The Text Engine

By Francesc Hervada-Sala (mail: francesc at the-text.net)

Published Sep 5, 2015. Revised Nov 26, 2015.

URL: http://the-text.net/text-engine

Abstract: This paper describes a projected experimental software for handling data that aims to serve as the foundation for further applications. This system is based on a single data structure—the «text»—and will not only facilitate data management, but also enable multiple applications to smoothly share data and cooperate. Moreover, all applications can be easily combined and extended both by other applications and by power users.

Contents

The intention of this paper is to introduce readers to a long-term project I have been working on. My motivation arises from the conviction that current software falls far short of its potential. I often asked myself, «Why is everything so complicated?» As a user, I had grown frustrated about having to deal with several stubborn autarchic applications that forced me to describe interconnected facts as unrelated pieces, and to bridge the resulting gaps through manual means (e.g., exporting and importing, copying and pasting). As a programmer, I felt discontent with having to use inflexible and clumsy programming systems that compelled me to write very similar code anew from scratch repeatedly. We could automate so much work and exploit so many logical dependencies! This brings us to the approach that will tackle these issues. I posit that we should build software systems that «describe» facts instead of «implement,» «simulate,» or «make analogies» with them. After all, describing facts is what we do with language, and notably, a single spoken language can fit all fact-describing purposes within it. Let us make computers handle «text,» not character strings, but parsed text; not words, but their meanings—the symbolic figures that populate our minds.

I intend to develop and test this approach with the project delineated in this paper: the Text Engine. The Text Engine is experimental software that will provide mechanisms for creating, storing, manipulating, and querying a data structure simply called «text.» Applications will use the Text Engine for both data handling and as an interface between them. In the following pages, readers will become acquainted with the basic points of the system, and later, its applications.

Fundamentals

Data: Text Structure

The Text Engine supports a symbolic language with which any text can be represented by specifying symbols and references between them. To declare a symbol, precede its name with a tilde and an equal sign:

```
~=species;
```

Subordinated symbols can then be added for every attribute that «species» will have:

```
~=species { ~=common name :string; ~=scientific name :string };
```

The curly brackets identify «child» symbols. Here, the «common name» and the «scientific name» each have a particular «type» (preceded by a colon). In this case, the data type is «string.» Having already defined the type «species,» one can then declare subtypes of it:

```
~=mammal :species; ~=elephant :mammal;
```

A symbol inherits all characteristics from its type, which in turn inherits them from its own type, and so on. For example, an elephant is a mammal and a mammal is a species, therefore each elephant species has a common and a scientific name, as well:

```
=African elephant ~elephant {
  ~common name "African bush elephant";
  ~scientific name "Loxodonta africana";
};
```

The expression above defines a particular elephant species with the given common and scientific name. The symbol names are not being defined here, just used; thus, they are preceded with only the tilde sign representing their «role.» We are saying that the character string «African bush elephant» plays—for this species—the role of the common name.

Suppose the user now wants to register a particular species of penguin. This can be achieved as follows:

```
~=penguin :species { ~=breeding pairs :integer };
=little penguin ~penguin {
  ~common name "Little blue penguin";
  ~scientific name "Eudyptula minor";
  ~breeding pairs "500,000";
};
=African penguin ~penguin {
  ~common name "African penguin";
  ~scientific name "Spheniscus demersus";
  ~breeding pairs "70,000";
};
```

The Text Engine allows users to create symbols and their references freely, provided they follow the text integrity rules. Every symbol used must be declared. Every child symbol must play a role compatible with its parent's type. For example, a penguin's child element can take on the role «breeding pairs,» because the definition of «penguin» includes this child element, whereas a child element of «elephant» cannot. Moreover, the data contained in this child element must be, according to the definition of «breeding pairs,» an integer.

Users can also reuse symbols. For example, to include them in a list:

```
~=threatened species { ~=endangered :species; ~=vulnerable :species };
~threatened species {
  ~endangered #African penguin;
  ~vulnerable #African elephant;
};
```

The number sign (#) represents symbolic equality. The above expression means that the symbols «African penguin» and «African elephant» themselves (not copies of them or instances of them as a type) belong to the list of threatened species.

