5 Parsing NCBI XML in Perl

Perl remains the programming language of choice for many in bioinformatics. Perl has excellent support for processing and manipulating text, finding regular expression patterns, retrieving files via the Internet, and connecting to a wide variety of relational databases. This makes it an ideal language for parsing flat text files, such as GenBank Flat File records, integrating biological data from multiple sources, and performing sequence analysis. Building on these strengths, the bioinformatics community has developed BioPerl [72], a very successful open source module that includes numerous features, including the ability to retrieve biological data from remote data sources, run BLAST searches, and manipulate sequence data.

Perl also has excellent support for XML, and is supported by a wide variety of third-party open source XML modules. This chapter provides an introduction to XML parsing in Perl, and introduces two standard interfaces: the Simple API for XML (SAX) and the Document Object Model (DOM). To explore SAX, we focus on the `XML::SAX` module, and to explore the DOM, we focus on the `XML::LibXML` module. To illustrate basic concepts, the chapter includes numerous examples for parsing XML documents from the National Center for Biotechnology Information (NCBI) at the U.S. National Institutes of Health. We also explore the NCBI EFetch service, and illustrate how to dynamically retrieve and parse sequence records from NCBI.

This chapter assumes that you have a basic familiarity with Perl programming, and understand the fundamentals of object-oriented programming in Perl. If you do not have such background, you may want to check out one of the recommended Perl references [56; 66; 67; 73; 74].

5.1 Introduction to XML Parsing in Perl

A quick search of CPAN, the Comprehensive Perl Archive Network, will reveal several hundred modules pertaining to XML. Given the sheer scope of XML-related modules, it is difficult to figure out where to begin. For example, both XML::Simple [59] and XML::Twig [63] provide "perlish" interfaces for parsing XML documents, and have proven to be quite successful. The original XML::Parser [75] module, originally written by Larry Wall himself, also remains quite popular. Given the multitude of options, we have chosen to focus exclusively on Perl modules, which adhere to specific well-known and well-documented standards. The chapter therefore focuses on Perl implementations of the Simple API for XML (SAX) and the Document Object Model (DOM). If you are interested in some of the other more "perlish" interfaces, such as XML::Simple or XML::Twig, check out some of the recommend references [59; 62].

For general questions about XML and Perl, check out the excellent Perl-XML FAQ [58], available at: *http://perl-xml.sourceforge.net/faq/*.

5.1.1 Tree-Based vs. Event-Based XML Parsers

XML parser interfaces are broadly divided into two categories: *tree-based* and *event-based*. A tree-based interface, such as the DOM, will parse an XML document and build an in-memory tree of all its XML elements. For example, consider the XML document in Listing 5.1. This is a sample GBSeq XML document, retrieved from NCBI. If you send this document through a tree-based interface, the parser will create a tree like that displayed in Figure 5.1. The root element is specified as the `GBSet` element and your application can navigate through the tree one node at a time. As your application traverses the tree, it can extract any and all data that it needs.

Contrast this with an event-based parser, such as SAX. An event-based interface will read the document one line at a time. Each time the parser encounters an important piece of data, it will immediately fire off an event. For example, when the parser reaches the start `<GBSeq_locus>` tag, it fires off a start element event. When it sees the text, "BC034957," it immediately fires off one or more character events. The same GBSeq example in Listing 5.1 will therefore trigger a very

Listing 5.1 Excerpt of a sample GBSeq document from NCBI

```
<?xml version="1.0"?>
<!DOCTYPE GBSet PUBLIC "-//NCBI//NCBI GBSeq/EN"
"http://www.ncbi.nlm.nih.gov/dtd/NCBI_GBSeq.dtd">
<GBSet>
 <GBSeq>
  <GBSeq_locus>BC034957</GBSeq_locus>
  <GBSeq_length>2547</GBSeq_length>
  <GBSeq_strandedness value="not-set">0</GBSeq_strandedness>
  <GBSeq_moltype value="mrna">5</GBSeq_moltype>
  <GBSeq_topology value="linear">1</GBSeq_topology>
  <GBSeq_division>PRI</GBSeq_division>
  <GBSeq_update-date>04-OCT-2003</GBSeq_update-date>
  <GBSeq_create-date>15-OCT-2002</GBSeq_create-date>
  <GBSeq_definition>Homo sapiens a disintegrin and metalloproteinase
  domain 2 (fertilin beta), mRNA (cDNA clone MGC:26432 IMAGE:4826530),
  complete cds</GBSeq_definition>
  <GBSeq_primary-accession>BC034957</GBSeq_primary-accession>
  <GBSeq_accession-version>BC034957.2</GBSeq_accession-version>
  <GBSeq_other-seqids>
    <GBSeqid>gb|BC034957.2|</GBSeqid>
    <GBSeqid>gi|34783181</GBSeqid>
  </GBSeq_other-seqids>
  ...
 </GBSeq>
</GBSet>
```

