Report on Starfish v. 1.31 A Perl-based System for Text-Embedded Programming and Preprocessing

(Starfish Version 1.31, Document Revision 392)

Vlado Kešelj

April 25, 2020

Abstract

This report is meant to be the most up-to-date documentation on Starfish. However, it has not been completed yet. A large part of it is a direct POD Documentation generated from in-code documentation.

Starfish is an open-source, Perl-based system for macro processing and text-embedded programming. It demonstrates relatively simple methodology based on regular expression matching and rewriting, which is implemented in Perl in a transparent way.

The main idea is similar to the text-embedding style of PHP and other systems, many of which where also implemented as Perl modules, but there are several essential novel features of Starfish.

Contents

Chapter 1

Introduction

1.1 Text-Embedded Programming

The concept of text-embedded programming is well-known and popular: snippets of source code are embedded in a document, and during processing, the snippets are replaced with the result of their evaluation. The technique is mainly used for generating HTML documents. Examples of programming languages or frameworks that use it are PHP, ASP, and JSP.[PHP] The current Web and IT technology in general includes consistently expansive use of text-based technologies; such as the use of HTML, XML, JSON, and semi-structured data.

The code snippets are marked in text with starting and ending delimiters, which function as escape sequences that toggle on and off the code processing. Typical string delimiters are "`<?`" and "`?>`" or "`<?php`" and "`?>`" in PHP, "`<\%`" and "`\%>`" in ASP, and "`<?`" and "`!>`" in ePerl. During processing, the text outside the code snippets is left intact, while the code snippets are evaluated and the evaluation results are used to replace the snippets. For example, in PHP, we could prepare an HTML document such as:

```
<html><head><title>PHP Test</title></head>
<body>
<?php echo '<p>Hello World</p>'; ?> </body></html>
```

and after processing with the PHP interpreter, the following output would be produced:

```
<html><head><title>PHP Test</title></head>
<body>
<p>Hello World</p>
</body></html>
```

Embedding the code inthis way is sometimes called *escaping* because a starting delimiter, such as "`<?`" serves as an escape sequence, triggering special processing of the snippet. Another kind of escaping, referred to as the *advanced escaping* in PHP is illustrated with the following example:

```
Good <?php if ($hour < 12) { ?> Morning <?php } else { ?> Evening
<?php } ?>
```

We will refer to this kind of escaping as *inverted escaping.* Inverted escaping can be interpreted in the following way: The complete input text is treated as code in which the plain text, i.e., the non-code text, is embedded between '`?>`' and '`<?php`' delimiters and it is translated into an '`echo "`*string*`";`' statement; and similarily, any part of the form '`?> plain text <?`' is interpreted as the statement:

```
echo " plain text ";
```

An implicit delimiter '`?>`' is assumed at the beginning of the text and an implicit delimiter '`<?php`' is assumed at the end of text. Although it is relatively easy to implement, we do not use inverted escaping in Starfish since its benefits are not very clear. On the other hand, inverted escaping does not follow the priciple that each snippet should be a well-defined block of code Additionally, Perl offers a plethora of string delimiting options, such as `q/.../` and `<<'EOT'`, which can be used in place of inverted escaping to include larger parts of plain text.

1.2 Overview

After this introductory chapter, Chapter 2 discusses related work, Chapter 3 presents user documentation, Chapter 4 describes the system design, and Chapter 5 gives more starfish details for reference. Chapter A contains the POD documentation. This documenation is written as a part of code and is included in the man page as well. As such, it is meant to be a reference that may be frequently needed during coding and actual use of Starfish.

Chapter 2

Related Work

We will describe here some previous work on Perl-based embedded programming. Our vision for text-embedded framework is not that it is a major characteristic of a programming language, but it should be an orthogonal framework that allows several programming languages as options.

