

# Bio::SearchIO HOWTO

**Jason Stajich**  
Duke University<sup>1</sup>  
University Program in Genetics<sup>2</sup>  
Center for Genome Technology<sup>3</sup>

Duke University Medical Center  
Box 3568  
Durham, North Carolina 27710-3568  
USA  
[jason-at-bioperl.org](mailto:jason-at-bioperl.org)

This is a HOWTO written in DocBook (SGML) for the reasoning behind the creation of the Bio::SearchIO system, how to use it, and how one goes about writing new adaptors to different output formats. We will also describe how the Bio::SearchIO::Writer modules work for outputting various formats from Bio::Search objects.

## Table of Contents

Background .....	3
Design .....	3
Parsing with Bio::SearchIO.....	3
Implementation.....	4
Writing and formatting output.....	5
Extending SearchIO.....	6



## Background

One of the most common and necessary tasks in bioinformatics is parsing analysis reports so that one can write programs which can help interpret the sheer volume of data that can be produced by processing many sequences. To this end the Bioperl project has produced a number of parsers for the ubiquitous BLAST report. Steve Chervitz wrote one of the first Bioperl modules for BLAST called `Bio::Tools::Blast`. Ian Korf allowed us to import and modify his `BPlite` (*Blast Parser*) `Bio::Tools::BPlite` module into Bioperl. This is of course in a sea of BLAST parsers that have been written by numerous people, but we will only cover the ones associated directly with the Bioperl project in this document. One of the reasons for writing yet another BLAST parser in the form of `Bio::SearchIO` is that even though both `Bio::Tools::Blast` and `Bio::Tools::BPlite` did their job correctly, could parse WU-BLAST and NCBI-BLAST output, they did not adequately genericize what they were doing. By this we mean everything was written around the BLAST format and was not easily applicable to parsing say, FastA alignments or a new alignment format. One of the powerful features of the Object-Oriented framework in Bioperl is the ability to read in say, a sequence file, in different formats or from different data sources like a database or XML-flatfile, and have the program code process the sequences objects in the same manner. We wanted to have this capability in place for analysis reports as well and thus the generic design of the `Bio::SearchIO` module.

## Design

The `Bio::SearchIO` system was designed with the following assumptions. That all reports parsed with it could be separated into a hierarchy of components. The `Result` which is the entire analysis for a single query sequence. Multiple results can be concatenated together into a single file (i.e. running `blastall` with a fasta database as the input file rather than a single sequence). Each result is a set of `Hits` for the query sequence. `Hits` are sequences in the searched database which could be aligned to the query sequence and met the minimal search parameters such as e-value threshold. Each `Hit` has one or more `High-scoring segment Pairs (HSP)s` which are the alignments of the query and hit sequence. So each `Result` has a set of one or more `Hits` and each `Hit` has a set of one or more `HSPs`. This relationship can be used to describe results from all pairwise alignment programs including BLAST, FastA, and implementations of the Smith-Waterman and Needleman-Wunsch algorithms.

A design pattern, called `Factory`, is utilized in object oriented programming to separate the entity which process data from objects which will hold the information produced. In the same manner that the `Bio::SeqIO` module is used to parse different file formats and produces objects which are `Bio::PrimarySeqI` compliant, we have written `Bio::SearchIO` to produce the `Bio::Search` objects. Sequences are a little less complicated so there is only one primary object (`Bio::PrimarySeqI`) which `Search` results need three main components to represent the data processed in a file: `Bio::Search::Result::ResultI` (top level results), `Bio::Search::Hit::HitI` (hits) and `Bio::Search::HSP::HSPI` (HSPs). The `Bio::SearchIO` object is then a factory which produces `Bio::Search::Result::ResultI` objects and the `Bio::Search::Result::ResultI` objects contain information about the query, the database searched, and the full collection of `Hits` found for the query.

## Parsing with `Bio::SearchIO`

This section is going to describe how to use the `SearchIO` system to process reports. We'll describe BLAST and FastA reports. The idea is that once you understand the methods associated with the objects you won't need to know anything special about new parsers.