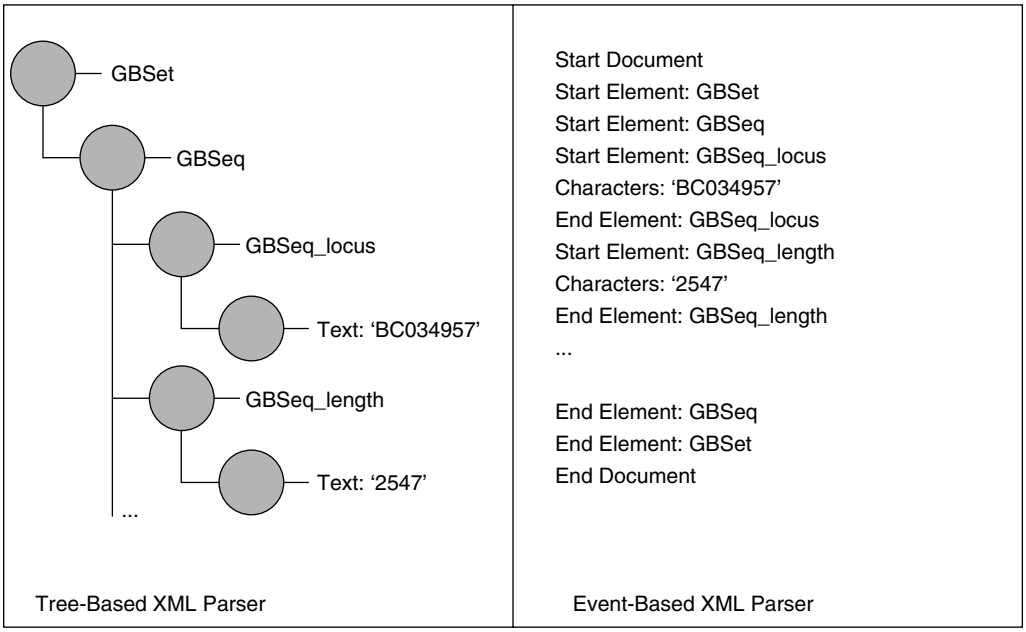

Figure 5.1 Tree-based vs. event-based XML parsing.

specific sequence of events (see Figure 5.1). To extract the XML data, your application must be registered to receive parsing events and act upon them appropriately.

In event-based parsing, the XML parser is typically referred to as an event *producer*, and your application handler is referred to as an application *consumer* [61]. As the XML parser reads in an XML document, the parser will "push" events to the application consumer, and the consumer can choose to record these events or ignore them. Note that event-based parsers are always sequential, and therefore do not provide random access to the XML document content.

5.1.2 Installing Modules via CPAN

All of the modules discussed in this chapter are available from CPAN, the Comprehensive Perl Archive Network. The easiest way to install them is via the interactive CPAN shell. For example, the following command starts the interactive CPAN shell and installs XML::SAX:

```
perl -MCPAN -e shell

cpan shell -- CPAN exploration and  modules installation (v1.61)
ReadLine support available (try  'install Bundle::CPAN')

cpan> install XML::SAX
```

Note, however, that the XML::LibXML module (discussed in the second half of the chapter) requires platform specific binary files, and may present specific installation challenges. If you are

using a Windows platform, you may be able to use the Active State Programmer's Package Module (PPM), to automatically install XML::LibXML. For up-to-date information about platform-specific installation issues, refer to the Perl-XML FAQ [58].

The CPAN Shell can sometimes be a bit daunting, even to those with considerable Perl experience. If you find yourself having difficulties, check out the official CPAN documentation at: *http://www.perl.com/doc/manual/html/lib/CPAN.html.* Alternatively, check out Section 2.4, "Getting and Installing Modules," in *Perl in a Nutshell, 2nd edition* (O'Reilly, 2002).

5.2 The Simple API for XML (SAX)

5.2.1 Introduction to SAX

The Simple API for XML (SAX) is a standard event-based interface for parsing XML documents [71]. Unlike XML itself or the Document Object Model (DOM), SAX is not an official standard of any organization, such as the World Wide Web Consortium (W3C). Rather, SAX is a de facto standard, developed by a group of volunteers, freely available to the public, and widely implemented by dozens of XML parsers. SAX was originally designed for Java, but SAX and SAX-inspired implementations are now available for other languages, including Perl, Python, C++, Visual Basic, and Pascal (for a complete listing of SAX implementations see [71]).

The official SAX web site is available at: *http://www.saxproject.org.* Continued SAX development is now hosted at SourceForge.net.

5.2.2 SAX and Bioinformatics Applications

Using an event-based XML parsing interface like SAX provides a number of advantages, particularly when used for bioinformatics applications. First, SAX is a de facto standard and has wide support within the industry. Second, you can learn the SAX interface in one language and immediately apply it to a second language; for example, you can learn the Perl XML::SAX interface and apply your knowledge to the Java SAX interface.

Third, SAX is very fast and requires little memory. Unlike a tree-based interface, SAX will not store a complete representation of the XML document within memory. After an event has been reported to the application handler, the parser will immediately discard the event. This saves memory and is particularly important when parsing very large XML documents. For example, consider the implications of parsing the complete contents of UniProt [55], a comprehensive database of protein sequences and annotations. From the UniProt web site, you can download a complete snapshot of the database in UnitProt XML format. However, the complete snapshot consists of a single XML document, which is several hundreds of megabytes long. To parse this document via a tree-based

interface, you need enough memory to hold the entire tree, and you always run the risk of receiving "out of memory" errors. In contrast, SAX requires very little overhead and is capable of parsing arbitrarily large documents.

Although SAX itself is very memory efficient, keep in mind that your application handler can choose to store any and all XML events within its own internal data structures, and this will require its own set of memory requirements. You therefore need to carefully evaluate the specific requirements for your application. For example, if you want to parse the UnitProt XML file, and locate only a specific subset of proteins, your application handler can choose to record events specific to the target set, and discard the rest. Alternatively, if you want to import the UnitProt XML file into a set of relational database records, you only need to store one protein record at a time. After each record is committed to the database, you can purge your internal data structures, and move onto the next record.

The main disadvantage of SAX is that it does not provide facilities for easily modifying existing XML documents, or easily creating new XML documents from scratch. Some programmers also find the capturing and processing of individual SAX events counterintuitive, and prefer the simplicity of a tree traversal API.

