# Building C/C++ libraries with Perl

## Author

Alberto Simões <ambs@cpan.org>.

## Biography: Alberto Simões

I am just another Perl hacker. Programming Perl for more than a dozen years, I use it mainly for web development and natural language processing.

My academic formation is in computer science, with a MSc and a PhD in natural language processing using Perl, of course.

My main activity is teaching at university level. Unfortunately I am not lucky to teach Perl. This last year I have taught Bison and Flex in a Languages Processing course, and used a pseudo-language for a course in Artificial Intelligence for Games.

I do research in natural language processing, with emphasis in the Portuguese language. This explains the amount of modules in the `Lingua::` and `Lingua::PT::` name-spaces. The need for speed makes me rely on C libraries which in turn causes my need for building C and C++ libraries using Perl.

Finally, in the Perl Community, I am member of the YAPC Europe Foundation board, the chair of The Perl Foundation Grants Committee, treasurer of the Portuguese Perl Programmers Association and a sporadic Dancer hacker.

## Abstract

We all know that the popular *AutoTools* is evil. *AutoConf* is messy, *AutoMake* a confusion, and the lack of an automated way to build this kind of projects on most Linux distributions, Mac OS X and Windows makes these applications portability poor.

If together with all this, your non-Perl code (let's talk mostly of C and C++) will be only used from within your own Perl module, it is a headache and a shame to make available these libraries in tarballs that somehow are not easy to install.

After years of user complains, I decided to bundle some libraries inside my Perl modules. I am sorry for anyone who wants to use the library from another language (I do not think there is such an user yet), but bundling the code in a Perl module that can be automatically installed by any `CPAN` tool is just great. And it is even greater if they work mostly out of the box on the three major operating systems (for Windows I need Strawberry Perl).

In this article I explain how I bundle some modules that include C or C++ libraries, like *Lingua::Identify::CLD*, *Lingua::NATools*, *Lingua::Jspell* or *Text::BibTeX*.

## Introduction and Motivation

*Before starting, I would like to make a statement: this article is my point of view on how things are in the open source world, and how I choose to solve some of those issues. My solution is not unique, and not the best, surely.*

Most C and C++ libraries that are available in open source projects are shipped in tarballs with a *configure* script and a *makefile*. That is an acceptable approach, but it is not the best. Other approaches are available, like *cmake*, but it didn't have much impact (at least, yet).

Most developers keep using the well known *AutoTools*, not because of their quality or even flexibility, but because everybody uses them, and the alternatives are not widespread. Also, there are lots of macros already available, and most `autoconf` scripts are just a bunch of copy and paste blocks from another projects, with minor changes, where maintainers just hope to work, and pray everyday.

What are these tools? They are mostly written in Perl (yes, that is true) and they use `M4` macros, a language nobody on earth understands and likes to use. These Perl scripts parse a definition of a configuration file, and generate a shell script, that will try to detect how to build your application. It will detect a C compiler, a bunch of libraries, the size of your integers, where the required libraries are,

and a lot of more useful and not so useful variables.

So, for a user to build one of these libraries, he needs to have a Perl interpreter, *autoconf*, *automake*, *libtool*, *m4*, *make*, and all the software needed to really build the library, typically a C compiler (gcc, probably) and all the libraries the software depends on.

This tool chain is too big to be easy to maintain, and to have users to install on non-Linux systems. Even Mac OS X that is supposed to be an Unix system has problems with these tools.

I have been bitten by *AutoTools* a lot of times, having to give support to users to install a library I maintain, just because they want to use a Perl module that uses that library.

I got enough from it, and decided to go all Perl. If *AutoTools* are written in Perl, users need Perl. Then, if I go for Perl as a requirement, I am not requesting that much. Then, I will probably need some modules that are not in the core. But those should be easy to install using any CPAN tool. With this in mind I decided to invest some time, and I got a tool chain of Perl modules to compile any C and C++ code that might be bundled with any of my Perl modules.