A symbol can have more than one type. For example, elephants are worshiped in some cultures:

```
=Asian elephant ~elephant { ~scientific name "Elephas maximus" };
~=veneration object { ~=religion :world religion };
#Asian elephant :veneration object { ~religion #Hinduism };
```

This way, an Asian elephant is registered as both a biological species and an object of veneration, and the text repository holds data about both aspects under a single symbol.

To summarize, «text» in this engine consists of «text units»—each of which has a parent, a type, and plays a role. A symbol can be defined by a single text unit or by more than one if grouped by symbolic equality.

The above is not to be confused as the fundamental data structure of the Text Engine. The root data structure is the aforementioned text unit, defined as «one or more references to other text units, with each reference written as a pair (genus, reference point).» Expressed in symbolic language:

```
*genus 1 >reference point 1 *genus 2 >reference point 2 ...
```

Both the genus and reference point are text units. The genus (prefixed with an asterisk) indicates the meaning of the reference. The reference point (prefixed with a greater-than sign) is the unit that the reference targets. A unit has one or more references, while it cannot have more than one of the same genus.

The first layer of the Text Engine creates the basic genera:

```
parent
type
role
symbol
```

The colon (:), tilde (~) and number sign (#) prefixes are not primitives of the language, but shortcuts for the basic genera. Take the following symbolic language token:

```
~endangered #African penguin
```

An equivalent expression for the above is:

```
*role >endangered *symbol >African penguin
```

This encompasses how we define the data structure «text.» The Text Engine implements no other data structure.

Code: Text Transformation

The Text Engine is programmable—it can execute code for the purpose of generating, querying, or manipulating text. Its code consists of procedures that receive one text unit as an input and yield another text unit as an output. A procedure is called a «transformation» and can be perceived as text conversion; it converts one form (the input unit) into another (the output unit).

To express such code, the Text Engine supports a script language. For instance, to output a list of all penguin species contained in the text instance, one can use the script instruction:

```
select :penguin;
```

This will output specific symbols, such as «little penguin» and «African penguin.» To know which penguin species are threatened, one could write code to output only those:

```
select :threatened species.penguin;
```

The above code will output only units of type «penguin» with the parent unit type «threatened species.»

The select-transformation queries the text repository and returns any text units that meet the specified criteria. The criteria refer to the text structure; users can select units with a specific parent, type, role, or any combination thereof.

The «select» command can execute a transformation on every matching unit. For example, take the following code:

```
select :penguin { println ~common name };
```

This will execute the «println» (print line) command for each penguin species. The output would consist of two lines:

```
Little blue penguin
African penguin
```

One can have an alphabetically sorted list returned by using:

```
select :penguin (order by ~common name) {
  println ~common name;
};
```

To store this list as a raw string file, execute:

```
save "./penguins.txt" {
  select :penguin (order by ~common name){
    println ~common name;
  }
};
```

A transformation can be generated via instructions written in script language.

```
define =export penguin list :transformation {
  save "./penguins.txt" {
    select :penguin (order by ~common name) {
      println ~common name;
    }
  }
};
```

After executing the above command, the penguin list can be saved as a file with up-to-date content with the below:

```
export penguin list;
```

A transformation can have an object (in the grammatical sense of thing being acted upon):

```
define =export penguin list :transformation
  (object =file name :string)
{
  save [v ~file name] {
    select :penguin (order by ~common name) {
      println ~common name;
    }
  }
};
```

Now the file name is given upon every execution:

```
export penguin list "penguins (Aug 2015).txt";
```

Transformations can handle one or more arguments:

```
define =export penguin list :transformation
  (object =file name :string; argument =charset :string)
{
  save [v ~file name] (charset [v ~charset]) {
    select :penguin (order by ~common name) {
      println ~common name;
    }
  }
};
```

Arguments are passed through brackets when calling a transformation:

```
export penguin list "penguins (apple).txt" (charset #mac-roman);
```

Alternatively, the object can be given as an argument:

```
export penguin list (file name "penguins (apple).txt"; charset #mac-roman);
```

Calling the argument «object» instead of by name yields an equivalent result:

```
export penguin list (object "penguins (apple).txt"; charset #mac-roman);
```

Using the object as an argument is useful when it needs to be further qualified with some child units.

