Report on Starfish v. 1.32
A Perl-based System for Text-Embedded Programming and Preprocessing

(Starfish Version 1.32, Document Revision 392)

Vlado Kešelj

July 2, 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 text-embedded programming and preprocessing. It demonstrates a 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

This report describes the system Starfish—a system for Perl-based text preprocessing and text-embedded programming. In this introduction, we will try to make a case for a need for such system, and its design, implementational, and maybe even philosophical goals.

1.1 Preprocessing and Text-Embedded Programming (PTEP)

There is a wide need in computer science for *text preprocessing* and *text-embedded programming,* or shorter *Preprocessing and Text-Embedded Programming (PTEP).*

We define *text preprocessing* as any operation that takes text as input and produces a similar text as output; and it serves as a way to automate manual editing of the text. It is called "preprocessing," since there is normally some standard use of that type of text that would be called "processing;" such as compilation or interpretation of a programming language, rendering of an HTML page, and similar. A typical example of a preprocessor is the C programming language preprocessor [4], which is mostly used for simple text inclusions or exclusions based on some configuration parameters and simple text replacements, before the C program code is passed to the compiler.

Text-embedded programming is a related but generally different concept than text preprocessing. We define *text-embedded programming* as any form of computer programming where code is embedded in an arbitrary text, and can be executed in-place, in that text context. One of the first examples of text-embedded programming can be considered the TEX typesetting system by Donald Knuth, released in 1978. [10] TEX has its own language of annotating text to prepare it for text typesetting and printing in form of papers, books and similar documents; but TEX language also includes a "macro" language for text transformation in-place before

the final preparation of the output pages. This macro part of the language is a form of preprocessing but also text-embedded programming because it is Turing-complete and one can write a general-purpose program in this language. The TEXbook [10] contains a famous example of a table of prime numbers, generated in the TEX macro programming language, written by Knuth.

The second, more obvious example of text-embedded programming is the PHP programming language [14]. A PHP program file is usually an HTML file with the snippets of PHP code inserted in the file. The file is processed, or we could say preprocessed, before being delivered to the HTML browser in such way that the PHP snippets are replaced with their output produced using the command `echo`. The snippets are delimited with the strings `<?php` and `?>`, or simply with `<?` and `?>`. This model is particularly convenient for fast development of web apps, where we can start with a static (pure) HTML page and incrementally replace pieces with dynamic PHP-generated code. A similar approach was used in ASP (Active Server Pages) [19] engine or JSP (JavaServer Pages) [18], both of which use `<%` and `%>` delimiters for code snippets.

The third example of text-embedded programming is the project Jupyter, formed in 2015, which supports inclusion of Python, Julia, or R programming language snippets in a file called Jupyter Notebook [7]. Although this example of text-embedded programming is not transparent in the sense that a Jupyter Notebook is not a plain text file, it is still very close to a text file (it is in the JSON format), marked in a language called Markdown, which gets translated into HTML, and it allows inclusion of arbitrary code in Python (or other allowed language), that can be executed. The result of the execution is shown in the notebook itself. This is a novelty, compared to PHP for example, that we will call the *update mode*, vs. the *replace mode* used in PHP, ASP, JSP, and similar template languages.

1.2 Goals of Starfish Development

Starfish development and use with that name started in 2001. These lines of documentation are written some 19 years later, so a natural question is whether there is still the same level of novelty in it, and whether there exist systems with needed functionality. The main goal of Starfish development is a universal preprocessing and text-embedded programming (PTEP), and I am not aware of another similar effort. I find the Perl particularly suitable to be an embedded language, and even more surely there is no such universal system with Perl as an embedded language. I understand that this particular Perl orientation may be seen as a subjective personal preference, but still it may represent a significant advantage to a portion of readership.

The main feature of Starfish is universal Preprocessing and Text-Embedded Pro-

gramming, which is a goal that that the system can be used in many different textual contexts, such as programming languages, web languages (e.g., HTML, CSS), data formats (e.g., JSON), typesetting languages (e.g., TeX, LaTeX), and others (e.g., make, procmail).