In the next section I will describe each of the modules I use in my tool chain for building my modules that include C or C++ code: *Lingua::Identify::CLD*, *Lingua::NATools*, *Lingua::Jspell*, *Text::BibTeX* or *Lingua::FreeLing3*. Then, I will describe briefly how some of these modules' build systems is working (no, I will not detail the code, you can see it on CPAN). At the end I will draw some conclusions.

Do not expect a step by step tutorial of how to write your module build system. Each module is different and has its own specifics. Also, some code blocks might have been simplified.

## Modules Tool Chain

In this section I describe the modules used, and for what they are used. I will not enter in detail on how to use them, or how to glue them with each other.

### Module::Build

Although the most used module in CPAN, unfortunately `ExtUtils::MakeMaker` has a big limitation: the rules are expressed in a *makefile*, where actions are shell commands. This makes it harder to detect what commands to run, and run them accordingly.

Regarding this, `Module::Build` has a big advantage. As the rules are expressed as Perl functions there is a lot more versatility. It is easy to write a module to subclass `Module::Build` and rewrite some of the methods according with our needs.

### ExtUtils::CBuilder

We are talking about building C software, which means we need a C compiler. For that, Perl bundles in its core modules the `ExtUtils::CBuilder` module, that is able to detect, using the `Config` module, which C compiler to use and with which flags. It also has information on how to link a library and how to link an executable. It even guesses some C++ flags quite well.

Its main drawback is that it is prepared to build Perl libraries: what I mean with Perl libraries is, the libraries that are built with XS code and are loaded with `DynaLoader`. These libraries have some differences from the standard libraries, at least on some operating systems. For instance, in Mac OS X, the Perl libraries are known as **bundles** and the standard C libraries are known as **dyld** libraries (standard dynamic libraries). They have different functionality (not that I am aware of the real differences, my knowledge in this aspect is just superficial).

### ExtUtils::ParseXS

Also in the Perl core modules is `ExtUtils::ParseXS`. It parses XS files and generates the corresponding C (or C++) code. If you use `ExtUtils::MakeMaker` or `Module::Build` built-in mechanism for building C extensions you are using this module by default. As I am building my extensions manually I need to call it myself, to create the C/C++ glue file.

#### ExtUtils::Mkbootstrap

Just like the previous module, `ExtUtils::Mkbooktrap` is also a Perl core module, and used by the usual build tools to compile XS code. It is used to create a file used by `DynaLoader` to load dynamically your extension. Again, as I am building everything by hand, it is part of my tool chain.

#### ExtUtils::PkgConfig / PkgConfig

At the moment I am using `ExtUtils::PkgConfig` to detect some libraries that bundle a `.pc` file. This mechanism was used originally with Gnome (if I recall correctly), and now is common on most libraries. To include a pkg-config file is easy, and can make the life of other programmers easier. So, why not.

The problem with `ExtUtils::PkgConfig` is that it uses the `pkg-config` binary file. This extra dependency is a problem. Some time ago it was discussed that it should exist a pure Perl implementation of this module. `PkgConfig` seems to be that module, but I am not using it yet. But I will probably migrate my build scripts to use `PkgConfig` in the future.

#### Config::AutoConf

With `Config::AutoConf` I start presenting two modules written by me (and with a lot of patches and contributions) to help me with the C and C++ libraries build process.

`Config::AutoConf` is a module to mimic some of the behavior you can get with `autoconf`. It has some methods to help detect if a library is available, if it can be linked with, if some header files are available, if some binary is available, etc.

To complete some of these tasks `Config::AutoConf` uses `ExtUtils::CBuilder` to build some simple C programs and detect if they build and run properly.

#### ExtUtils::LibBuilder

Finally, `ExtUtils::LibBuilder` is probably the most relevant module to build standalone system libraries, but also, the module that needs more work. It is a hairy module that uses a set of heuristics to, based on the information from `Config` and using `ExtUtils::CBuilder`, detect how to build a standalone system library. For that, it uses some magic, looks to the system type, tries to build a bunch of files, and if it succeeds, it returns the required flags for building the library.