A transformation can have a body:

```
define =export penguin list :transformation
  (
    object =file name :string;
    body :transformation;
    argument =charset :string;
  )
{
  save [v ~file name] (charset [v ~charset]) {
    select :penguin (order by ~common name) { [v ~body] };
  }
};
```

Now the list can be exported with customized content on each call:

```
export penguin list "penguins (apple).txt" (charset #mac-roman) {
  println "Penguin species: [v ~common name]";
}
```

The above would save the following lines into the file:

```
Penguin species: African penguin
Penguin species: Little blue penguin
```

Object, body, and arguments can be given default values:

```
define =export penguin list :transformation
  (
    object =file name :string (default "penguins.txt");
    body :transformation (default { println ~common name });
    argument =charset :string (default #utf-8);
  )
{
  save [v ~file name] (charset [v ~charset]) {
    select :penguin (order by ~common name) { [v ~body] };
  }
};
```

This way, passing an argument is optional and assumes default values, if necessary. In order to output a list of common names as a UTF-8 file named «penguins.txt,» simply run:

```
export penguin list;
```

The Text Engine handles all transformations as text. For example, take the following script:

```
save "penguins.txt" {
  select :penguin (order by ~common name) { println ~common name };
};
```

This gets translated internally as the following text unit:

```
~transformation :save {
  ~file name "penguins.txt";
  ~content {
    ~transformation :select {
      ~target { ~type #penguin };
      ~order by { ~role #common name };
      ~action {
        ~transformation :println {
          ~segment :variable { ~role #common name };
        }
      }
    }
  }
};
```

Any transformation can be expressed in symbolic language. In general, everything in the Text Engine can be accessed through the symbolic language because everything is mapped into text units. This also applies, for instance, to the internal state of a program. To match unit names regardless of capitalization, execute the following script instruction:

```
set ~case sensitive names "false";
```

Or enter an expression in symbolic language:

```
#current process { ~settings { ~case sensitive names #false }};
```

Although the symbolic language does suffice, it is convenient to use the script language for transformations because the expressions it produces are more concise, easier to write, and intuitive to grasp. Symbolic language expressions, similar to assembly language code, can become too unwieldy for us humans to read or write effectively, and therefore, become a true challenge to manage.

The script language can actually be used to express some units that are not a transformation, as well. The first token is assumed to be the role (or possibly a subtype of it). For example, take the symbolic language expression:

```
~penguin {
  ~common name "African penguin";
  ~scientific name "Spheniscus demersus";
  ~breeding pairs "70,000";
};
```

This can be expressed in script language as such:

```
penguin {
  common name "African penguin";
  scientific name "Spheniscus demersus";
  breeding pairs "70,000";
};
```

The script language cannot replace the symbolic language for defining units, symbolic equivalence, or types.

Parsers

Two notations supported by the Text Engine have been introduced thus far: symbolic language and script language. There are more notations available, including user-defined ones. Users can mix notations into any string. To invoke an alternate parser for a substring, delimit the substring by containing it between two sets of square brackets indicating the alternate parser as follows:

```
~action [script/] select :threatened species [/script];
```

In the above example, the block parser scans the line and identifies three segments: before the block, inside the block, and after the block. It induces the corresponding parser to treat each segment in sequence. First, the symbolic language parser processes the string:

```
~action
```

This creates a unit with the «action» role. Next, the script language parser processes the following string:

```
select :threatened species
```

It then creates the «select» transformation. After that, the symbolic language parser processes the semicolon, which signals the end of the unit's definition. The context of the «select» transformation is the «action» unit because the substring occurs before the semicolon. Thus, the final parsed unit can be expressed as:

```
~action {
  ~transformation :select {
    ~target { ~type #threatened species };
  }
};
```

Quotation marks act as shortcuts for an alternate parser block. The following two lines have equivalent meaning:

```
~common name "African penguin";
~common name [quote/]African penguin[/quote];
```

The parser «quote» is a generic parser that invokes the implicit parser associated with the context's type. Above, the context is the role «common name,» and its type is «string.» Thus, the standard string parser is activated to load the substring «African penguin.» The longhand result is as follows:

```
~common name :string {
  ~character :letter #a { ~is upper case #true };
  ~character :letter #f;
  ~character :letter #r;
  ~character :letter #i;
  ~character :letter #c;
  ~character :letter #a;
  ~character :letter #n;
  ~character :punctuation mark #space;
  ~character :letter #p;
  ~character :letter #e;
  ~character :letter #n;
  ~character :letter #g;
  ~character :letter #u;
  ~character :letter #i;
  ~character :letter #n;
}
```

The type «string» has an implicit formatter that corresponds to the implicit parser. Consequently, the Text Engine will output this unit in symbolic language by default as:

```
~common name "African penguin";
```

The Text Engine can parse files, which must include an explicit parser block. For example, the entirety of content in file «species.te» may be:

```
*****
Scientific classification of species
(on the occasion of the Threatened Species Report)
*****
[symbol/]
~=species;
~=species {
  ~=common name :string;
  ~=scientific name :string;
};
~=mammal :species;
~=elephant :mammal;
[/symbol]
-----
Created by Mary Aug 2015
```

The lines before and after the symbolic language block are ignored.