Some of the main Starfish features that were the reason and goal of its development are:

- **Universal Preprocessing and Text-Embedded Programming (PTEP):** is the goal of creating a universal system that can be used for PTEP for various types of text files (e.g., HTML, LaTeX, Java, Makefile, etc.)

- **Update and Replace Modes:** Starfish supports two modes of operation: *replace mode* — similarly to PHP or C preprocessor, where the snippets are replaced with the snippet output and the complete output is saved in the output file or produced to the standard output; or *update mode* — similarly to Jupyter, where the snippet output is appended to the snippet in the updated source file.

- **Flexible PTEP:** Starfish is flexible, in the sense that we can modify the patterns that are used to detect active pieces of text. The basic pattern used to detect and execute code snippets, can be generalized to make active "hooks" from any string, pair of delimiters, or regular expression pattern. Starfish also provides flexibility in defining the way snippets (active code) is evaluated.

- **Configurable PTEP:** Starfish allows user-defined configuration per directory, and it uses directory hierarchy for a more wide configuration definition.

- **Transparent PTEP:** Starfish provides transparency in the sense that when a file with Starfish snippets is processed, assuming it is a static file, we do not need Starfish any more to use it. For example, an HTML file is still an HTML file viewable by a browser, a LaTeX file is still a LaTeX file processable by LaTeX, and so on. As a comparison, a Jupyter file is a special-format JSON file, which needs to be processed to produce HTML, LaTeX, or other forms usable by a user.

- **Embedded Perl:** Even though the main principles of Starfish could be implemented in many languages, Perl is particularly convenient for both Starfish implementation, use in code snippets, and configuration. As a comparison, TEX uses its own language for PTEP and it is difficult to use since its paradigm and notation style are so different from the main main-stream languages. The C preprocessor works well for C, but in attempts to use it in other systems, like `Imake` for the `Makefiles`, it was not very successful and it was difficult to use since it was not meant for that kind of context. Using a general-purpose

language that is used for other purposes has a clear advantage, and Perl succinctness and expressiveness in working with strings in particular, makes it an excellent candidate.

1.3 Java Preprocessor Example

Let us start in the introduction with one example to illustrate Starfish operation. As we mentioned, a preprocessing example is the C preprocessor, which is a useful and unique feature of the C programming language. It is a part of the C compiler, but it is a simple language in its own, which does simple text manipulation before feeding it to the proper C compiler. One use of the preprocessor is inclusion or exclusion of parts of code depending on values of some configuration variables. It preprocesses C source code as a general text, without a detailed use of C syntax or semantics. It is sometimes criticized for not using deeper semantics of the language, and it is also praised for the same reason because it is very clear what it does and it can be used on text other than C programs. For example, it was used in the `Imake` system [21, 2] for preprocessing `Makefiles` [**?**, **?**]. Java does not have a preprocessor and it would be useful in some situations.

Sometime around 2001, I was working on a Java software system where I needed two versions of source code: a test version to be used for testing and development, and a release version to be the production release. The test version could carry around a lot of meta information on data structures, be able to produce verbose debug code, make additional expensive run-time checks, and simlar, while the release version would be efficient and slim in code size and running time. This means that at various places in the source code, I needed to write two versions of code snippets: a test version and a release version, and the appropriate version would be included everywhere based on the value of some global variable. This could be simulated using Java constructs, but the release code would be bloated and running-time efficiency of the release code would not be easy to achieve.

As an example, we will consider the following simple Java code:

```
/**
   A simple Java file.
*/

public class simple {

  public static in main(String[] args) {

    System.out.println("Test version");
    System.out.println("Release version");
```

```
    return 0;
  }
}
```

where the red line would be included in the test version of the code, and the blue line would be included in the release version of the code.