Here's an example which processes a BLAST report finding all the hits where the HSPs are > 100 residues and the percent identity is < 75 percent.

```

use strict;
use Bio::SearchIO;

my $in = new Bio::SearchIO(-format => 'blast',
                           -file   => 'report.bls');
while( my $result = $in->next_result ) {
  while( my $hit = $result->next_hit ) {
    while( my $hsp = $hit->next_hsp ) {
      if( $hsp->length('total') > 100 &&
          $hsp->percent_identity >= 75 ) {
        print "Hit= ", $hit->name,
              ", len=", $hsp->length('total'),
              ", percent_id=", $hsp->percent_identity, "\n";
      }
    }
  }
}

```

You can read up on the Bio::Search::HSP::HSPI object that is produced by Bio::SearchIO to see what other methods besides length and percent\_identity are supported. The best place for this <http://doc.bioperl.org> which provides HTML-ified version of the Perl POD (Plain Old Documentation) that is embedded in every (well written) Perl module.

Note. There is some confusion often associated by the objects because of the nature of the active development of this system. Steve Chervitz and myself (Jason) have implemented different parsers in this system. Steve created the psiblast parser (which does parse regular blast files too) and a host of objects named Bio::Search::XXX::BlastXXX where XXX is HSP, Hit, and Result. These objects are created by his Bio::SearchIO::psiblast implementation. The objects I have created are called Bio::Search::XXX::GenericXXX where again XXX is HSP, Hit, and Result. Because of some of the assumptions made in Steve's implementation and his utilization of what is known as 'lazy parsing', it is probably not going to be very easy to maintain his system without his help. While I have tried (perhaps unsuccessfully?) to make my implementations much easier to follow because all the parsing is done in one module. The important take home message is that you cannot assume that methods in the BlastXXX objects are in fact implemented by the GenericHSP objects. More than likely the BlastXXX objects will be deprecated and dismantled as their functionality is ported to the GenericHSP objects.

## Implementation

This section is going to describe how the SearchIO system was implemented, it is probably not necessary to understand all of this unless you are curious or want to implement your own Bio::SearchIO parser. We have utilized an event-based system to process these reports. This is analagous to the SAX (Simple API for XML) system used to process XML documents. Event based parsing can be simply thought of as simple start and end events. When you hit the beginning of a report a start event is thrown, when you hit the end of the report an end event is thrown. So the report events are paired, and everything else that is thrown in between the paired start and end events is related to that report. Another way to think of it is as if you pick a number and color for a card in a standard deck. Let's say you pick red and 2. The you start dealing cards from our deck and pile them one on top of each other. When you see your first red 2 you start a new pile, and start dealing cards onto that pile until you see the next red 2. Everything in your pile that happened between when you saw the beginning red 2 and ending red 2 is data you'll want to keep and process. In the same way all the events you see between a pair of start and end events (like 'report' or 'hsp') are data associated with object or child object in its hierarchy. A listener object processes all of these events, in our example the listener is the table where the stack of cards is sitting, and later it is the hand which moves the pile of cards when a new

stack is started. The listener will take the events and process them. We've neglected to tell you of a third event that is thrown and caught. This is the characters event in SAX terminology which is simply data. So one sends a start event, then some data, then an end event. This process is analagous to a finite state machine in computer science (and I'm sure the computer scientists reading this right are already yawning) where what we do with data received is dependent on the state we're in. The state that the listener is in is affected by the events that are processed.

A small caveat, in an ideal situation a processor would throw events and not need to keep any state about where it is, it would just be processing data and the listener would manage the information and state. However, a lot of the parsing of these human readable reports requires contextual information to apply the correct regular expressions. So in fact the event thrower has to know what state it is in and apply different methods based on this. In contrast the XML parsers simply keep track of what state they are in, but can process all the data with the same system of reading the tag and sending the data that is inbetween the XML start and end tags.