5.2.3 SAX 2.0

The SAX API was originally developed by a group of volunteers, coordinating over the xml-dev mailing list. SAX 1.0 was originally released in May 1998 [65]. SAX 2.0.1 was officially released in January 2002. Several major changes were introduced in SAX 2.0, including complete support for XML Namespaces, standard methods for configuring XML parser features and properties, and support for SAX filters, enabling you to chain XML applications together [57; 64]. This book focuses on SAX 2.0 only. For details regarding SAX 1.0, refer to the SAX web site at: *http://www.saxproject.org/?selected=sax1*.

5.2.4 Introduction to XML::SAX

The Perl XML::SAX [70] module provides a complete implementation of the SAX 1.0 and 2.0 interfaces. The module is available for download from CPAN, and includes an XML parser written in Perl, called, XML::SAX::PurePerl [68]. PurePerl is considered quite slow, but it enables XML::SAX to work right out of the box on all platforms. XML::SAX also works with other faster SAX-compliant XML parsers, such as XML::LibXML [69], and provides a simple interface for selecting an XML parser at runtime.

To get started with XML::SAX, follow these three steps:

- Create a SAX event handler. As the XML parser reads a document, it will encounter specific XML constructs and notify your handler via callback methods. For example, when the XML parser encounters a new start element tag, it will call the event handler `start_element()` method, and pass information about the element. As a convenience, your event handler can extend `XML::SAX::Base`, and provide implementations of only those call-back methods that are of interest.
- Obtain an XML parser and register your event handler with the parser. XML::SAX provides a factory class, called `ParserFactory`, which will locate, instantiate, and initialize an XML

parser of your choosing. Later in this section, we will also explore options for selecting an XML parser at runtime.

- Initiate parsing by calling one of the `parse_xxx()` methods. For example, you can parse a local document by calling the `parse_uri()` method and supplying an absolute or relative path to the file. Alternatively, you can use the `parse_file()` method and pass in a stream or file handle, such as an `IO::File`. You can also use the `parse_string()` method and pass in the complete XML document as one string. In all cases, the XML parser will immediately start parsing the document of your choosing, and report each XML event to the registered handler.

To illustrate the basic concepts of XML::SAX, consider the sample Perl code in Listing 5.2.

The source code in Listing 5.2 instantiates a parser object via the `XML::SAX::ParserFactory` class. By default, the `ParserFactory` will look for a package variable to determine which parser to instantiate. For example, the following package variable will load the PurePerl SAX parser:

```
$XML::SAX::ParserPackage="XML::SAX::PurePerl";
```

Alternatively, the following package variable will load the LibXML SAX parser:

```
$XML::SAX::ParserPackage="XML::LibXML::SAX::Parser";
```

If no package variable is set, XML::SAX will automatically search all the directories in @INC in search of a SAX.ini file. The SAX.ini file uses a simple key=value format. For example, the following file specifies the LibXML SAX parser:

```
ParserPackage = XML::LibXML::SAX::Parser
```

If XML::SAX is unable to find a package variable or a SAX.ini file, it will automatically default to the PurePerl SAX parser.

Listing 5.2 First XML::SAX application

```
#!/usr/bin/perl
# Basic SAX Example.
# Author: Ethan Cerami
use strict;
use XML::SAX;
use ContentReporter;

# Create ElementReporter Instance
my $handler = ContentReporter->new;

# Obtain SAX Parser via ParserFactory
my $parser  = XML::SAX::ParserFactory->parser(Handler =>$handler);

# Parse TinySeq XML Document
my $element_counter = $parser->parse_uri("sample/ncbi.xml");

print "Total Number of Elements: $element_counter\n";
```

Tip: If you want to confirm which SAX parser is currently in use, you can use the very helpful `Data::Dumper` Perl module. Just obtain an XML parser via the `ParserFactory` and dump out its data. For example:

```
use Data::Dumper;
...
my $parser = XML::SAX::ParserFactory->parser(
         Handler =>$handler);
print Dumper ($parser);
```

The `Data::Dumper` will display all information about the parser, including the package name of the selected parser, and a list of all features and properties.

The bulk of the work in using XML::SAX goes into creating a custom SAX event handler. A sample event handler, called `ContentReporter`, is shown in Listing 5.3. Examine the code now, and we will survey its components below.

The `ContentReporter` in Listing 5.3 extends `XML::SAX:Base`, and selectively listens for four types of events: start_document, end_document, start_element, and end_element. The `ContentReporter` also keeps a running count of the number of elements encountered, and returns the total count to the main calling application. When we use this event handler on our sample GBSeq document in Listing 5.1, we get the following output:

```
Start Document
Start Element: GBSet
Start Element: GBSeq
Start Element: GBSeq_locus
End Element: GBSeq_locus
Start Element: GBSeq_length
End Element: GBSeq_length
Start Element: GBSeq_strandedness
...
End Document
Total Number of Elements: 320
```

Depending on the call-back method, you may or may not receive additional information about the event. For example, when the parser encounters the beginning of an XML document, it will call the `start_document()` method, but will not pass any additional event details. By contrast, when the parser encounters a new element, it will call the `start_element()` method, and pass along specific element details. Event details are specified as a hash reference with specific predefined keys. For example, the `start_element()` method receives a hash reference with several element specific keys, including "Name," "LocalName," and "Prefix." You can then reference these keys to display additional information about the event. For example:

```
sub start_element {
    my ($self, $element) = @_;
    my $name = $element->{"Name"};
    print "Start Element: $name\n";
}
```