The Perl programming language is particularly suitable for implementation of text-embedded capability due to its string-processing functionalities, and its ability for run-time code code interpretation and execution (the 'eval' function). In 1998, when the work on Starfish started, there was a system that that partially implemented needed functionality — it was called ePerl (Embedded Perl Language) by Ralf S. Engelshall. [ePerl]

The langauge ePerl was developed in the period from 1996 to 1998. It is an embedded Perl langauge in the sense that we described, but ultimately there were two reasons why it did not fit needed requirements: (1) it seemed to be too heavy-weight, and (2) it did not support the *update* mode, that will be described in the next chapters. Let us explain the "too heavy-weight" comment: The language ePerl is a package of 195KB, created by modifying the Perl source code and requiring compilation during installation. We prefer a more convenient solution, which requires that the standard Perl is installed, and our solution is installed as a Perl module. Some other authors noted the heavy-weight nature of ePerl as well.

For example, David Ljung Madison developed an "ePerl hack" [ePerl-h] which is a Perl script of some 1400 lines that has functionality similar to ePerl.

Text::Template by Mark Jason Dominus [Text::Template] is another Perl module with similar functionality. An interesting and probably indepenent simiarity is that Starfish uses `$O` vas the output variable, while `$OUT` is used in Text::Template. The default embedded code delimiters in Text::Template are '`{`' and '`}`', with an additional condition that braces have to be properly nested. For example, '`{{{"abc"}}}`' is a valid snippet with delimiters. The module allows the user to change the default delimiters to other alternative delimiters.

The well-known Perl module HTML::Mason [HTML::Mason] by Jonathan Swartz, Dave Rolsky, and Ken Williams can also be seen as an embedded Perl system, but it is a larger system with the design objective being a a high-performance, dynamic web-site authoring system.

Starfish is a lighter-weight system than eperl or Mason, but it is, in a certain sense, more flexible than Text::Template and the ePerl hack, so I belive it deserves survial among the pack. Since I followed some ePerl design parameters in the beginning, I called the system SLePerl initially, as an abbreviation for "Something Like ePerl", and changed it later to Starfish. The name Starfish came from a vague metaphor of starfish feeding on shellfish and shellfish containing pearls. The metaphor is somewhat inverted, since the Starfish program is looking for Perl snippets, digests them, and replaces or updates them with the output.

!!!TO CONTINUE

Chapter 3

Starfish Use

This chapter describes some implementational topic regarding Starfish.
⋮

3.1 Hooks

Starfish uses the concept of a “hook” (or triggers) and evaluators to initiate processing on a text. The name “hook” is inspired by a similar term used in Emacs. For example, the delimiters ‘`<?`’ and ‘`!>`’ represent a hook, which is associated with an evaluator that will evaluate the code between the delimiters and produce the result that will replace the hook. In the update mode, the code will be replaced with something like:

```
<? code !>
#+
...output
#-
```

while in the replace mode, the code is replaced with “`...output`,”.
⋮

The regex (regular expression) hooks pass captured substrings as arguments to the replacement function. If the whole captured string (`$&`) is needed, it can be obtained from the `$self->{currenttoken}` field.
⋮

3.2 Iterative Processing

In the default mode of processing, Starfish reads input file, processes it, and writes it back to the output file if the output is different. The processing of the text could be repeated. The number of iterations is set by default to 1, but it could be larger. If the number of iterations is very large, the number of actual iterations could be smaller if a fixed point is reached earlier. A typical code of setting the number of iterations to 2 is the following:

```
$Star->{Loops} = 2;
```

We can read the number of the current loop with `$Star->{CurrentLoop}`.

In case of replace mode, the iterations are repeated on the original input file and only in the last iteration the replaced output is produced.

Chapter 4

System Design

The typical use of the system is by calling the `starfish` command, which runs the `starfish_cmd` function with arguments being passed from the command line.

The `starfish_cmd` function creates a Starfish object `$sf`, translates the options from the command line into the appropriate settings of the object, and runs the method `$sf->process_files(@tmp)`, where `@tmp` contains the names of the files given in the command line. The function `starfish_cmd` returns the object `$sf` at the end.

The method `$sf->process_files` expects a list of files. Each file is processed and writen out. The main part of processing start with seting `$sf->{data}` to the file contents, calling the method `$sf->digest()`, and writing the processed output, which is found in `$sf->{Out}`.

The method `$sf->digest()` mainly consists of a loop which scans the `$sf->{data}` and prepares output in `$self->{Out}`. A custom function defined as reference `$sf->{Final}` can be used for final processing of the output.

4.1 Function digest

4.2 Function scan

Scans text and finds the next token.