With parser blocks, there is no longer a need to specify a file's format in order to be read. Procedures can parse arbitrarily formatted strings. Moreover, any expression can include segments of arbitrary notation, including user-defined ones. Besides providing flexibility, the block parser simplifies the implementation of specific parsers, which only need to deal with their own format because the block parser exclusively handles all parser interoperability issues.

The basics of the Text Engine have now been explored. The text repository can hold all kinds of data and code, and can be programmatically queried and modified through transformations that are included in the repository. Next, features that the program can deliver will be discussed, and then some possible applications will be considered.

Basic Features

Version 1.0 of the Text Engine will support the text model defined above and come with support for elementary data types, such as «integer» and «string,» as well as some basic coding block transformations (e.g. lists, loops, conditional statements), selectors, text and string generators, and file readers and writers. There will be parsers for both symbolic and script language, as well as support for explicit parser blocks. A shell will enable the user to interact with the system and launch batch processes in order to enter data or write and execute code.

Over time, enhanced functionalities will be added. Some planned additions include user-defined integrity constraints, triggers, automatic on-demand unit generation based on dependencies, updatable and cacheable mappings, extensibility through modules (written in the Text Engine's languages), multiple name spaces, data persistency (disc storage), remote access (web and other Internet services), and more. Enhanced parser support will allow users to customize their data entry format with ease. The resources used for programming may even evolve toward the distant but attractive goal of the Text Engine becoming a full-fledged programming system.

Applications

The Text Engine lays the foundation for building applications used for particular purposes.

Document Generation

An application that will be available from the outset will be document generation. Once information has been entered into a text repository, it can be queried to select parts of that information and output files built by using the «select» and «print» commands. For example, the following generates a list of penguin species as an HTML page:

```
print "<ul>";
select :penguin { print "<li>[xml [v ~common name]]</li>" };
print "</ul>";
```

Now, to output the information as a LaTeX document:

```
print "\begin{itemize}";
select :penguin { print "\item [latex [v ~common name]]" };
print "\end{itemize}";
```

And to output the information as an XML document:

```
print "<animals>";
select :penguin { print "<species>[xml [v ~common name]]</species>" };
print "</animals>";
```

In the examples above, the tag «v» found within square brackets evaluates the «common name» role of the respective species selected. The tag «xml» prepares the string containing the common name for embedding in XML (e.g. replacing reserved characters with their XML entity references). The tag «latex» prepares the string for embedding in a LaTeX source file.

The Text Engine offers a mapping mechanism that simplifies text generation. The type «mapping» is defined as:

```
~=mapping {
  ~=source type { ~=is sequence :boolean };
  ~=target type { ~=is sequence :boolean };
  ~=argument;
  ~=convert :transformation;
};
```

Mapping encapsulates the conversion between unit types. For example, the «mapping» type can be used to generate HTML strings representing species lists by entering the following symbolic language expression:

```
#mappings {
  ~mapping {
    ~source type #species { ~is sequence #true };
    ~target type #html;
    ~convert {
      [script/]
      print "<ul>";
      select #source.~item { print "<li>[xml [v ~common name]]</li>" };
      print "</ul>";
      [/script]
    }
  }
};
```

The same can be achieved using this script language command:

```
define :mapping
  (source type #species { is sequence #true }; target type #html)
  {
    print "<ul>";
    select #source.~item { print "<li>[xml [v ~common name]]</li>" };
    print "</ul>";
  }
};
```

In order to convert the penguin list into HTML, execute the following:

```
map :html { select :penguin (order by ~common name) };
```

The system determines what mapping to use according to the source and target types. In the above, it will map the «species» type to HTML, since «penguin» is a species.