This module is known to work in all Linux variants, Mac OS X and Windows with Strawberry Perl. I also think it should work with Cygwin, but I didn't check it myself. I would love to have it work with other compilers, like the Microsoft ones, but I do not have them for test, neither the time or interest to do it myself.

### My Build Modules

As I stated previously, I build my modules sub-classing `Module::Build`, and rewriting rules to compile the C code, build libraries, etc. This section describes briefly the structure of these modules implementation.

#### Lingua::Jspell

To start with, let us look into `Lingua::Jspell`. This module is a morphological analyzer, whose code, written in C, is derived from the well known `ispell` (therefore the name: `i++ = j`). It is mainly used for the Portuguese language, but it is language independent (well, at least for western European languages) accordingly with the dictionary used.

Regarding the technical details, `Lingua::Jspell` is composed by C code that is linked into a standard C library (`libjspell`), some C code that should be linked against `libjspell`, and a pure Perl module (the interface at the moment is performed using a bidirectional pipe, but in the future I plan to interface using XS).

With this description you can argue that the library should be shipped in an independent tarball. You are correct. But we end up noticing that nobody was using the C library by itself, and the work involved in maintaining *AutoTools* scripts was too much.

From the build chain described above, this module uses `Config::AutoConf`, `ExtUtils::CBuilder`, `ExtUtils::LibBuilder` and, of course, `Module::Build`.

The `Build.PL` script's algorithm is composed by:

1 The script starts by using `Config::AutoConf` to detect the `ncurses` library. In fact, I look for the header file,

```
Config::AutoConf->check_header("ncurses.h");
```

and then, for the `tgoto` function, for checking link capability:

```
Config::AutoConf->check_lib("ncurses", "tgoto");
```

2 Follows a big hack to find out where to install the C standard library. I do not want to install it in the usual place where Perl places the XS libraries, or the system will have trouble finding it, for instance, for the standalone binaries. For Unix systems I get the path where Perl would install binaries (usually `/usr/bin/` or `/usr/local/bin/`) and I replace the `bin` portion with `lib64` or `lib`, and check if they exist. I use the first one available. If none is available, I create the folder and cross my fingers.

For Windows I used to install in the `C:\Windows` path, but with Windows version 7 that folder is write protected. The solution (not the best, I know) was to split the `PATH` environment variable and try to write a dummy file in each folder. The first one that allows me that operation is the place where I will place the `dll` file.

3 All this information is stored both as `Module::Build` configure data (that will create a module named `Lingua::Jspell::ConfigData`) and in the builder notes. I also add build elements, so the `Module::Build` knows where to place the built files.

Example of a builder note being stored:

```
$builder->notes('libdir'   => $libdir);
```

Saving in the configure data:

```
$builder->config_data("libdir"  => $libdir);
```

And defining build elements:

```
$builder->add_build_element('usrlib');
$builder->install_path( 'usrlib' => $libdir );
```

Truthfully, not just the library folder is computed here, but also (for Unix systems) the `pkg-config` path (where the `.pc` file should be placed). But I will skip these details.

4 Finally, generate the build scripts, invoking the `Module::Build` method:

```
$builder->create_build_script;
```

Regarding the `Module::Build` subclass, a lot of more work is needed. `Module::Build` subclasses redefine build rules, where each rule is named `ACTION_actioname`. For instance, `ACTION_code` is called when you run `Build`, after running `Build.PL` for configuring the module.