The element hash reference also contains an optional "Attributes" key, which contains all attribute data associated with the element. For example, the following code will extract all attribute data and display it to the console:

```
# Extract All Attributes
my $attributes = $element->{"Attributes"};
foreach my $key (keys %$attributes) {
   my $attribute = $attributes->{$key};
   my $name = $attribute->{"Name"};
   my $value = $attribute->{"Value"};
   print "Attribute: $name --> $value\n";
}
```

Listing 5.3 ContentReporter.pm

```
#!/usr/bin/perl
# Basic SAX Handler
# Reports Basic Content Events
use strict;
package ContentReporter;

# Extend XML::SAX::Base
use base qw (XML::SAX::Base);

my $element_counter = 0;

# Report Start Document Event.
sub start_document {
    my ($self, $doc) = @_;
    print "Start Document\n";
}

# Report Start Element Events.
sub start_element {
    my ($self, $element) = @_;
    my $name = $element->{"Name"};
    print "Start Element: $name\n";
    $element_counter++;
}

# Report End Element Events.
sub end_element {
    my ($self, $element) = @_;
    my $name = $element->{"Name"};
    print "End Element: $name\n";
}

# Report End Document.
sub end_document {
    my ($self, $doc) = @_;
    print "End Document\n";
    return $element_counter;
}
1;
```

Table 5.1 provides a listing of the main methods in `XML::SAX::Base`. Note in particular that the `end_document()` method is the final method called, and that its return value is returned by the `parse_xxx()` methods. This provides a convenient mechanism to propagate data from the event handler back to the main calling application.

Table 5.1 Main methods of `XML::SAX::Base`

Method	Description
`attribute_decl ($self, $attribute_info)`	Indicates a DTD Attribute Declaration. The `$attribute_info` parameter is a hash reference containing the following keys: • Type: the attribute type, e.g. CDATA or ID • eName: the element name • aName: the attribute name • ValueDefault: attribute default flag, e.g. #REQUIRED or #IMPLIED • Value: default value, or undef if a default value is not supplied
`characters ($self, $text)`	Indicates a character event. The `$text` parameter is a hash reference containing a single key: • Data: contains the character content. Given a string of XML text, each SAX parser is free to report character events as it sees fit. For example, one parser may report the entire text string via a single call to `characters()`; a second parser may split the string and call `characters()` twice. Given this flexibility, it is important that your handler provide some type of character buffering
`comment ($self, $comment)`	Indicates an XML comment. The `$comment` parameter is a hash reference containing a single key: • Data: contains the comment text
`element_decl ($self, $element_info)`	Indicates a DTD Element Declaration. The `$element_info` parameter is a hash reference containing the following keys: • Name: name of the element • Model: content model of the element
`end_cdata ($self)`	Indicates the end of a CDATA section
`end_document ($self)`	Indicates the end of an XML document. The return value of `end_document()` is returned by the `parse_xxx()` methods, and therefore provides a convenient mechanism for propagating data from the event handler back to the main calling application
`end_element ($self, $element)`	Indicates the end of an XML element. The `$element` parameter is a hash reference containing the same keys as those defined in `start_element()`. See `start_element()` for details
`end_prefix_mapping ($self, $namespace_info)`	Indicates the end scope of a namespace declaration. The `$namespace_info` parameter is a hash reference containing the following keys: • Prefix: the namespace prefix, e.g., "psi" • NamespaceURI: the namespace URI, e.g., "net:sf:psidev:mi"
`processing_instruction ($self, $pi)`	Indicates a processing instruction. The `$pi` parameter is a hash reference containing the following keys: • Target: the target of the processing instruction • Data: the complete text of the processing instruction
`set_document_locator ($self, $doc_locator)`	Sets a document locator object. This is usually the very first method called, directly before a call to `start_document()`. If you want to determine the location of all subsequent SAX events, store the `$doc_locator` object locally and reference it within other SAX call-back methods. The `$doc_locator` parameter is a hash reference, containing the following keys: • ColumnNumber: column number where the event occurred • LineNumber: line number where the event occurred

Table 5.1 (*cont.*)

Method	Description
	• SystemId: the system identifier of the current document or undef, if it is not defined • PublicId: the public identifier of the current document or undef, if it is not defined Note that SAX parsers are strongly encouraged to supply a document locator, but are not required to do so
`start_cdata ($self)`	Indicates the start of a CDATA section. The actual content of the CDATA section is subsequently reported via the `characters()` method
`start_document ($self)`	Indicates the start of an XML document
`start_element ($self, $element)`	Indicates the start of an XML element. The `$element` parameter is a hash reference containing the following keys: • Prefix: the namespace prefix of the element, e.g., "psi" • LocalName: the local name of the element. This is the name of the element, without its namespace prefix, e.g., "interaction" • Name: a fully qualified element name. This is the name of the element with its namespace prefix, e.g., "psi:interaction" • NamespaceURI: the namespace URI of this element • Attributes: a hash reference containing all the element's attributes If the element has attributes, the Attributes hash reference will contain one key for each attribute. The key is specified in the following form: "{NamespaceURI}LocalName". If the attribute is not associated with any namespace, it will have an empty NamespaceURI, e.g., "{ }LocalName". Each attribute will in turn contain the following keys: • Prefix: the namespace prefix of the attribute • LocalName: the local name of the attribute • Value: the attribute value • Name: a fully qualified attribute name • NamespaceURI: the namespace URI of the attribute
`start_prefix_mapping ($self, $namespace_info)`	Indicates the beginning scope of a namespace declaration. The `$namespace_info` parameter is a hash reference containing the following keys: • Prefix: the namespace prefix, e.g., "psi" • NamespaceURI: the namespace URI, e.g., "net:sf:psidev:mi"
`xml_decl ($self, $declaration)`	Indicates an XML declaration. The `$declaration` parameter is a hash reference containing the following keys: • Version: XML version, e.g., "1.0" • Encoding: character encoding, e.g., "UTF-8"