One solution would be to use the C preprocessor. However, the C preprocessor is a part of the C compiler and it is not meant and not convenient to use independently. Its functionality is tailored to the C language, and it is not as easy to use for general and more flexible text processing that we may want to have. It is more convenient to write a text processor in Perl from scratch than to rely on the C preprocessor. That leads to the second solution: write an independent preprocessor from scratch in text-friendly high-level language like Perl. This is how we can get an idea of a general-purpose preprocessing system. The system `m4` [**?**, **?**] is one such system, but it is limited to general-purpose macro processing, has its own, specific syntax, and it does not support the update mode of operation.

1.3.1 Fully-Embedded Preprocessor

There are many ways we could approach our preprocessing task, which basically need to include or exclude annotated parts of code. Similarly to the C preprocessor, we could read our source Java file and produce another Java file, which will then be ready for compilation. To distinguish these two files, we could come up with a different name extension for the first Java file, which we could call a *meta-source code* file. One issue with this approach is that we now must manage two files for the same Java source file, and the second issues is solving the question of how exactly our preprocessor should look like. We could emulate the functionality of the C preprocessor, but designing a new universal preprocessor would allow us to think bigger and aim at a more open-ended general functionality. Both of these issues are addressed with a *fully-embedded preprocessor*, which combines preprocessing instructions and preprocessing result in the same file, and allows for a quite general Perl preprocessing code. Starfish provides this functionality.

Our example Java file could be written in the following way using the Starfish convention:

```
/**
   A simple Java file.
*/
// Uncomment version:
//<?   $Version = 'Test';    !>
//<? # $Version = 'Release'; !>
```

```
public class simple {

  public static int main(String[] args) {

    //<? $O = "    ".($Version eq 'Test' ?
    // 'System.out.println("Test version");' :
    // 'System.out.println("Release version");' );
    //!>

    return 0;
  }
}
```

Starfish code is embedded perl code found between delimiters `<?` and `!>`, and it is commented out using the Java line comment notation `//`. The blue and red lines are used to choose version of the software that we want to produce. The red line contains code commented out in Perl, so that chosen version is the "Test" version. The green snippet code shows how we can select the appropriate line of Java and produce it. The Perl variable `$O` is used as a special variable to specify the generated code. Starfish has also a command `echo` that effectively appends to this variable.

The result of preprocessing is not a new file, but the source file is actually updated. We call this the *update mode* of Starfish and it is the default mode. This is why we call Starfish a *fully-embedded preprocessor*. If the name of the Java file is `simple.java` then we would run the following command:

```
$ starfish simple.java
```

and the contents of the file `simple.java` is now:

```
/**
   A simple Java file.
*/
// Uncomment version:
//<?   $Version = 'Test';    !>
//<? # $Version = 'Release'; !>

public class simple {

  public static int main(String[] args) {

    //<? $O = "    ".($Version eq 'Test' ?
    // 'System.out.println("Test version");' :
```

```
    // 'System.out.println("Release version");' );
    //!>//+
    System.out.println("Test version");//-

    return 0;
  }
}
```

We can see that the desired line of code has been generated and inserted in the file (magenta-colored part). The generated part is delimited with strings `//+` and `//-`, so if we run the `starfish` again on the file, the file will not be changed because the generated part would be replaced with the same generated string. If by coincidence our output code contains the string `//-`, Starfish will insert a number in the delimiters, e.g., `//3+` and `//3-` so that the ending delimiter is not confused with the generated code.