Chapter 5

Starfish Reference

5.1 Styles

There is a set of predefined styles for different input files: HTML (html), HTML templating style (.html.sfish), TeX (tex), Java (java), Makefile (makefile), PostScript (ps), Python (python), and Perl (perl).

Appendix A

POD Documentation

A.1 NAME

Text::Starfish.pm and starfish - A Perl-based System for Text-Embedded Programming and Preprocessing

A.2 SYNOPSIS

starfish [**-o=***outputfile*] [**-e=***initialcode*] [**-replace**] [**-mode=***mode*] *file...*

where files usually contain some Perl code, delimited by `<?` and `!>`. To produce output to be inserted into the file, use variable `$O` or function `echo`.

A.3 DESCRIPTION

(The documentation is probably not up to date.)

Starfish is a system for Perl-based text-embedded programming and preprocessing, which relies on a unifying regular expression rewriting methodology. If you know Perl and php, you probably know the basic idea: embed Perl code inside the text, execute it is some way, and interleave the output with the text. Very similar projects exist and some of them are listed in §A.13. Starfish is, however, unique in several ways. One important difference between `starfish` and similar programs (e.g. php) is that the output does not necessarily replace the code, but it follows the code by default. It is attempted with Starfish to provide a universal text-embedded programming language, which can be used with different types of textual files.

There are two files in this package: a module (Starfish.pm) and a small script (starfish) that provides a command-line interface to the module. The options for the script are described in subsection "starfish_cmd list of file names and options".

The earlier name of this module was SLePerl (Something Like ePerl), but it was changed it to `starfish` -- sounds better and easier to type. One option was 'oyster,' but some people are thinking about using it for Perl beans, and there is a (yet another) Perl module for embedded Perl `Text::Oyster`, so it was not used.

The idea with the '`starfish`' name is: the Perl code is embedded into a text, so the text is equivalent to a shellfish containing pearls. A starfish comes by and eats the shellfish... Unlike a natural starfish, this `starfish` is interested in pearls and does not normally touch most of the surrounding meat.

A.4 EXAMPLES

A simple example

A simple example, after running `starfish` on a file containing:

```
<? $O= "Hello world!" !>
```

we get the following output:

```
<? $O= "Hello world!" !>
#+
Hello world!
#-
```

The output will not change after running the script several times. The same effect is achieved with:

```
<? echo "Hello world! !>
```

The function echo simply appends its parameters to the special variable $O.

Some parameters can be changed, and they vary according to style, which depends on file extension. Since the code is not stable, they are not documented, but here is a list of some of them (possibly incorrect):

```
 - code prefix and suffix (e.g., <? !> )
 - output prefix and suffix (e.g., \n#+\n \n#-\n )
 - code preparation (e.g., s/\\n(?:#+|%+\/\/+)/\\n/g )
```

HTML Examples

Example 1

If we have an HTML file, e.g., `7.html` with the following content:

```
<HEAD>
<BODY>
<!--<? $O="This code should be replaced by this." !>-->
</BODY>
```

then after running the command

```
starfish -replace -o=7out.html 7.html
```

the file `7out.html` will contain:

```
<HEAD>
<BODY>
This code should be replaced by this.
</BODY>
```

The same effect would be obtained with the following line:

```
<!--<? echo "This code should be replaced by this." !>-->
```

Output file permissions

The permissions of the output file will not be changed. But if it does not exist, then:

```
starfish -replace -o=7out.html -mode=0644 7.html
```

makes sure it has all-readable permission.

Example 2

Input file `21.html`:

```
<!--<? use CGI qw/:standard/;
       echo comment('AUTOMATICALLY GENERATED - DO NOT EDIT');
!>-->
<HTML><HEAD>
<TITLE>Some title</TITLE>
</HEAD>
<BODY>
<!--<? echo "Put this." !>-->
</BODY>
</HTML>
```

Output:

```
<!-- AUTOMATICALLY GENERATED - DO NOT EDIT -->
<HTML><HEAD>
<TITLE>Some title</TITLE>
</HEAD>
<BODY>
Put this.
</BODY>
</HTML>
```

Example from a Makefile

```
LIST=first second third\
 fourth fifth

<? echo join "\n", getmakefilelist $Star->{INFILE}, 'LIST' !>
#+
first
second
third
fourth
fifth
#-
```

Beside $O, $Star is another predefined variable: It refers to the Starfish object currently processing the text.