This approach to document generation differs from common documentation systems in that the information can be collected with regard to a document's contents (e.g. species names, a list of threatened species) instead of the document's parts (e.g. chapters, paragraphs). Users are free to create and modify type hierarchies and are not bound to those that were hardcoded. Further, data can be queried semantically in order to generate content, both to restrict the represented information and to transform it (e.g., conditional filtering, sorting lists, include or omit details). The more cross-referenced the content and the more varying its structure over time, the more advantageous the Text Engine's document generation.

The Text Engine will include an additional parser for the so-called «prose format,» which is designed for entering large amounts of written text in prose by marking paragraphs with an unobtrusive notation. The beginning of this paper, for example, would look like this:

```
The intention of this paper is to introduce readers to [...].
I intend to develop and test this approach [...].
~h2 Fundamentals
~h3 Data: Text Structure
The Text Engine supports a symbolic language [...]:
~c "~=species;"
Subordinated symbols can then be added [...]:
```

Special marks identify the document's structure. As a result, a prose document will look the same as printed, with the exception of a few marks that indicate the document's different parts. In the example above, marks signal a section title (h2, h3) and a code sample (c).

Another useful extension for prose documents includes a reader for word processor files. With such an extension, the user can take advantage of a word processor program for writing a file that the Text Engine would then accept. Avoiding the usage of special marks is entirely possible because the reader can take the document's structure and formatting into consideration. It then feeds units with corresponding roles (e.g. a header line of level 2 as a unit with role h2).

Virtual Files

The next logical step will be to implement a virtual file system based on the Text Engine. For example, a file named «threatened.odt» may contain a report that uses the word processor «LibreOffice Writer.» This file could be included in a virtual directory so that it would no longer exist as a byte sequence on the user's hard disc, but instead generate on demand by the Text Engine.

To write a report concerning threatened species, for example, first extend the unit's definition to include the report's sentences:

```
#threatened species { ~=report paragraph :string };
```

The next step is to write a transformation that generates an open document file and maps from a sequence of species to a file:

```
define :mapping
  (
    source type #species { is sequence #true };
    target type #file;
    argument =title :string;
  )
{
  save :open document {
    output ~title [v ~title];
    output ~content {
      output ~heading1 [v ~title];
      select #source.:species (order by ~common name) {
        output ~heading2 [v ~common name];
        select ~report paragraph { output ~paragraph [v] };
      }
    }
  }
};
```

This presupposes the existence of the «open document» type that maps the semantics of this file format into text units and offers a «save» transformation that generates the open document file. The next step is to establish a mapping between the virtual file and a particular report:

```
map ~file "threatened.odt" (title "Species Report") {
  select :threatened species.species;
};
```

From now on, whenever the file «threatened.odt» is opened with the appropriate word processor, the above script executes and generates a text unit similar to the following:

```
~open document {
  ~title "Species Report";
  ~content {
    ~heading1 "Species Report";
    ~heading2 "African bush elephant";
    ~paragraph "Threatened by illegal hunting.";
    ~heading2 "African penguin";
    ~paragraph "Threatened by commercial fisheries.";
    ~paragraph "Eggs are considered a delicacy.";
    ~paragraph "Susceptible to pollution.";
  }
};
```

This text unit would be serialized according to the open document format and be delivered to the proper word processing program, which then opens the report, that looks like this:

In order for any changes made through the external application to be merged back into the Text Engine, an additional script is necessary. The mapping, which would map from the file to the sequence of species (i.e. the opposite of before) should be flagged as updatable:

```
map ~file "threatened.odt"
  (title "Species Report"; update #on save)
{
  select :threatened species.species;
};
```

With the above settings applied, it is simple to make changes to the report that would otherwise require a lot of manual operation. For example, it becomes possible to obtain another sort order, and more or less detailed contents, by adapting the conversion script. Additionally, both the external document and the Text Engine data are kept up-to-date without any manual intervention.

Another interesting aspect is that it is possible work on the same report in more than one document simultaneously.

```
#threatened species { ~=report paragraph :string; ~=summary :string };
```

For instance, after defining a summary paragraph, users can work on two different documents—the first containing only the summary for each species, and the second containing the summary plus all other paragraphs. Note that users can update the list of threatened species and edit the summaries in any document; the other document will automatically reflect any changes made the next time it is opened.