If we comment out the 'Test' line and uncomment the 'Release' line in the new `simple.java` file as follows:

```
/**
   A simple Java file.
*/
// Uncomment version:
//<? # $Version = 'Test';    !>
//<?   $Version = 'Release'; !>

public class simple {

  public static int main(String[] args) {

    //<? $0 = "    ".($Version eq 'Test' ?
    // 'System.out.println("Test version");' :
    // 'System.out.println("Release version");' );
    //!>//+
    System.out.println("Test version");//-

    return 0;
  }
}
```

and run:

```
 $ starfish simple.java
```

again, the file `simple.java` file will look as follows:

```
/**
   A simple Java file.
*/
// Uncomment version:
//<? # $Version = 'Test';    !>
//<?   $Version = 'Release'; !>

public class simple {

  public static int main(String[] args) {

    //<? $O = "    ".($Version eq 'Test' ?
    // 'System.out.println("Test version");' :
    // 'System.out.println("Release version");' );
    //!>//+
    System.out.println("Release version");//-

    return 0;
  }
}
```

Since we can include arbitrary Perl code in the snippets, including imports of external libraries and code, this framework provides a very general way of code preprocessing. Starfish includes a few more features to support wider management of code base within a directory, which we will discuss in the next couple of subsections.

1.3.2 Preprocessing Multiple Files

If we want to preprocess a number of Java files in a project, it would be tedious and error-prone to modify each of them to set them to the appropriate Test or Release version. There are several ways how this problem could be solved and we will describe three of them:
(1) using Perl `require` command,
(2) using Make and Starfish `-e` option, and
(3) using the Starfish `starfish.conf` configuration file.

(1) Using Perl `require` command: To have one `$Version` parameter controlling many files, we could simply have a Perl file called `configuration.pl` with the following content:

```
#!/usr/bin/perl
$Version = 'Test'; # Test or Release
1;
```

and one of the first lines in each Java source file would be:

```
//<? require 'configuration.pl' !>
```

In this way, we would have one point of control for the Test or Release version of all files.

(2) Using Make and the Starfish `-e` option: Starfish has an option `-e` for initial Perl code execution, somewhat similar to Perl, and we can use it to set the Version variable. For example, if we use a Makefile to compile all Java files in a project, we could add a preprocessing command for each of them in the following way in the Makefile:

```
VERSION=Test
#VERSION=Release

simple.class: simple.java
        starfish -e='$$Version="$VERSION"' $<
        javac $<
```

We would again have one point of version control, this time in the Makefile.

(3) Using the Starfish `starfish.conf` configuration file: The idea of using a Perl configuration file, as shown in (1), is so common in many situations that we use a standard name for the configuration file called `starfish.conf` to include this information. Similarly to (1), the contents of the file `starfish.conf` would be:

```
#!/usr/bin/perl
$Version = 'Test'; # Test or Release
1;
```

and one of the first lines in each Java source file would be:

```
//<? read_starfish_conf !>
```

This is the common way to represent per-directory configuration in Starfish. One important difference between this approach and the earlier approach with the standard Perl configuration file (1) is that `read_starfish_conf` behaves in a special way. Namely, the command `read_starfish_conf` will look for a file named `starfish.conf` in the current directory; if found, it will then look for the same named file in the

parent directory. Again, if it is found, it will look into the parent of the parent directory and so on until it cannot find a file with that name, or until it reaches the top directory. After that, it will execute, or more precisely `require` in the Perl terminology, all found files `starfish.conf` from top to bottom. Each file is executed in its own directory as the current directory. This provides for a hierarchical per-directory configuration. A similar process is used sometimes in the system of Makefile in a project with multiple directories [5], and in the Imake system for Makefile generation.[2, 21]

1.3.3 Replace Mode

Finally, if we want to produce a version of Java code without preprocessing code, we can use the Starfish *replace mode.* In this mode, the preprocessing code is removed as well as markup around the generated code. We must specify an output file in the replace mode because we normally do not want to permanently loose the preprocessing code. For example, if we run he following command:

```
$ starfish -replace -o=release/simple.java simple.java
```

on the above file in which `$Version` variable is set to the value `"Release"`, the resulting file `release/simple.java` would contain the following contents:

```
/**
   A simple Java file.
*/
// Uncomment version:



public class simple {

  public static int main(String[] args) {

        System.out.println("Release version");

    return 0;
  }
}
```

With this, we would like to finish this introductory Starfish example.