TeX and LaTeX Examples

Simple TeX or LaTeX Example

Generating text with a variable replacement:

```
<?echo "
When we split the probability reserved for unseen characters equally
among the remaining $UnseenNum characters, we obtain the final estimated
probabilities:
"!>
```

Example from a TeX file

```
% <? $Star->Style('TeX') !>

% For version 1 of a document
% <? #$Star->addHook("\n%Begin1","\n%End1",'s/\n%+/\n/g');
%    #$Star->addHook("\n%Begin2","\n%End2",'s/\n%*/\n%/g');
%    #For version 2
%    $Star->addHook("\n%Begin1","\n%End1",'s/\n%*/\n%/g');
%    $Star->addHook("\n%Begin2","\n%End2",'s/\n%+/\n/g');
% !>

%Begin1
%Document 1
%End1

%Begin2
Document 2
%End2
```

LaTeX Example with Final Routine used for Slides

```
% -*- compile-command: "make 01s 01"; -*-
%<? ##read_starfish_conf();
%  $TexTarget = 'slides';
%  sfish_add_tag('sl,l', 'echo');
%  sfish_add_tag('slide', 'echo');
%  sfish_ignore_outer;
%  $Star->add_final( sub {
%    my $r = shift;
%    $r =~ s/^% -\*- compile-command.*\n//;
%    $r.= "\\end{document}\n";
%    return $r;
%  } );
% !>

\section{Course Introduction}

Not in slide.

%slide:In slide.

%<sl,l>
In slides and lectures.
%</sl,l>
```

Example with Test/Release versions (Java)

Suppose you have a stanalone java file p.java, and you want to have two versions:

```
p_t.java -- for complete code with all kinds of testing code, and
p.java -- clean release version.
```

Solution:
Copy p.java to p_t.java and modify p_t.java to be like:

```
/** Some Java file.  */

//<? $O = defined($Release) ?
// "public class p {\n" :
// "public class p_t {\n";
//!>//+
public class p_t {
//-

  public static int main(String[] args) {

    //<? $O = "    ".(defined $Release ?
    //qq[System.out.println("Test version");] :
    //qq[System.out.println("Release version");]);
    //!>//+
    System.out.println("Release version");//-

    return 0;
  }
}
```

In Makefile, add lines for updating p_t.java, and generating p.java (readonly, so that you do not modify it accidentally):

```
p.java: p_t.java
      starfish -o=$@ -e='$$Release=1' -mode=0400 $<
tmp.ind: p_t.java
      starfish $<
      touch tmp.ind
```

Command-line Examples

The following are the reference examples. For further information, please lookup the explanations of the command-line options and arguments.

starfish -mode=0400 -replace -o=*paper.tex* -mode=0400 *paper.tex.sfish*

In the above line, Starfish is used on top of a TeX/LaTeX file. The Starfish is separated from the .tex file to keep the source clean. However, a user in this situation may by mistake start editing the paper.tex file, so we set the output file mode to 0400 to prevent this accidental editing.

Macros

Note: This is a quite old part of Starfish and needs a revision. Macros are a form of code folding (related terms: holophrasting, ellusion(?)), expressed in the Starfish framework.

Starfish includes a set of macro features (primitive, but in progress). There are two modes, hidden macros and not hidden, which are indicated using variable $Star->{HideMacros}, e.g.:

```
starfish -e='$Star->{HideMacros}=1' *.sfish
starfish *.sfish
```

Macros are activated with:

```
<? $Star->defineMacros() !>
```

In Java mode, a macro can be defined in this way:

```
//m!define macro name
...
//m!end
```

After //m!end, a newline is mandatory. After running Starfish, the definition will disapear in this place and it will be appended as an auxdefine at the end of file.

In the following way, it can be defined and expanded in the same place:

```
//m!defe macro name
...
//m!end
```

A macro is expanded by:

```
//m!expand macro name
```

When macro is expanded it looks like this:

```
//m!expanded macro name
...
//m!end
```

Macro is expanded even in hidden mode by:

```
//m!fexpand macro name
```

and then it is expanded into:

```
//m!fexpanded macro name
...
//m!end
```

Hidden macros are put at the end of file in this way:

```
//auxdefine macro name
...
//endauxdefine
```

Old macro definition can be overriden by:

```
//m!newdefe macro name
...
//m!end
```

A.5 PREDEFINED VARIABLES AND FIELDS

$O

After executing a snippet, the contents of this variable represent the snippet output.