Error Handling

It is important to note that if your XML parser encounters an error in well-formedness, the parser will consider this a fatal error and stop program execution via a call to `die()`. (Recall from Chapter 2 that an XML document is considered well-formed if it follows the basic rules of document construction, e.g., all start tags must have matching end tags, elements must be properly nested, attributes must appear in quotes, etc.) If you want to prevent your program from dying, you can wrap your call to `parse_xxx()` inside an eval block. For example:

```
eval {
    $parser->parse_uri("sample/ncbi.xml");
};
```

```
if ($@) {
   my $message = $@->{"Message"};
   my $line_number = $@->{"LineNumber"};
   print "Error! -->  $message\n";
   print "Error Occurred at line number: $line_number\n";
}
```

Note that the $@ hash reference contains a number of predefined keys, such as "Message," "LineNumber," and "ColumnNumber," allowing you to access specific details about the error.

5.2.5 Using NCBI EFetch and XML::SAX

Now that you understand the basics of `XML::SAX` , you can apply this knowledge to dynamically retrieve and parse sequence data from NCBI. Fortunately for us, NCBI provides a web service, called EFetch that simplifies the process of retrieving sequence records. EFetch is actually an example of a REST-based web service (for details on REST-based web services, refer to Chapter 9). In a nutshell, client applications connect to EFetch via HTTP and specify search criteria with a set of URL parameters. Based on the search criteria, the EFetch service will connect to the NCBI Entrez back-end database system, find a matching record, and return the requested record in the format of your choosing. EFetch currently provides access to several NCBI Entrez databases, including sequence, literature, and taxonomy databases; and can return data in several data formats, including text, HTML, ASN.1 and XML. If you are eager to try out a few sample EFetch requests, refer to Table 5.2.

As of this writing, the base URL for connecting to EFetch is:

http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi

To retrieve a specific nucleotide sequence record, you must append a database parameter and an ID parameter, which uniquely identifies the record. For example, the following URL retrieves the complete genome record for the SARS coronavirus, formatted in the GenBank flat file format:

Table 5.2 Example EFetch queries

How to Retrieve a Nucleotide Record:
Example #1: Retrieves information regarding the BRCA2 gene in Human, and formats the results in TinySeq XML:
http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db= nucleotide&id=U43746&rettype=fasta&retmode=xml
Example #2: Retrieves information regarding the BRCA2 gene in Human, and formats the results in GenBank XML:
http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nucleotide&id=U43746&rettype=gb&retmode=xml
How to Retrieve a Protein Record:
Example: Retrieves information regarding the BRCA2 protein in Human, and formats the results in GenPept XML:
http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=protein&id=AAB07223.1&rettype=gp&retmode=xml
How to Retrieve a Literature Record:
Example: Retrieves citation and abstract information regarding PMID: 14597658, and formats the results in XML:
http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=pubmed&id=14597658&retmode=xml
How to Retrieve a Taxonomy Record:
Example: Retrieves the species name for NCBI Taxonomy ID: 7227. In this case, E-Fetch returns a single string: "Drosophila melanogaster".
http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=taxonomy&id=7227&report=brief

http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nucleotide&rettype=gb&retmode=text&id=30271926

In the URL above, the *db* parameter specifies the NCBI nucleotide database, *rettype* specifies the GenBank flat file format, *retmode* specifies text content, and *id* specifies the NCBI GI number for the SARS virus. Conveniently, the *id* parameter accepts both NCBI GI numbers and NCBI accession numbers.

For XML content, set the *retmode* parameter to "xml." For example, to retrieve data in the NCBI TinySeq XML format, set *rettype=fasta* and *retmode=xml.* To retrieve data in the more comprehensive NCBI GBSeq XML, set *rettype=gb* and *retmode=xml* . For example, the following URL retrieves the same SARS virus record, but this time it is formatted in GBSeq XML:

http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nucleotide&rettype=gb&retmode=xml&id=30271926

Complete details regarding NCBI EFetch are available online at:
http://eutils.ncbi.nlm.nih.gov/entrez/query/static/efetchseq_help.html.

Our goal is to write a Perl program capable of automatically retrieving sequence data from EFetch and extracting a small subset of the XML content for display to the console. The program expects a single command line argument, indicating an NCBI GI number or accession number. A sample run of the application is shown below:

```
>fetch.pl NC_004718
Downloading XML from NCBI E-Fetch
Using URL: http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.
fcgi?db=nucleotide&rettype=gb&retmode=xml&id=30271926
Definition: SARS coronavirus, complete genome
Accession: NC_004718
Locus: NC_004718
Organism: SARS coronavirus
Sequence (0..20):  ATATTAGGTTTTTACCTACC...
```

Source code for the Perl fetcher is shown in Listings 5.4 and 5.5. Examine the code now and we will describe its main components below.

As in our first SAX application, the fetch application consists of two parts: a main application, which initiates parsing (Listing 5.4), and a SAX event handler (Listing 5.5). The main application uses the World Wide Web library for Perl (LWP) [60] to connect to NCBI EFetch and retrieve the specified sequence record. It also obtains an XML parser via the SAX factory, and initiates parsing via the `parse_string()` method. The `parse_string()` method returns an associative array, which we then print to the console.