1.4 The Name of the Game

The parts of Starfish system were developed from 1998, and then in 2001 I searched for other systems that would have features that I needed and have been working on. I did not find a good match, but there were a number of systems with with some similarity. The closest one was ePerl [3] system created by Ralf S. Engelshall in 1998, implemented in C but using an installed Perl system. There was also a Perl implementation of the same ePerl system by David Ljung Madison [12]. The system was meant to be used for at least two different text styles: plain text and HTML, and its vision and philosophy were generally very close to my goals. However, it was not a close enough match, so I started a new project, and for a lack of better name, I called it SLePerl, which was short for "Something Like ePerl". At some point during 2001, a friend of mine at the graduate school asked "What is SLePerl!?" and we started pondering about the better name. Since the system runs on Perl code snippets in text, a metaphor could be a shellfish with embedded pearls. We agreed that Oyster sounded like a great name, but there was actually (yet another) Perl module for embedded Perl called `Text::Oyster` [13]. Starfish seemed to be a good name, since it feeds on shellfish, with an exception that the system Starfish feeds on Perl snippets within text. The module Text::Starfish was created in 2002 and uploaded to the CPAN soon after. It was described in an article in the Perl Journal in 2005. [9]

1.5 Overview

After this introductory chapter, in Chapter 2 we discuss background and related work; Chapter 3 presents user documentation, Chapter 4 describes the system design, and Chapter 5 gives more starfish details for reference, and Chapter 6 is a conclusion. Appendix A contains the POD documentation. This documentation 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

Background and Related Work

We will describe in this chapter some background information on Preprocessing and Text-Embedded Programming (PTEP), and some existing related work in this area. The PTEP area does not exist as a recognized coherent area, but there has been a lot of fragmented related work within the context of different programming languages, and different applied area of Computer Science, such as in web systems development (PHP, ASP, JSP), software development (C preprocessor, make, imake), electronic publishing (TEX, LATEX), and machine learning and data science (Jupyter).

Text and text files: We define *text* to be any string of characters, generally including the new-line character, and it will typically be saved in a file, which we call a *text file.* We will assume characters to come from the ASCII set, but they may include extended ASCII (i.e., numerical values from 0 to 255), or they may have UTF-8 encoding, so characters may be from the Unicode set. A text is usually created manually in a plain-text editor, such as `emacs` or `vi` in Linux or other Unix-like systems, or `notepad` in Windows OS. If the text follows certain formal rules (grammar), we will say that text is in certain *style*. Otherwise, if we do not recognize a particular formal grammar of the text, we will say that text has a *default style*. It could be for example general natural language text, such as English, or any kind of text that is not on our list of recognized styles. We will also talk about specific styles, such as the C-program style, if the text is a program in the C programming language, a Java style, an HTML style, LATEX style, and similar.

Text preprocessing: We define *text preprocessing* to be an operation that takes text as input and produces a similar text as output, and it serves as a way to automate manual editing of the text. Again, this is not a very precise definition, an we will have to rely on some of our common sense and experience in recognizing what constitutes preprocessing. The name "preprocessing" comes from the idea that this operations does not change the main style of the text, and it is done before

any proper *processing* designed for this style of text, such as compilation of a C program, rendering of an HTML page, or translating a LaTeX text into a PDF document. A typical representative preprocessor is the C programming language preprocessor [4, 20].

2.1 Text-Embedded Programming

We define *text-embedded programming* as a form of computer programming where programming source code is embedded in text of arbitrary style, and this code can be executed in place; i.e., in the original embedded context.

One could argue that any programming source code is embedded, since code is generally mixed with documentation comments, but there is a significant conceptual difference in thinking about a text file as a program with comments, rather than a text of arbitrary style, with some code snippets inserted. We also leave some freedom in how the code snippets are executed, to what purpose, or how they interact with the surrounding text. However, we will see very soon some typical usage for such snippets. Before that, let us define some notions in this model.