$Star

More precisely, it is $::Star. $Star is the Starfish object executing the current code snipet (this). There can be a more such objects active at a time, due to executing Starfish from a starfish snippet. The name is introduced into the main namespace, which might be a questionable decision.

$Star->{Final}

If defined, it should be an array of CODE references, which are applied as functions on the final output before writing it out. These are used as final routines, typically to add or remove some of the first lines or finals lines. Each functionj takes input as a parameter and returns it after processing. The variable should accessed using the method `add_final`.

$Star->{INFILE}

Name of the current input file.

$Star->{Loops}

Controls the number of iterations. The default value is 1, but we may want to repeat starfishing the text several times, or even until a fix-point is reached. For example, by setting the number of Loops to be at least 2, as in:

```
$Star->{Loops} = 2 if $Star->{Loops}<2;
```

we require Starfish to proces the input in at least two iterations.

$Star->{Out}

Output content of the current processing unit. For example, to use #-style line comments in the replace Starfish mode, one can make a final substitution in an HTML file:

```
<!--<? $Star->{Out} =~ s/^#.*\n//mg; !>-->
```

It is important to have in mind that the contents of this variable is the output processed so far, so any final output processing should be done in a snippet where no new output is produced.

$Star->{OUTFILE}

If option `-o=*` is used, then this variable contains the name of the specified output file.

A.6 METHODS

Text::Starfish->new(options)

The method for creation of a new Starfish object. If we are already processing within a Starfish object, we may use a shorter variant $Star->new().

The options, given as arguments, are a list of strings, which may include the following:

`-infile=*` Specifies the name of the input file (field INFILE). The file will not be read.

`-copyhooks` Copies hooks from the Star object (`$::Star`). This option is also available in `loadinclude`, `getinclude`, and `include`, from which it is passed to `new`. It causes the new object to have similar properties as the current Star object. It could be generalized to include any specified object, or to use the prototype object that is given to the constructor, but there does not seem to be need for this generalization. More precisely, `-copyhooks` copies the fields: `Style`, `CodePreparation`, `LineComment`, `IgnoreOuter`, and per-component copies the array `hook`.

$o->add_tag($tag, $action)

Normally used by `sfish_add_tag` by translating the call to `$Star->add_tag($tag, $action)`. Examples:

```
$Star->add_tag('slide', 'ignore');
$Star->add_tag('slide', 'echo');
```

See `sfish_add_tag` for a few more details.

$o->add_hook($ht,...)

Adds a new hook. The first argument is the hook type, which is a string. The following is the list of hook types with descriptions:

regex, *regex*, *replace*

The hook type `regex` is followed by a regular expression and a replace argument. Whenever a regular expression is matched in text, it is "starfished" according to the argument replace. If the argument replace is the string "`comment`", it is treated as the comment. If the argument replace is code, it is used as the evaluation code. For example, the following source in an HTML file:

```
<!--<? $Star->add_hook('regex', qr/^.section:(\w+)\s+(.*)/,
sub { $_="<a name\"$_[2]\"><h3>$_[3]</h3</a>" }) !>-->
```

```
line before
.section:overview Document Overview
line after
```

will produce the following output, in the replace mode:

```
line before
<a name"overview"><h3>Document Overview</h3></a>
line after
```

$o->addHook

This method is deprecated. It will be gradually replaced with add_hook, which is better defined since it includes hook type.