All of this framework has been built up so to implement a new parser one only needs to write a module that produces the appropriate start and end events and the existing framework will do the work of creating the objects for you. Here's how we've implemented event-based parsing for Bio::SearchIO. The Bio::SearchIO is just the front-end to this process, in fact the processing of these reports is done by different modules in the Bio/SearchIO directory. So if you look at your bioperl distribution at the modules in Bio/SearchIO you'll see modules in there like: blast.pm, fasta.pm, blastxml.pm, SearchResultEventBuilder.pm, EventHandlerI.pm (depending on what version of the toolkit there may be more modules in there). There is also a SearchWriterI.pm and Writer directory in there but we'll save that for later. If you don't have the distribution handy you can navigate this at the bioperl CVSweb page<sup>5</sup>.

Lets use the blast.pm module as an example to describe the relationship of the modules in this dir (could have subsituted any of the other format parsers like fasta.pm or blastxml.pm - these are always lowercase for historical reasons). The module has some features you should look for - the first is the hash in the BEGIN block called %MAPPING. This key value pairs here are the shorthand for how we map events from this module to general event names. This is only necessary because if we have an XML processor (see the blastxml.pm module) the event names will be the same as the XML tag names (like <Hsp\_bit-score> in the NCBI BLAST XML DTD). So to make this general we'll make sure all of the events inside our parser map to the values in the %MAPPING hash - we can call them whatever we want inside this module. Some of the events map to hash references (like Statistics\_db-len) this is so we can map multiple values to the same top-level attribute field but we know they will be stored as a hash value in the subsequent object (in this example, keyed by the name 'dbentries'). The capital "RESULT", "HSP", or "HIT" in the value name allow us to encode the event state in the event so we don't have to pass in two values. It also easy for someone to quickly read the list of events and know which ones are related to Hits and which ones are related to HSPs. The listener in our architecture is the Bio::SearchIO::SearchResultEventBuilder. This object is attached as a listener through the Bio::SearchIO method add\_EventListener. In fact you could have multiple event listeners and they could do different things. In our case we want to create Bio::Search objects, but an event listener could just as easily be propigating data directly into a database based on the events. The SearchResultEventBuilder takes the events thrown by the SearchIO classes and builds the appropriate Bio::Search::HSP:: object from it.

Sometimes special objects are needed that are extensions beyond what the GenericHSP or GenericHit objects are meant to represent. For this case we have implemented Bio::SearchIO::SearchResultEventBuilder so that it can use factories for creating its resulting Bio::Search objects - see Bio::SearchIO::hammer\_initialize method for an example of how this can be set.

## Writing and formatting output

Often people want to write back out a BLAST report for users who are most comfortable with that output or if you want to visualize the context of a weakly aligned region to use human intuition to score the confidence of a putative homologue. Bio::SearchIO is for parsing in the data and Bio::SearchIO::Writer is for outputting the information. The simplest way to output data as a pseudo-BLAST HTML format is as follows.

```
my $writerhtml = new Bio::SearchIO::Writer::HTMLResultWriter();
my $outhtml = new Bio::SearchIO(-writer => $writerhtml,
    -file => ">searchio.html");
# get a result from Bio::SearchIO parsing or build it up in memory
$out->write_result($result);
```

If you wanted to get the output as a string rather than write it out to a file, simply use the following.

```
$writerhtml->to_string($result);
```

The HTMLResultWriter supports setting your own remote database url for the sequence links in the event you'd like to point to your own SRS or local HTTP based connection to the sequence data, simply use the `remote_database_url` method which accepts a sequence type as input (protein or nucleotide).

You can also override the `id_parser()` method to define what are the unique IDs from these sequence ids in the event you would like to use something other than accession number that is gleaned from the sequence string.

If your data is instead stored in a database you could build the Bio::Search objects up in memory directly from your database and then use the Writer object to output the data. Currently there is also a Bio::SearchIO::Writer::TextResultWriter which supports writing BLAST textfile output.