In text-embedded programming, programming source code is embedded in text as a sequence of continuous text segments. These segments are sometimes called *code snippets*, *active code*, or *live code*. The text outside the segments is called *outer text*. The code snippets are usually easily recognizable by defined text delimiters, but depending on the rules that we use, any text can be recognized as a snippet. This is why the name *active code* is very appropriate: The TeX system uses a labeling of all characters at run-time that can denote any character to be an 'active' character, and as such initiate special processing after the system reads this character. This character is also sometimes called an *escape* character. A similar generalized approach is adopted in Starfish, in which the active code is recognized by *hooks*.

One of the first examples of text-embedded programming was the TEX typesetting system developed by Knuth, released in 1978. [10] The system processes text and prepares it for typesetting pages for print, but in the process it recognizes TEX commands by detecting the *escape* backslash character (`\`), i.e., an *active* character, which triggers special execution behaviour based mostly on macro expansions. This macro expansion model can be regarded as a computation model; i.e., model for code execution, but it is difficult to learn for coding purposes as indicated by the author itself. There were approaches to developing TEX preprocessors in other languages such as Lisp, as published by Iwesaki in 2002 [6].

Text-Embedded Programming is particularly popular and useful in the context of HTML documents. The HTML language was after its design mostly used for creationg of static documents, viewable and browsable by users, and a very natural

way to make the documents dynamic through programming is by inserting code snippets in HTML pages. After execution of the snippets, they are replaced with the generated text output, and the resulting page is used for viewing and browsing. This model is used in the very popular PHP language [14], and also in ASP (Active Server Pages) [19] and JSP (JavaServer Pages) [18].

The code snippets are marked in text with starting and ending delimiters, which are arbitrary small strings. Other than simple markers for snippets, we can think of them as escape sequences that toggle on and off code processing. For example, the 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>
```

where we show snippet delimiters in red, and the snippet itself in blue color. 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>
```

where we show the generated output in the green color. Embedding the code in this 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 or *outer text*, is embedded between '`?>`' and '`<?`php' delimiters and it is translated into an '`echo "`*string*`";`' statement; and similarly, 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 this type of escaping 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 principle that each snippet should be a well-defined block of code. If we want large pieces of outer text to be conditionally included or excluded, Perl offers many string delimiting options for large text segments, such as `q/.../` and `<<'EOT'`, which can be used in place of inverted escaping.

2.2 Perl-based Embedded Programming

We will describe here some previous work on Perl-based embedded programming. Our vision for universal PTEP 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 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.[3] The language ePerl was developed in the period from 1996 to 1998. It is an embedded Perl language in the sense that we described, but ultimately there were several reasons why it did not fit needed requirements: (1) it seemed to be too heavy-weight, (2) it did not support the *update* mode, that will be described in the next chapters, and (3) it does not the universal PTEP approach with multiple text styles. Other authors [12] also noted that ePerl is heavy-weight: The language ePerl was a binary package created by modifying the Perl source code and requiring compilation during installation. We prefer a more light-weight solution that relies on the standard Perl, and our solution is installed as a Perl module.

For example, David Ljung Madison developed an "ePerl hack" [12] which is a Perl script of some 1400 lines that has functionality similar to ePerl.

Text::Template [1] by Mark Jason Dominus is another Perl module with similar functionality. It is a very popular module designed to "expand template text with embedded Perl", created in 1995 or 1996 and maintained with contributions by many users until now. An interesting and probably independent similarity is that Starfish uses `$O` as the output variable, while `$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 philosophy of Text::Template module

has a lot of similarity with Starfish, however the Text::Template module is primarily meant to be used in templating style; which means that a template file is created as a more passive object and it always requires a handling Perl script to generate the output target file. An additional difference is that the Text::Template module does not support the update mode. The use of default delimiters creates issues with JavaScript code, although there are workarounds. The system is not applied to many text styles other than plain text and HTML.