Usually I sucblass this action with a method that just calls my build methods. Also, I prepare a `ExtUtils::LibBuilder` instance that will be used to build the library and compile the source code. This could be done every time I need it, but this way the initial tests performed by `ExtUtils::LibBuilder` to configure the build system are executed only once.

```
sub ACTION_code {
  my $self = shift;

  # create some folders I need to use
```

```
    for my $path (catdir("blib","pcfile"),
                  catdir("blib","incdir"),
                  catdir("blib","bindoc"),
                  catdir("blib","script"),
                  catdir("blib","bin")) {
        mkpath $path unless -d $path;
    }

    # create the LibBuilder object and save it
    my $libbuilder = ExtUtils::LibBuilder->new;
    $self->notes(libbuilder => $libbuilder);

    # dispatch every needed action
    $self->dispatch("create_manpages");
    $self->dispatch("create_yacc");
    $self->dispatch("create_objects");
    $self->dispatch("create_library");
    $self->dispatch("create_binaries");

    # and now, call superclass.
    $self->SUPER::ACTION_code;
}
```

These are the methods invoked, and how they behave:

`create_manpages`

As the name says, this method creates manpages from some `pod` files I have to document the C binaries that will be built. This is done searching for all `pod` files in a specific directory, and running `pod2man`.

At the moment I am running `pod2man` binary with exactly this name. This might be a problem for some installations that have a version concatenated in the binary name, or cases in which the binary is not available in the default binary path.

```
  sub ACTION_create_manpages {
    my $self = shift;

    # get a list of pod files
    my $pods = $self->rscan_dir("src", qr/\.pod$/);
    # get our module version
    my $version = $self->notes('version');

    # for each pod file
    for my $pod (@$pods) {
        # compute the man page name (and its path)
        my $man = $pod;
        $man =~ s!.pod!.1!;
        $man =~ s!src!catdir("blib","bindoc")!e;

        # skip if the file is up to date
        next if $self->up_to_date($pod, $man);

        # now, run directly the pod2man command
        `pod2man --section=1 --center="Lingua::Jspell"
                 --release="Lingua-Jspell-$version" $pod $man`;
    }
  }
```

Note that here I place the manpages in the `blib\bindoc` folder. Citing the documentation, *Documentation for the stuff in script and bin. Usually generated from the POD in those files. Under Unix, these are manual pages belonging to the 'man1' category.*

`create_yacc`

In this method I compute the C file from the `yacc` source file. The method is quite straightforward.

```
  sub ACTION_create_yacc {
    my $self = shift;

    my $ytabc  = catfile('src','y.tab.c');
    my $parsey = catfile('src','parse.y');

    return if $self->up_to_date($parsey, $ytabc);

    my $yacc = Config::AutoConf->check_prog("yacc","bison");
    if ($yacc) {
        `$yacc -o $ytabc $parsey`;
    }
  }
```

Although the generated file is shipped in the module tarball, if `yacc` or `bison` are available, I recompute it. This makes it easy for me to make changes and use the same `makefile` to build the module.

`create_objects`

I build the library in two steps. First I compute the object files in this method. To create the object files I do not need to use the `LibBuilder` module, therefore I access the `cbuilder` field in the Builder object. Then, search for all the C files, set the compile flags to use `ncurses` accordingly with the detection performed in `Build.PL`, and build each file, if needed.

```
  sub ACTION_create_objects {
    my $self = shift;

    my $cbuilder = $self->cbuilder;
    my $c_files  = $self->rscan_dir('src', qr/\.c$/);
    my $extra_compiler_flags = "-g " . $self->notes('ccurses');

    for my $file (@$c_files) {
        my $object = $file;
        $object =~ s/\.c/.o/;
        next if $self->up_to_date($file, $object);
        $cbuilder->compile(object_file  => $object,
                           source       => $file,
                           include_dirs => ["src"],
                           extra_compiler_flags =>
$extra_compiler_flags);
    }
  }
```

`create_library`

The final step to create the library is just to link the object files that were built in the last step in a standard dynamic library (`.so`, `.dyld` or `.dll` accordingly with the operating system). This is one of the places where I need to use the `LibBuilder` object.

The process is quite simple. Start by obtaining the `LibBuilder` object, detect which

extension the current operating system uses, and define the object files that are needed for the library. I could search for all the files with `.o` extension, but there are some files that I do not want to include in the library. Therefore, I decided to just list them all.

