Report on Starfish v. 1.10
A Perl-based System for Text-Embedded Programming and Preprocessing

(Starfish Version 1.10, Document Revision 1.6)

Vlado Kešelj

September 19, 2007

Abstract

This report is meant to be the most up-to-date documentation on Starfish. A large part of it is a direct POD Documentation generated from in-code documentation.

Contents

Chapter 1

Introduction

1.1 Text-Embedded Programming

1.2 Overview

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

Chapter 3

Implementation

⋮

3.1 Hooks

⋮

The regex hooks pass captured substrings as argumements to the replacement func-
tion. If the whole captured string (`$&`) is needed, it can be obtained from `$self->{currenttoken}`.

⋮

Chapter 4

Starfish Reference

4.1 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.12. 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.

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
```

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.  */

//<? $0 = defined($Release) ?
// "public class p {\n" :
// "public class p_t {\n";
//!>//+
public class p_t {
//-

  public static int main(String[] args) {

    //<? $0 = "    ".(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
```

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 contetns 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->{INFILE}

Name of the current input file.

$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; !>-->
```

$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`, and per-component copies the array `hook`.

$o->addHook

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 string 'ignore'

> equivalent to producing no echo,

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 ad-
> dHook understands only replacement 'comment', which 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 mathched 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->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*)

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

getinclude(*filename and options*)

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 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*)

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.

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

> appends stuff to the special variable $O.

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).

getfile *file*

> grabs the content of the file into a string or a list.

getmakefilelist *makefile*, *var*

> returns a list, which is a list of words assigned to the variable *var*; e.g.,

```
FILE_LIST=file1 file2 file3\
  file4

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

> Embedded variables are not handled.

htmlquote *string*

> The following definition is taken from the CIPP project.
>
> (*http://aspn.activestate.com/ASPN/CodeDoc/CIPP/CIPP/Manual.html*)
>
> 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;
```

putfile _filename_, _list_

>   opens file, writes the list elements to the file, and closes it. 'putfile _filename_'
>   "touches" the file.

read_records _string_

>   The function takes one string argument. If it 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 \\ effectively removes
>   the "\\\\n" string at the end of line, but "\n[\t]" combination is replaced
>   with "\n". Comments, starting with the hash sign (#) are allowed between
>   records. 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.

read_starfish_conf

>   Reads recursively (up the dir tree) configuration files `starfish.conf`.

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   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.10   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.11   AUTHORS

```
2001-2007 Vlado Keselj http://www.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://www.cs.dal.ca/~vlado/srcperl/*.

A.12   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.

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.

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.