Another well-known Perl module HTML::Mason [15], authored by Jonathan Swartz, Dave Rolsky, and Ken Williams, can also be seen as an embedded Perl system. It is a larger system with the major design objective to be a high-performance, dynamic, web-site authoring system.

A relatively minimalistic approach is used in development of the module Text::-Oyster [13] by Steve McKay in the period 2000–3. The module is template module for evaluating Perl embedded in text between delimiters '`<?`' and '`?>`'.

HTML::EP [16] is another Perl module for embedding Perl into HTML. Its specific approach is that code delimiters are HTML-like tags that start with '`ep-`'. For example, comments are delimited by `<ep-comment>` and `</ep-comment>`, and active code is delimited with tags `<ep-perl>` and `</ep-perl>`. The last value in the embedded code is the generated string. The module is meant to be used in a dynamic way over the Apache web server and the use of Apache module `mod-perl`, so the documentation gives a nice overview of how to set up a Perl module that supports embedded programming to run efficiently in this setting. The set of tags is further extended, so it includes `<ep-email>` for generating emails from a web page, `<ep-database>` and `ep-query` for working with a database, `<ep-list>` for generating HTML lists, then conditionals, and so on. It is an interesting idea that in text embedding like this we can modify the language to be simpler in some situations than Perl, but it is still not clear that it is justified to introduce all these new constructs, when equivalent Perl code exists.

Starfish is a lighter-weight system than eperl or Mason, but it is more flexible and universal than Text::Template, the ePerl hack, and HTML::EP. Starfish has a large coverage goal of covering more text styles than other systems, provides other unique innovations, such as more flexibility in defining active code detection patterns, per-directory configuration, update mode, and full embedding when compared to some systems. Under *full embedding* we refer to capability that all functionality and customizability, such as adaptation of patters, can be achieved with code inside the snippets embedded in the text file.

2.2.1 PTEP in the Update Mode

We can have generally different ways in which embedded code executed and how its output is used. For example, even the concept of Literate Programming [17]

introduced by Knuth in 1984 [11] can be considered text-embedded programming, although the code is only executed after it is automatically gathered into the source files, and then compiled.

All the systems that we discussed until now in this section, support a more of execution that we will call the *replace mode.* In the *replace mode,* the code snippets are replaced with the output of those snippets, and the file produced in this way is either sent over internet to a brower to be viewed, or saved into a target file. The Starfish system was designed to support a new mode of operation, called the *update mode,* from its initial public release in 2001 [8]. This mode was briefly discussed in the Perl Journal article in 2005 [9]. The main property of the update mode is that rather than replacing the code snippets with their output, the output is appended to snippets. This has several advantages: (1) we do not need to set up translation process from source file to target files, which makes the proces simpler; (2) it provides an easy inspection of the embedded code and the output it produces, which is very convenient in prototyping process, for example; and (3) it provides an easy way for the system to be used as a preprocessor for text files of arbitrary style. We will describe in more detail how the update mode works, but it should be mentioned in the related work since a well-known system Jupyter [7, **?**], released in 2015, operates in the update mode. The Jupyter system is used to produce so-called Jupyter Notebooks, which are JSON-type files with a mixture of plain text and embedded Python code. The execution of the notebook appends the output of embedded code immediatelly to the code. This is used to create documents in which the code and results of code execution are intermixed. We would describe this as text-embedded programming with the update mode, with a minor exception that Jupyter Notebook itself is not in plan-text format but needs a viewer software to be presented in that way.

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 `$sf->process_files` expects a list of files. Each file is processed and written out. The main part of processing start with setting `$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).

Chapter 6

Conclusion

To be added.

Bibliography

[1] Mark Jason Dominus and et al. Perl module text::template, 1999-2019 (accessed Jun 23, 2020). `https://metacpan.org/pod/Text::Template`.