Then, define the library name and the place where it will be placed, define the linker flags, and run the `link` method in the `LibBuilder` object.

```
  sub ACTION_create_library {
    my $self = shift;

    # get details on the builder and lib extension
    my $libbuilder = $self->notes('libbuilder');
    my $LIBEXT = $libbuilder->{libext};

    # define what files will be linked together
    my @files = qw!correct defmt dump gclass good hash jjflags
                   jslib jspell lookup makedent sc-corr y.tab!;
    my @objects = map { catfile("src","$_.o") } @files;

    # define where the resulting library will be placed
    my $libpath = $self->notes('libdir');
    $libpath = catfile($libpath, "libjspell$LIBEXT");
    my $libfile = catfile("src","libjspell$LIBEXT");

    # define the linker flags
    my $extralinkerflags =
$self->notes('lcurses').$self->notes('ccurses');
    $extralinkerflags.=" -install_name $libpath" if $^O =~ /darwin/;

    # link if the library is not up to date
    if (!$self->up_to_date(\@objects, $libfile)) {
        $libbuilder->link(module_name => 'libjspell',
                          extra_linker_flags => $extralinkerflags,
                          objects => \@objects,
                          lib_file => $libfile,
                         );
    }

    # create a folder where to place the library
    my $libdir = catdir($self->blib, 'usrlib');
    mkpath( $libdir, 0, 0777 ) unless -d $libdir;

    # copy if needed
    $self->copy_if_modified( from   => $libfile,
                             to_dir => $libdir,
                             flatten => 1 );
  }
```

This code could be cleaned a little bit, but probably in a later release.

`create_binaries`

This method is very similar to the `create_objects` but, instead of creating object files from source files, it will compile binary files from object files. Again, this method will use the `LibBuilder` object for this task.

```
  sub ACTION_create_binaries {
    my $self = shift;

    # get details on the builder and binary extension
```

```
        my $libbuilder = $self->notes('libbuilder');
        my $EXEEXT = $libbuilder->{exeext};

        # define flags
        my $extralinkerflags =
    $self->notes('lcurses').$self->notes('ccurses');

        # define the binary that will be created
        my $exe_file = catfile("src", "jspell$EXEEXT");

        # what is the needed object file
        my $object = catfile("src", "jmain.o");

        # if needed, link the executable
        if (!$self->up_to_date($object, $exe_file)) {
            $libbuilder->link_executable(
                exe_file => $exe_file,
                objects  => [ $object ],
                extra_linker_flags => "-Lsrc -ljspell
    $extralinkerflags");
        }

        # and if needed, copy the file
        $self->copy_if_modified( from    => $exe_file,
                                 to_dir  => "blib/bin",
                                 flatten => 1);
        }
      }
```

As it can be seen, this division in small tasks makes it easy to follow. And they all have a similar base (very similar with *Makefile* rules): find the source files, apply a command (or more than one) to each of them, and obtain a set of target files.

The usual next step in the build process is to run `Build test`. This invokes the `ACTION_test` method. Usually I would not need to subclass this method, but as my tests need the binary to work, I need it to find the proper library at run time. More important, I need it to find the library it just linked, and not another version that may be hanging in the file system. For that, I just tweak the library search path, taking care to do it correctly accordingly with the operating system in which we are running.

```
sub ACTION_test {
  my $self = shift;

  if ($^O =~ /mswin32/i) {
      $ENV{PATH} = catdir($self->blib,"usrlib").";$ENV{PATH}";
  }
  elsif ($^O =~ /darwin/i) {
      $ENV{DYLD_LIBRARY_PATH} = catdir($self->blib,"usrlib");
  }
  elsif ($^O =~ /(?:linux|bsd|sun|sol|dragonfly|hpux|irix)/i) {
      $ENV{LD_LIBRARY_PATH} = catdir($self->blib,"usrlib");
  }
  elsif ($^O =~ /aix/i) {
      my $oldlibpath = $ENV{LIBPATH} || '/lib:/usr/lib';
      $ENV{LIBPATH} = catdir($self->blib,"usrlib").":$oldlibpath";
  }

  $self->SUPER::ACTION_test
```

```
}
```

Finally, if everything worked correctly, the next usual command would be `Build install`. Or, probably `Build fakeinstall` if you want to test how things would be installed before really installing them. In my case, both `ACTION_install` and `ACTION_fakeinstall` start by calling a custom action named `pre_install`.