## Extending SearchIO

The framework for Bio::SearchIO is just a starting point for parsing these reports and creating objects which represent the information. If you would like to create your own set of objects which extend the current functionality we have built the system so that it will support this. For example, if you've built your own HSP object which supports a special operation like, `realign_with_sw` which might realign the HSP via a Smith-Waterman algorithm pulling extra bases from the flanking sequence. You might call your module Bio::Search::HSP::RealignHSP and put it in a file called `Bio/Search/HSP/RealignHSP.pm`. Note that you don't have to put this file directly in the `bioperl src` directory - you can create your own local directory structure that is in parallel to the `bioperl release src` code as long as you have updated your `PERL5LIB` to contain your local directory or use the `'use lib'` directive in your script. Also, you don't have to use the namespace Bio::Search::HSP as namespaces don't mean anything about object inheritance in perl, but we recommend you name things in a logical manner so that others might read your code and if you feel encouraged to donate your code to the project it might easily be integrated with existing modules.

So, you're going to write your new special module, you do need to make sure it inherits from the base Bio::Search::HSP::HSPI object. Additionally unless you want to reimplement all the initialization state in the current Bio::Search::HSP::GenericHSP you should just plan to extend that object. You need to follow the chained constructor system that we have setup so that the arguments are properly processed. Here is a

sample of what your code might look like (don't forget to write your own POD so that it will be documented, I've left it off here to keep things simple).

```
package Bio::Search::HSP::RealignHSP;
use strict;
use Bio::Search::HSP::GenericHSP;
use vars qw(@ISA); # for inheritance
@ISA = qw(Bio::Search::HSP::GenericHSP); # RealignHSP inherits from GenericHSP

sub new {
    my ($class,@args) = @_;
    my $self = $class->SUPER::new(@args); # chained constructor

    # process the 1 additional argument this object supports
    my ($ownarg1) = $self->_rearrange([OWNARG1],@args);

    return $self; # remember to pass the object reference back out
}

sub realign_hsp {
    my ($self) = @_;
    # implement my special realign method here
}
```

The above code gives you a skeleton of how to start to implement your object. To register it so that it is used when the SearchIO systems makes HSPs you just need to call a couple of functions. The code below outlines them.

```
use Bio::SearchIO;
use Bio::Search::HSP::HSPFactory;
use Bio::Search::Hit::HitFactory;

# setup the blast parser, you can do this with and SearchIO parser however
my $searchio = new Bio::SearchIO(-file => $blastfile,
                                -format => 'blast');

# build HSP factory with a certain type of HSPs to make
# the default is Bio::Search::HSP::GenericHSP
my $hspfact = new Bio::Search::HSP::HSPFactory(-type =>
        'Bio::Search::HSP::RealignHSP');

# if you wanted to replace the Hit factory you can do this as well
# additionally there is an analagous
# Bio::Search::Result::ResultFactory for setting custom Result objects
my $hitfact = new Bio::Search::Hit::HitFactory(-type =>
        'Bio::Search::Hit::SUPERDUPER_Hit');

$searchio->_eventHandler->register_factory('hsp', $hspfact);
$searchio->_eventHandler->register_factory('hit', $hitfact);
```

We have to register the HSPFactory which is the object which will create HSP objects, by allowing this to be built by a factory rather than a hardcoded `Bio::Search::HSP::GenericHSP->new(...)` call we are permitting a user from taking advantage of the whole parsing structure and the ability to slot their own object into the process rather than reimplementing very much. We think this is very powerful and is worth the system overhead which may not permit this to be as efficient in parsing as we would like. Future work will hopefully address speed and memory issues with this parser. Volunteers and improvement code is always welcome.

## **Notes**

1. <http://www.duke.edu>
2. <http://upg.duke.edu>
3. <http://cgt.genetics.duke.edu>
4. <http://doc.bioperl.org>
5. <http://cvs.open-bio.org/cgi-bin/viewcvs/viewcvs.cgi/bioperl-live/Bio/SearchIO/?cvsroot=bioperl>