close" to the one in the input document?

By "close" I mean that the DTD should include the external entities (both
the declaration and the call) but not the content of those entities.

If I have
<!ENTITY % ent SYSTEM "ent.file">
%ent;

I just want to get that back in the output.
At the moment I can get the content of ent.file parsed and I can output
the entities defined in t, but I have found no way to get ANYTHING about
the 2 lines above.

When I define handlers the Entity one does not see the first line and
the ExternEnt one is never called.

Any one managed to do better?

John Doe
Senior Programmer-Analyst
First Author Company
john.doe@another.site.net
http://somehomepage.html


----------------------------------------------------------------------

Subject: Re: Another XML::Parser 2.28 question
From: Jane Doe <jane.doe@different.site.net>
```

Date: Sat, 8 Apr 2000 11:14:00 –0400 (EDT)
X–Message–Number: 2

On Sat, 8 Apr 2000, John Doe wrote:

> Hi,
>
> When I define handlers the Entity one does not see the first line and
> the ExternEnt one is never called.

I found the problem. XML::Parser::new sets the default handlers for ExternEnt AFTER setting the user ones, thus overwriting the users'

An easy work around is to use SetHandlers after the new.

To fix it in XML::Parser lines 93–100 should be moved before the previous block, to line 67.

I'm talking about those line:

```
    if ($have_LWP) {
        $handlers->{ExternEnt} = &lwp_ext_ent_handler;
        $handlers->{ExternEntFin} = &lwp_ext_ent_cleanup;
    }
    else {
        $handlers->{ExternEnt} = &file_ext_ent_handler;
        $handlers->{ExternEntFin} = &file_ext_ent_cleanup;
    }
```

Jane

–––

END OF DIGEST

–––
You are currently subscribed to perl–xml as: [user@some.site.net]
To unsubscribe, forward this message to
leave–perl–xml–108857H@lyris.ActiveState.com
For non–automated Mailing List support, send email to
ListHelp@ActiveState.com

Index