The `pre_install` action does some paths cleanups, and copies some files that don't need to be built to the proper place in the `blib` staging folder. I will not share here the code, as it doesn't have much to talk about, and I prefer not to waste space with it. The more interesting portion might be a call to `fix_shebang_line` and `make_executable` to some scripts that I edit manually, and therefore `Module::Build` doesn't place in the correct place. I also check if the path where the library will be installed is a standard one, and if not, warn the user to add the path to ldconfig, or things will not work properly.

The install action has just one extra step, that is running `ldconfig` if it is available, the operating system is Linux and user is root. In fact, I probably should look to the *uid*. In a next release.

**Lingua::Identify::CLD**

This module is a Perl interface to the Google's Chromium Compact Language Detector (CLD) library. CLD is not available as a tarball at the moment I am writing this. It is available in a Google code repository, only. After talking with its maintainer, I decided to bundle the entire library code in my module.

There aren't many differences from the previous module. The main differences are the use of C++ code, and the fact that in this case I have XS code, and therefore, I need to use `ExtUtils::ParseXS` and `ExtUtils::Mkbootstrap`.

To compile the C++ code, and link the library, the only differences are adding some libraries like `libstdc++`, and set the `C++` option to true when invoking the `compile` or `link` methods:

```
$cbuilder->compile(object_file  => $object,
                   source       => $file,
                   include_dirs => ["cld-src"],
                   extra_compiler_flags => $extra_compiler_flags,
                   'C++' => 1);
```

and

```
$libbuilder->link(module_name => 'libcld',
                  extra_linker_flags => $extralinkerflags,
                  objects => $o_files,
                  lib_file => $libfile,
                  'C++' => 1);
```

Other than that, I need to take care of building the XS portion. For inspiration I gave a look to other modules that do this by hand, like the case of `Glib`. This will be, probably, the longest method in this article.

The code is commented, but the main steps are:

1. Define the XS source file, the C file that will be generated, and the object file that will be created.
2. Process the XS file with `ExtUtils::ParseXS`, saying I am processing a C++ file.
3. Given the C++ source file, create the object file from it. For this I use the standard `CBuilder` builder object.
4. The next step is the creation of a bootstrap file. It is created by `ExtUtils::Mkbootstrap`

module. I am not aware of the details of this file, nor why it's not always created. I confess I just copied this bunch of code from another module, and it seems to work. Open source is great.