Adds a new hook. The method can take two or three parameters:

```
($prefix, $suffix, $evaluator)
```

or

```
($regex, $replacement)
```

In the case of three parameters `($prefix, $suffix, $evaluator)`, the parameter $prefix is the starting delimiter, $suffix is the ending delimiter, and $evaluator is the evaluator. The parameters $prefix and $suffix can either be strings, which are matched exactly, or regular expressions. An empty ending delimiter will match the end of input. The evaluator can be provided in the following ways:

special string 'default'

in which case the default Starfish evaluator is used,

special strings 'ignore' and 'echo'

'ignore' ignores the hook and produces no echo, 'echo' simply echos the contests between the delimiters.

other strings

are interpreted as code which is embedded in an evaluator by providing a local $_, $self which is the current Starfish object, $p - the prefix, and $s the suffix. After executing the code $p.$_.$s is returned, unless in the replacement mode, in which $_ is returned.

code reference (sub {...})

> is interpreted as code which is embedded in an evaluator. The local $_ provides the captured string. Three arguments are also provided to the code: $p - the prefix, $_, and $s - the suffix. The result is the value of $_.

For the format with two parameters, `($regex, $replacement)`, currently in this mode addHook understands replacement 'comment' and code reference (e.g., sub { ... }). The replacement 'comment' will repeat the token in the non-replace mode, and remove it in the replace mode; e.i., equivalent to no echo. The regular expression is matched in the multi-line mode, so ˆ and $ can be used to match beginning and ending of a line. (Caveat: Due to the way how scanner works, beginning of a line starts after the end of previously matched token.)

Example:

```
$Star->addHook(qr/^#.*\n/, 'comment');
```

$o->ignore_outer()

Sets the mode for ignoring the outer text in the replace mode. The function `sfish_ignore_outer` does the same on the default object `Star`. If an argument is given, it is used to set the mode, so as a consequence the mode can be turned off by giving the argument ''.

$o->last_update()

Or just last_update(), returns the date of the last update of the output.

$o->process_files(@args)

Similar to the function starfish_cmd, but it expects already built Starfish object with properly set options. Actually, starfish_cmd calls this method after creating the object and returns the object.

$o->rmHook($p,$s)

Removes a hook specified by the starting delimiter $p, and the ending delimiter $s.

$o->rmAllHooks()

Removes all hooks. If no hooks are added, then after exiting the current snippet it will not be possible to detect another snippet later. A typical usage could be as follows:

```
    $Star->rmAllHooks();
    $Star->addHook('<?starfish ','?>', 'default');
```

\$o->setStyle(\$s)

Sets a particular style of the source file. Currently implemented options are: html, java, latex, makefile, perl, ps, python, TeX, and tex. If the parameter $s is not given, the stile given in $o->{STYLE} will be used if defined, otherwise it will be guessed from the file name in $o->{INFILE}. If it cannot be correctly guessed, it will be the Perl style.

A.7 PREDEFINED FUNCTIONS

include(*filename and options*) -- starfish a file and echo

Reads, starfishes the file specified by file name, and echos the contents. Similar to PHP include. Uses getinclude function.

getinclude(*filename and options*) -- starfish a file and return

Reads, starfishes the file specified by file name, and returns the contents (see also include to echo the content implicitly). By default, the program will not break if the file does not exist. The option -noreplace will starfish file in a non-replace mode. The default mode is replace and that is usually the mode that is needed in includes (non-replace may lead to a suprising behaviour). The option -require will cause program to croak if the file does not exist. It is similar to the PHP function require. A special function named require is not used since `require` is a Perl reserved word. Another interesting option is `-copyhooks`, for using hooks and some other relevant properties from the Star object (`$::Star`). This option is eventually passed to `new`, so you can see the constructor new for more details.

The code for get include is the following:

```
sub getinclude($@) {
    my $sf = loadinclude(@_);
    $sf->digest();
    return $sf->{Out};
}
```

and it can be used as a useful template for using `loadinclude` directly. The function `loadinclude` creates a Starfish object, and reads the file, however it is not digested yet, so one can modify the object before this.

loadinclude(*filename and options*) -- load file and get ready to digest

The first argument is a filename. Loadinclude will interpret the options `-replace`, `-noreplace`, and `-require`. A Starfish object is created by passing the file name as an `-infile` argument, and by passing other options as arguments. The file is read and the object is returned. By default, the program will not break if the file does not exist or is not readable, but it will return undef value instead of an object. See also documentation about `include`, `getinclude`, and `new`.