The NcbiHandler.pm module listens for specific SAX events, and selectively stores specific GBSeq elements in an internal associative array. There are a few important items to note. First, the `characters()` method uses a character buffer. This is important because SAX parsers are free to perform character "chunking"—for example, one SAX parser may report a line of text via a single call to `characters()` , whereas a second SAX parser may break the line into two "chunks" and report it via two calls to `characters()` . Since there is no way to know ahead of time which chunking method the parser will use, it is always safest to assume multiple calls to `characters()` and to append to a character buffer each time. Second, the `end_element()` method is used to

Listing 5.4 Parsing NCBI EFetch data via the SAX API

```
#!/usr/bin/perl
# Fetches NCBI XML from the NCBI E-Fetch Utility.
# Author: Ethan Cerami
use XML::SAX;
use LWP::Simple;
use NcbiHandler;
use strict;

# Display Command Line Usage
if (@ARGV == 0) {
    print "Usage: fetch.pl ncbi_identifier (NCBI GI or Accession
      Number)\n";
die "Example: fetch.pl 30271926\n";
}

# Download File from NCBI e-Fetch; uses LWP Module
my $ncbi_url = get_ncbi_url($ARGV[0]);
print "Downloading XML from NCBI E-Fetch\n";
print "Using URL: $ncbi_url\n";
my $xml_doc = LWP::Simple::get($ncbi_url);

# Parse XML Document
my $handler = NcbiHandler->new;
my $parser = XML::SAX::ParserFactory->parser(Handler =>$handler);
my %data = $parser->parse_string($xml_doc);

# Output Results of Parsing
my $sequence = $data{"GBSeq_sequence"};
$sequence = substr ($sequence, 0, 20);
print "Definition: ",  $data{"GBSeq_definition"};
print "\nAccession: ",  $data{"GBSeq_primary-accession"};
print "\nLocus: ", $data{"GBSeq_locus"};
print "\nOrganism: ", $data{"GBSeq_organism"};
print "\nSequence (0..20): $sequence...\n";

# Gets NCBI Identifier from user, and returns an absolute URL
# to the NCBI E-Fetch Utility.
sub get_ncbi_url {
    my $id = $_[0];

    # Set Base URL for NCBI E-Fetch
    my $baseurl = "http://eutils.ncbi.nlm.nih.gov/entrez/eutils/"
    ."efetch.fcgi?db=nucleotide&rettype=gb&retmode=xml&id=";

    return ($baseurl . $id);
}
```

Listing 5.5 NcbiHandler.pm

```
#!/usr/bin/perl -w
# Parses NCBI GBSeq XML Documents, and extracts only
# selected elements.
package NcbiHandler;
use strict;

# Extend XML::SAX::Base
use base qw (XML::SAX::Base);

my ($current_text, %data);

# Report Start Element Events.
# Each time we get a start element event,
# reset the character buffer.
sub start_element {
    my ($self, $element) = @_;
    $current_text = "";
}

# Selectively store element information.
sub end_element {
    my ($self, $element) = @_;
    my $name = $element->{"Name"};
    if ($name eq "GBSeq_locus"
           || $name eq "GBSeq_primary-accession"
           || $name eq "GBSeq_definition"
           || $name eq "GBSeq_organism"
           || $name eq "GBSeq_sequence" ) {
       $data{$name} = $current_text;
    }
}

# Keep Character Buffer.
sub characters {
    my ($self, $characters) = @_;
    $current_text .= $characters->{"Data"};
}

# Return Associative Array to main application.
sub end_document {
    my ($self, $doc) = @_;
    return %data;
}
1;
```

selectively filter for specific GBSeq elements. For those specific elements of interest, we store the current character buffer into an associative array and use the element name as a hash key. We subsequently return the associative array to the main calling application by returning it from the `end_document()` method.

5.3 The Document Object Model (DOM)

The Document Object Model (DOM) is a standard tree-based interface for reading, modifying, and creating XML documents. The DOM is an official recommendation of the W3C, and the DOM API is specified in the Object Management Group Interface Definition Language (OMG IDL). This enables the DOM API to be both platform and language independent. DOM implementations are available in numerous programming languages, including Perl, Java, JavaScript, VBScript, C/C++, and Python. In this section, we provide an introduction to the Perl `XML::LibXML` module, and illustrate its support for the DOM standard. We also revisit the NCBI EFetch service, and re-create the same functionality as our earlier SAX application. This enables you to directly compare and contrast the DOM approach with the SAX approach.

5.3.1 DOM Traversal with XML::LibXML

The `XML::LibXML` Perl module provides an interface to the very popular libxml parser, an XML parser written in C and developed for the Linux Gnome project. The libxml parser itself is packed with numerous features, including support for XML Namespaces, SAX, DOM, XPath, XPointer, and XInclude. For our purposes, we will be focusing exclusively on the DOM implementation provided by libxml. If you are interested in the other features of libxml, check out the main libxml web site at: *http://xmlsoft.org*.

Let us jump right in with our first DOM example. Listing 5.6 provides an example application, which will traverse through all the elements in our sample NCBI XML document. Examine the code now, and we will describe its main components below.