5 The next step is building the library that will be loaded by `DynaLoader`. First I define the linker flags, and then use the standard `CBuilder` builder object to create the library. This time I am not using `LibBuilder` because I am not building a standalone one, but one that will be used only by the Perl module.

```
sub ACTION_compile_xscode {
  my $self = shift;
  my $cbuilder = $self->cbuilder;

  my $archdir = catdir( $self->blib, qw'arch auto Lingua Identify CLD');
  mkpath( $archdir, 0, 0777 ) unless -d $archdir;

  # set file names
  my $cfile = catfile("CLD.cc");
  my $xsfile= catfile("CLD.xs");
  my $ofile = catfile("CLD.o");

  # create CLD.cc from CLD.xs
  if (!$self->up_to_date($xsfile, $cfile)) {
      ExtUtils::ParseXS::process_file( filename   => $xsfile,
                                       'C++'      => 1,
                                       prototypes => 0,
                                       output     => $cfile);
  }

  # create CLD.o from CLD.cc
  my $extra_compiler_flags = $self->notes('CFLAGS');
  if (!$self->up_to_date($cfile, $ofile)) {
      $cbuilder->compile( source              => $cfile,
                          include_dirs        => [ catdir("cld-src") ],
                          'C++'               => 1,
                          extra_compiler_flags => $extra_compiler_flags,
                          object_file         => $ofile);
  }

  # Create .bs bootstrap file, needed by Dynaloader.
  my $bs_file = catfile( $archdir, "CLD.bs" );
  if ( !$self->up_to_date( $ofile, $bs_file ) ) {
      ExtUtils::Mkbootstrap::Mkbootstrap($bs_file);
      if ( !-f $bs_file ) {
          # Create file in case Mkbootstrap didn't do anything.
          open( my $fh, '>', $bs_file ) or confess "Can't open $bs_file:
$!";
      }
      utime( (time) x 2, $bs_file );    # touch
  }

  # set linker flags
  my $extra_linker_flags = "-Lcld-src -lcld -lstdc++";
  $extra_linker_flags .= " -lgcc_s" if $^O eq 'netbsd';
```

```
    my $objects = [ $ofile ];

    my $lib_file = catfile( $archdir, "CLD.$Config{dlext}" );
    if ( !$self->up_to_date( [ @$objects ], $lib_file ) ) {
        my $btparselibdir = $self->install_path('usrlib');
        $cbuilder->link(
                        module_name => 'Lingua::Identify::CLD',
                        extra_linker_flags => $extra_linker_flags,
                        objects     => $objects,
                        lib_file    => $lib_file,
                       );
    }
}
```

Of course I could split these steps in different methods, dispatched from here. That can be done in a future release, who knows.

**Text::BibTeX**

`Text::BibTeX` is a module to parse BibTeX files. It uses a C library, named `btparse` for that task. This library was available (and an old version still is) as a separated tarball, but because of the same reasons already discussed, I bundled the C code in the Perl module.

It uses the same idea of the previous modules, with no big differences.

**Lingua::FreeLing3**

This module is an API to a C++ library, named `FreeLing`. It is used to perform natural language processing tasks, like parsing, dependency parsing, name entity recognition, etc.

In this case I do not bundle any C code from the library, only a XS file generated from SWIG. Some of the code described above was used in this module as well.

The main reason not to use the `Module::Build` capabilities to build XS code directly is that I needed to detect FreeLing by myself, and detect which libraries it needs to be linked against. Therefore, I decided to take the build system in my hands.

**Lingua::NATools**

Finally, `Lingua::NATools` is a toolkit for processing and aligning parallel corpora. Is has been used by a lot of researchers to extract probabilistic translation dictionaries. At first, I bundled the Perl module inside the AutoTools tarball. It was a mess, and few users were able to install and compile it properly.

With the expertise I gained with the other modules (mainly `Lingua::Jspell` and `Text::BiBTeX`) I decided to do the other way around, and included the C part in a Perl module.

The main difference from this tool to the others is the high amount of C and Perl dependencies (the method to detect is the same as above, for `Lingua::Jspell`), and the fact that the tests include some binaries for themselves (that are not installed, just compiled and executed during the testing stage).

For that, the `ACTION_test` method starts by dispatching to a `create_test_binaries` action, very similar to the `create_binaries` action, that builds the binaries.

## Final Conclusions and Remarks

All this process is very tedious to maintain, but once I get it working, I do not need to change it much. I am not completely happy with the code quality or maintainability. Also, not happy with `ExtUtils::LibBuilder` implementation. But for now, this seems to work on my major target architectures.

When I have time, and a lower stack of to-do items, I might patch `Module::Build`, or create some

`Module::Build::Library`, that automates some of these tasks, as they seem easy to automate. The result of not having that module yet, is that some of the modules I have described have better build implementations that others.

Finally, I would be happy with suggestions, patches, and comments. I think this kind of tool chain is important to the Perl community, and can make the difference for some specific projects.