`-noreplace` option will set up the Starfish object in the no-replace mode. The default mode is replace and that is usually the mode that is needed in includes. The option `-require` will cause program to croak if the file does not exist. An interesting option is `-copyhooks`, which is documented in the `new` method.

read_starfish_conf

This function is usually called at the begining of a starfish file, in order to read local configuration. it tests whethere there exists a filed named `starfish.conf` in the current directory. If it does exist, it checks for the same file in the parent directory, then gran-parent directory, etc. Once the process stops, is starts executing the configuration files in the order from first ancestor down. For each file, it changes directory to the corresponding directory, and requires (in Perl style) the file in the package main.

sfish_add_tag (*tag*, *action*)

Used to introduce simple tags such as line tag `%sl,l:` and %<sl,l>...</sl,l> in TeX/LaTeX for inclusion and exclusion of text. Example:

```
sfish_add_tag('sl,l', 'echo');
sfish_add_tag('slide', 'ignore');
```

and, for example, the following text is included:

```
%sl,l:some text to the end of line
%<sl,l>
more lines of text
%</sl,l>
```

and the following text is excluded:

```
%slide:this line is excluded
%<slide>
more lines of text excluded
%</slide>
```

sfish_ignore_outer()

Sets the default object `$Star` in the mode for ignoring outer text if in the replace mode. If an argument is given, it is used to set the mode, so as a consequence the mode can be turned off with `sfish_ignore_outer('')`.

starfish_cmd *list of file names and options*

The function `starfish_cmd` is called by the script `starfish` with the `@ARGV` list as the list of arguments. The function can also be used from Perl code to "starfish" a file, e.g.,

```
    starfish_cmd('somefile.txt', '-o=outfile', '-replace');
```

The arguments of the functions are provided in a similar fashion as argument to the command line. As a reminder, the command usage of the script starfish is:
starfish [**-o=***outputfile*] [**-e=***initialcode*] [**-replace**] [**-mode=***mode*] *file...*
The options are described below:

-o=*outputfile*

specifies an output file. By default, the input file is used as the output file. If the specified output file is '-', then the output is produced to the standard output.

-e=*initialcode*

specifies the initial Perl code to be executed.

-replace

will cause the embedded code to be replaced with the output. WARNING: Normally used only with **-o**.

-mode=*mode*

specifies the mode for the output file. By default, the mode of the source file is used (the first one if more outputs are accumulated using **-o**). If an output file is specified, and the mode is specified, then `starfish` will set temporarily the u+w mode of the output file in order to write to that file, if needed.

Those were the options.

appendfile *filename*, *list*

appends list elements to the file.

echo *string*

appends string to the special variable $0.

DATA FUNCTIONS

read_records($string)

The function reads strings and translates it into an array of records according to DB822 (db8 for short) data format. If the string starts with 'file=' then the rest of the string is treated as a file name, which contents replaces the string in further processing. The string is translated into a list of records (hashes) and a reference to the list is returned. The records are separated by empty line, and in each line an attribute and its value are separated by the first colon (:). A line can be continued using backslash (\) at the end of line, or by starting the next line with a space or tab. Ending a line with \ will replace "\\\n" with "\n" in the string, otherwise "\n[\t]" are kept as they are. Lines starting with the hash sign (#) are considered comments and they are ignored, unless they are part of a multi-line string. An example is:

```
id:1
name: J. Public
phone: 000-111

id:2
etc.
```

If an attribute is repeated, it will be renamed to an attribute of the form att-1, att-2, etc.

DATE AND TIME FUNCTIONS

current_year

returns the current year in string format.

file_modification_time

Returns modification time of this file (in format of Perl time).

file_modification_date

Returns modification date of this file (in format: Month DD, YYYY).

FILE FUNCTIONS

getfile($filename)

reads the contents of the file into a string or a list.

getmakefilelist($makefilename, $var)

returns a list, which is a list of words assigned to the variable `$var` in the makefile named `$makefilename`; for example:

```
FILE_LIST=file1 file2 file3\
  file4

<? echo join "\n", getmakefilelist $Star->{INFILE}, 'FILE_LIST' !>
```

Embedded variables are not handled.

putfile($filename,@list)

Opens the file `$filename`, wries the list elements to the file, and closes it. '`putfile` *filename*' will only touch the file.

A.8 STYLES

There is a set of predefined styles for different input files: HTML (html), HTML templating style (.html.sfish), TeX (tex), Java (java), Makefile (makefile), PostScript (ps), Python (python), and Perl (perl).