[2] Paul DuBois. *Software Portability with imake*. O'Reilly Media, 2nd ed. edition, September 1996. `https://archive.org/details/softwareportabil00dubo`.

[3] Ralf S. Engelshall. Ossp eperl: Embedded perl language, 1996–8 (accessed Jun 23, 2020). `http://www.ossp.org/pkg/tool/eperl/`.

[4] GCC.GNU.org. The C preprocessor, 2020 (accessed Jun 23, 2020). GNU GCC Documentation, `https://gcc.gnu.org/onlinedocs/cpp/`.

[5] GNU.org. Gnu make: 5.7 recursive use of make, 2020 (accessed Jun 23, 2020). `https://www.gnu.org/software/make/manual/make.html#Recursion`.

[6] Hideya Iwesaki. Developing a Lisp-based preprocessor for tex documents. *Software: Practice and Experience*, 32(14):1345–1363, 2002.

[7] Jupyter.org. Project Jupyter, 2020 (accessed Jun 22, 2020). `https://jupyter.org`.

[8] Vlado Kešelj. Perl module text::starfish and starfish: A perl-based system for preprocessing adn text-embedded programming, 2001–20 (accessed Jul 1, 2020). `https://metacpan.org/pod/Text::Starfish`.

[9] Vlado Kešelj. Starfish: A Perl-based framework for text-embedded programming and preprocessing. *The Perl Journal*, June 2005.

[10] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Reading, MA, USA, 1986.

[11] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[12] David Ljung Madison. eperl (rewrite in perl), 2001–8 (accessed Jun 23, 2020). `http://marginalhacks.com/Hacks/ePerl/`.

[13] Steve McKay. Perl module text::oyster: Evaluate perl code embedded in text, 2000–3 (accessed Jun 23, 2020). https://metacpan.org/pod/Text::Oyster.

[14] PHP.net. PHP: Hypertext Preprocessor, 2020 (accessed Jun 19, 2020). http://www.php.net.

[15] Dave Rolsky, Jonathan Swartz, Ken Williams, and et al. Perl module html::mason, 1998-2020 (accessed Jun 23, 2020). https://metacpan.org/pod/HTML::Mason.

[16] Jochen Wiedmann. Perl module html::ep, 1998-2001 (accessed Jun 23, 2020). https://metacpan.org/pod/HTML::EP.

[17] Wikipedia.org. Literate programming, 1984 (accessed Jul 1, 2020). https://en.wikipedia.org/wiki/Literate_programming.

[18] Wikipedia.org. Javaserver pages, 1999 (accessed Jun 23, 2020). https://en.wikipedia.org/wiki/JavaServer_Pages.

[19] Wikipedia.org. Active server pages, 2000 (accessed Jun 23, 2020). https://en.wikipedia.org/wiki/Active_Server_Pages.

[20] Wikipedia.org. C preprocessor, 2020 (accessed Jun 23, 2020). https://en.wikipedia.org/wiki/C_preprocessor.

[21] Wikipedia.org. imake, 2020 (accessed Jun 23, 2020). https://en.wikipedia.org/wiki/Imake.

Appendix A

POD Documentation

A.1 NAME

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

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. Starfish is chosen as natural starfish feeds on shellfish, with a difference that the system `starfish` feeds on embedded pearls.

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 ^ 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, makefile, perl, ps, python, and tex (same as latex, 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.

Setting a style can have various side effects, but it typically involves setting the following variables:

```
$o->{Style}            # style string id
$o->{CodePreparation}  # function to clean the code before running
$o->{LineComment}      # string starting a line comment
$o->{OutDelimiters}    # array ref with four elements: $b1, $b2 for
                       # starting output delimiter, and $e1, $e2 for
                       # the ending output delimiter; $b1 and $e1
                       # must not end with a digit, and $b2 and $e2
                       # must not start with a digit
$o->{IgnoreOuter}      # boolean variable to ignore outer text, false
                       # by default
$o->{hook}             # array ref, list of hooks
```

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.