There are several important elements to note about the code in Listing 5.6. First, we instantiate a new `LibXML` parser object and direct the parser to parse a local file:

```
# Instantiate LibXML Parser
my $parser = XML::LibXML->new();

# Parse Sample Document
my $doc = $parser->parse_file("sample/ncbi.xml");
```

The `parse_file()` method returns a DOM `Document` object. This document object contains a complete tree representation of our XML document. To begin tree traversal, we request the root document element:

```
my $root = $doc->getDocumentElement();
```

We then pass this root element to the recursive `traverse_node()` method. In the DOM data model, all XML constructs, e.g., elements, attributes, text data, and processing instructions, are represented as DOM nodes, and all nodes provide a number of very useful attributes/methods. For example, you can retrieve the node name or node type:

```
my $node_name = $node->nodeName;
my $node_type = $node->nodeType;
```

Depending on the node type, our code takes different actions. For example, if we encounter a text node, we extract the embedded text content. If we encounter an element node, we retrieve a list of all its child nodes and pass these nodes recursively to the `traverse_node()` method:

```
my @children = $node->childNodes();
foreach my $child (@children) {
   traverse_node ($child, $indent+1);
}
```

Listing 5.6 First DOM example

```
# DOM Traversal Example
# Illustrates Basic DOM Functionality

use XML::LibXML;
use strict;

# Instantiate LibXML Parser
my $parser = XML::LibXML->new();

# Parse Sample Document
my $doc = $parser->parse_file("sample/ncbi.xml");

# Get Document Root Element
my $root = $doc->getDocumentElement();

# Initiate DOM Traversal at root
traverse_node ($root, 0);

# Recursive Function to Traverse DOM Tree
sub traverse_node {
    my ($node, $indent) = @_;

    # Indent X Characters
    my $line = "." x ($indent * 2);
    print "$line";

    # Get Node Name and Type
    my $node_name = $node->nodeName;
    my $node_type = $node->nodeType;

    # Take Different Actions depending on node type
    if ($node_type == 3) {
            # This is a text node
            my $text_content = $node->textContent;
            $text_content =~ s/\n/[new line]/;
            print "$node_name: $text_content\n";
    } elsif ($node_type == 1) {
           # This is an Element Node
           print "Element: $node_name\n";
           # Iterate through all Child Nodes
           my @children = $node->childNodes();
           foreach my $child (@children) {
                   traverse_node ($child, $indent+1);
           }
    }
}
```

By examining each element, and recursively exploring each of its child nodes, our sample application is capable of traversing through the entire XML document tree. An excerpt from this traversal is shown below:

```
Element: GBSet
..text: [new line]
..Element: GBSeq
....text: [new line]
....Element: GBSeq_locus
......text: BC034957
....text: [new line]
....Element: GBSeq_length
......text: 2547
[output continues...]
```

If you want to determine if an element has attributes, you can use the `hasAttributes` query method, and then retrieve those attributes via the `attributes` property. For example, the following code excerpt outputs all element attributes to the console:

```
if ($node->hasAttributes) {
   my @attributes = $node->attributes;
   foreach my $attribute (@attributes) {
           my $name = $attribute->nodeName;
           my $value = $attribute->value;
           print "Attribute Name: $name --> Value: $value\n";
   }
}
```

The DOM API also supports several methods for finding specific subelements. For example, the `getChildrenByTagName()` method will find all direct children with the specified tag name. By contrast, the `getElementsByTagName()` method will recursively search all descendants of the current node, and return all descendants with the specified tag name. The LibXML module also provides support for XPath, a W3C specification that enables you to pinpoint specific elements or sets of elements within an XML document. For example, the code snippet below uses the XPath `find` feature to extract two specific GBSeq elements:

```
my $doc = $parser->parse_file("sample/ncbi.xml");
my $locus = $doc->find("/GBSet/GBSeq/GBSeq_locus");
my $def = $doc->find("/GBSet/GBSeq/GBSeq_definition");
print "Locus: $locus\n";
print "Definition: $def\n";
```

The complete LibXML DOM API is quite large, and we could not hope to cover it in its entirety in this chapter. For complete documentation on all relevant classes and methods, refer to the LibXML documentation, available online at: *http://search.cpan.org/dist/XML-LibXML*. If you are working extensively with LibXML, you may find it useful to print out the documented API for `XML::LibXML::Node` , the base class used to represent all DOM nodes, and `XML::LibXML::Element` , the class used to represent element nodes.

5.3.2 Validating XML Documents with XML::LibXML

The LibXML Perl module provides built-in support for validating XML documents against DTDs. To turn XML validation on, simply pass a true value to the parser `validation()` method, and then initiate parsing. If LibXML encounters an error in well-formedness or validity, it will immediately `die()` and report the error to the console. If you want to prevent your program from exiting, you can wrap the parse call in an eval block. For example:

```
my $parser = XML::LibXML->new();
$parser->validation(1);

eval {
    my $doc = $parser->parse_file($file_name);
};

if ($@) {
   print "Error --> $@\n";
} else {
   print "----> OK\n";
}
```

5.3.3 Creating New Documents with XML::LibXML

In addition to reading in XML documents, the Document Object Model also provides convenient mechanisms for modifying existing documents or creating new XML documents from scratch. For example, the code in Listing 5.7 uses the DOM API to create an entirely new document in the NCBI TinySeq XML format.

To create a new XML document, you must first instantiate a `Document` object:

```
my $document = XML::LibXML::Document->new ();
```

You then need to create root element, and add this to the document:

```
my $root = $document->createElement ("TSeq");
$document->addChild ($root);
```

You can then proceed to create new elements, and add these to the root element. Note that if you want to add text to an element, you must first create a text node, and then add this to the element node. This operation is automatically performed by the `appendTextNode()` method. In the same manner, to create an element with attributes, you must first create one or more attribute objects, and then add each attribute to the element node. The DOM API provides dozens of other methods for creating new nodes, removing nodes, and copying nodes. However, the scope of the complete API is beyond the scope of what we hope to cover here. For the complete API, refer to the LibXML API documentation, available online at: *http://search.cpan.org/dist/XML-LibXML*.