HTML Style (html)

HTML Templating Style (.html.sfish)

This style is similar to the HTML style, but it is supposed to be run in the replace mode towards a target .html file, so it allows for more hooks. The character `#` (hash) at the beginning of a line denotes a comment.

Makefile Style (makefile)

The main code hooks are <`?` and >.

Interestingly, the makefile style has similar special requirements as Python. For example, in the following expansion:

```
starfish: tmp
        starfish Makefile
        #<? if (-e "file.tex.sfish")
        #{ echo "\tstarfish -o=tmp/file.tex -replace file.tex.sfish" } !>
        #+
        starfish -o=tmp/file.tex -replace file.tex.sfish
        #-
```

it is convenient to have the embedded output indented in the same way as the embedded code.

A.9 STYLE SPECIFIC PREDEFINED FUNCTIONS

get_verbatim_file(*filename*)

Specific to LaTeX mode. Reads textual file *filename* and returns a string ready for inclusion in a LaTeX document. It untabifies the file contests for proper representation of whitespace. The function code is basically:

```
    return "\\begin{verbatim}\n".
           untabify(scalar(getfile($f))).
           "\\ end{verbatim}\n";
```

Note: There is no space betwen \\ and `end{verbatim}`.

htmlquote(*string*)

The following definition is taken from the CIPP project.

(*http://aspn.activestate.com/ASPN/CodeDoc/CIPP/CIPP/Manual.html*, link does not seem to be active any more)

This command quotes the content of a variable, so that it can be used inside a HTML option or <TEXTAREA> block without the danger of syntax clashes. The following conversions are done in this order:

```
      &  =>  &amp;
      <  =>  &lt;
      "  =>  &quot;
```

A.10 LIMITATIONS AND BUGS

The script swallows the whole input file at once, so it may not work on small-memory machines and with huge files.

A.11 THANKS

I'd like to thank Steve Yeago, Tony Cox, Tony Abou-Assaleh for comments, and Charles Ikeson for suggesting the include function and other comments.

A.12 AUTHORS

```
 2001-2020 Vlado Keselj http://web.cs.dal.ca/~vlado
           and contributing authors:
      2007 Charles Ikeson (overhaul of test.pl)
```

This script is provided "as is" without expressed or implied warranty. This is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

The latest version can be found at *http://web.cs.dal.ca/~vlado/srcperl/*.

A.13 SEE ALSO

There are several projects similar to Starfish. Some of them are text-embedded programming projects such as PHP with different programming languages, and there are similar Perl-based projects. When I was thinking about a need of a framework like this one (1998), I have found ePerl project. However, it was too heavy weight for my purposes, and it did not support the "update" mode, vs. replace mode of operation. I learned about more projects over time and they are included in the list below.

[ePerl] ePerl

This script is somewhat similar to ePerl, about which you can read at

http://www.ossp.org/pkg/tool/eperl/. It was developed by Ralf S. Engelshall in the period from 1996 to 1998.

php

http://www.php.net

[ePerl-h] ePerl hack by David Ljung Madison

This is a Perl script simulating the ePerl functionality, but with obviously much lower weight. It is developed by David Ljung Madison, and can be found at the URL: *http://marginalhacks.com/Hacks/ePerl/*

[Text::Template] Perl module Text::Template by Mark Jason Dominus.

http://search.cpan.org/~mjd/Text-Template/ Text::Template is a module with similar functionality as Starfish. An interesting similarity is that the output variable in Text::Template is called $OUT, compared to $O in Starfish.

[HTML::Mason] Perl module HTML::Mason by Jonathan Swartz, Dave Rolsky, and Ken Williams.

http://search.cpan.org/~drolsky/HTML-Mason-1.28/lib/HTML/Mason/Devel.pod The module HTML::Mason can also be seen as an embedded Perl system, but it is a larger system with the design objective being a "high-performance, dynamic web site authoring system".

[HTML::EP] Perl Module HTML::EP - a system for embedding Perl into HTML, by Jochen Wiedmann.

http://search.cpan.org/~jwied/HTML-EP-MSWin32/lib/HTML/EP.pod It seems that the module was developed in 1998-99. Provides a good CGI support, run-time support, session handling, a database server interface.