5.3.4 Using NCBI EFetch and XML::LibXML

As our final topic, we revisit the NCBI EFetch service. Our goal is to retain the exact same functionality as our first EFetch application (see Section 5.2.5), but to replace the SAX code with

Listing 5.7 Creating a TinySeq XML document via the DOM API

```
# Creates a new TinySeq XML Document via the DOM API.
use XML::LibXML;
use strict;

# Instantiate New Document Object
my $document = XML::LibXML::Document->new ();

# Create Root Element
my $root = $document->createElement ("TSeq");
$document->addChild ($root);

# Add Sequence Type with Attribute
my $seq_type = $document->createElement ("TSeq_seqtype");
my $attribute = $document->createAttribute("value", "nucleotide");
$seq_type->addChild ($attribute);
$root->addChild ($seq_type);

# Add Other Sub-elements
add_element ($document, $root, "TSeq_gi", "11497606");
add_element ($document, $root, "TSeq_sid", "ref| NM_001464.2| ");
add_element ($document, $root, "TSeq_taxid", "9606");
add_element ($document, $root, "TSeq_orgname", "Homo sapiens");
add_element ($document, $root, "TSeq_defline",
    "Homo sapiens a disintegrin and metalloproteinase domain 2 "
    . "(fertilin beta) (ADAM2), mRNA");
add_element ($document, $root, "TSeq_length", "2659");
add_element ($document, $root, "TSeq_sequence", "CATCTCGCACTTC...");

# Convert to XML String, with indentation
my $xml = $document->toString(1);
print "XML Document:\n$xml";

# Adds New Element with Single Text Value
sub add_element {
    my ($document, $parent, $element_name, $text_str) = @_;
    my $child = $document->createElement ($element_name);
    $child->appendTextNode ($text_str);
    $parent->addChild ($child);
}
```

DOM code. By comparing the two examples, you can therefore directly compare the SAX and DOM interfaces and gain insight into both approaches.

The complete source code for our new fetch application is shown in Listing 5.8.

As in our first DOM example, we parse an NCBI XML document and immediately extract its root element. However, instead of traversing the entire XML object tree, we now selectively traverse the tree in search of five specific GBSeq elements. To do so, we first obtain the `GBSeq` element:

```
my @seq_children = $root->getElementsByTagName("GBSeq");
my $seq_node = $seq_children[0];
```

Listing 5.8 Parsing NCBI EFetch data via the DOM API

```
# Fetches NCBI XML from the NCBI E-Fetch Utility.
# Author: Ethan Cerami
use XML::LibXML;
use LWP::Simple;
use strict;

# Display Command Line Usage
if (@ARGV == 0) {
    print "Usage: fetch.pl ncbi_identifier (NCBI GI or Accession
     Number)\n";
    die "Example: fetch.pl 30271926\n";
}

# Download File from NCBI e-Fetch; uses LWP Module
my $ncbi_url = get_ncbi_url($ARGV[0]);
print "Downloading XML from NCBI E-Fetch\n";
print "Using URL: $ncbi_url\n";
my $xml_doc = LWP::Simple::get($ncbi_url);

# Instantiate LibXML Parser and Parse Document
my $parser = XML::LibXML->new();
my $doc = $parser->parse_string($xml_doc);

# Get Document Root Element
my $root = $doc->getDocumentElement();

# Get First GBSeq Element
my @seq_children = $root->getElementsByTagName("GBSeq");
my $seq_node = $seq_children[0];

# Extract Individual Elements
my $def_line = get_element_text ($seq_node, "GBSeq_definition");
my $acc = get_element_text ($seq_node, "GBSeq_primary-accession");
my $locus = get_element_text ($seq_node, "GBSeq_locus");
my $organism = get_element_text ($seq_node, "GBSeq_organism");
my $sequence = get_element_text ($seq_node, "GBSeq_sequence");
$sequence = substr ($sequence, 0, 20);

print "Definition: $def_line\n";
print "Accession: $acc\n";
print "Locus: $locus\n";
print "Organism: $organism\n";
print "Sequence (0..20): $sequence...\n";

# Gets Element Text
sub get_element_text {
    my ($node, $target_name) = @_;

    # getChildrenByTagName gets direct children only.
    # getElementsByTagName gets all descendants.
```

Listing 5.8 (*cont.*)

```
        my @elements = $node->getChildrenByTagName($target_name);

        if (@elements) {
                my $element = $elements[0];
                my $text = $element->textContent;
        } else {
                return "Not Available";
        }
}

# Gets NCBI Identifier from user, and returns an absolute URL
# to the NCBI E-Fetch Utility.
sub get_ncbi_url {
    my $id = $_[0];

    # Set Base URL for NCBI E-Fetch
    my $baseurl = "http://eutils.ncbi.nlm.nih.gov/entrez/eutils/"
    . "efetch.fcgi?db=nucleotide&rettype=gb&retmode=xml&id=";

    return ($baseurl . $id);
}
```

We then selectively search for direct children with specific tag names. To do so, the local `get_element_text()` method uses the DOM `getChildrenByTagName()` method to find direct children with the specified tag name. If any matching children are found, we take the first matching child and immediately return its text content. We are therefore able to easily extract any piece of GBSeq data and immediately display it to the console.

In conclusion, Perl provides excellent support for XML. In this chapter, we have discussed the fundamental differences between tree-based and event-based XML parsers, and have illustrated these differences by exploring the SAX and DOM interfaces. Event-based parsers, such as SAX, are generally faster and require less memory than comparable tree-based parsers. However, tree-based parsers provide random access to any node or branch in the XML document and also provide facilities for modifying or creating new documents. If you have intense performance requirements, or are working with very large documents, you may have no choice but to use an event-based parser. However, for moderate-sized XML documents, you may find a tree-based interface easier to use. In either case, by sticking to well-defined public standards, such as SAX or DOM, you can more easily apply your XML knowledge to other programming languages, such as C++ or Java.