The ability to use more than one document containing the same data is not restricted to a single application. Users could keep a file in a virtual directory called «threats chronicle.ods» that works with a spreadsheet application and collects historical data about the evolution of populations and conservation statuses for every threatened species. Again, changes to the list in the spreadsheet would be immediately reflected on the word processor document, and vice versa.

If a document cannot be updated through the Text Engine, a read-only file will result. Further updates are intended to address issues of version control.

Repository

Initially, the Text Engine can keep its data as temporary data for as long as the program runs. On each execution, all required source files would be parsed again. At some point, the system must be equipped with persistent data options. That will allow the user to store the most current data and resume it upon a subsequent execution of the program. This improves not only system performance, but also makes it possible to store changes made at runtime, which is indispensable for some features. For example, data persistence would allow users to rename the «threatened species report» symbol as «threats report» with a single operation. No longer will users need to update each and every file in which this particular term occurs.

Moreover, data persistence will make a further application possible: a text repository. A text repository can be used as a replacement for a file system. The following defines a «directory» text unit:

```
~=directory { ~=subdirectory :directory; ~=file :binary };
```

Above, a directory is defined as containing files and, recursively, other directories. Users are able to store single files as byte sequences into the directory hierarchy. For instance, to store a presentation that is still in progress into the repository, enter:

```
~file =home.mary.panels."over-hunting.odp" "./panels/over-hunting.odp";
```

To refer back to it later on, use a common selector, such as:

```
select =home.mary.panels."over-hunting.odp";
```

An implicit parser for the type «file» allows the following expression, which is equivalent to the one above:

```
select :file "/home/mary/panels/over-hunting.odp";
```

Alternatively, to store this file on a USB thumb drive, run:

```
save "/mnt/usb/over-hunting.odp" {
  select :file "/home/mary/panels/over-hunting.odp";
};
```

To extend the file definition with some additional metadata, for instance, use:

```
~=directory {
  ~=label :string;
  ~=subdirectory :directory;
  ~=file {
    ~=label :string;
    ~=creation time :date;
    ~=content :binary;
  }
};
```

Get a full directory list with the following:

```
select =home.mary.panels.~file { print ~label };
```

A shortcut term, «ls,» could be set for the expression above in order to list files in a way similar to a Unix shell:

```
ls "/home/mary/panels";
```

To get a list of files sorted by creation time, use the following:

```
select =home.mary.panels.~file (order by ~creation time) {
  print ~label;
};
```

Remarkably, a file system created through these means would be extensible right out of the box. For example, to add some metadata to a file:

```
~=work file :file { ~=completion :percent };
```

This way, each file has a percent number added to it, which indicates its completion degree. A query can then be run, such as:

```
select :work file (having ~completion (less than "50%"));
```

The above retrieves all files for which less than half of the work has been done. Note that a user could easily add metadata at any time both manually and by executing transformations, even when some files already exist.

In a similar manner, a user could easily add tags to files:

```
~=tagable { ~=tag :string };
#file :tagable;
#"over-hunting.odp" { ~tag "difficult" };
```

A user could also group files in predefined categories:

```
~=categorizable {
  ~=category;
  ~=job :category;
  ~=hobby :category;
};
#file :categorizable;
#"over-hunting.odp" { ~category #job };
```

An existing file could be included in additional directories without necessitating the creation of a copy by using:

```
~file #"panels.over-hunting.odp" ="drafts.over-hunting.odp";
```

The above adds an entry into the virtual directory «drafts» that shares metadata and content information with the file of the same name in the virtual directory «panels.»

If the above virtual file support is implemented, directories can include both the real and virtual files; that is, any file's contents can be either static or dynamic.

The Text Engine will eventually support customizable user-defined constraints to ensure data consistency, which is especially important for persisted data and repositories. A constraint in the Text Engine is a transformation that is automatically executed each time a particular type is instantiated, stops program execution, and rolls back the current transaction if the required conditions are not met.

For example, a user can require that each species has a common name by running:

```
define :constraint (for :species) {
  if undefined ~common name {
    throw :missing argument (for ~common name);
  }
};
```

With the above constraint, if a user creates a new species but does not enter a common name for it, the execution will abort and throw an error message for that argument stating that the common name is missing.

Archive

A Text Engine repository can be used for long-term archiving purposes. Users will not merely store their files as byte sequences, but will also register its semantics. To recall the virtual file example above, a user would store the unit «threats report» as an instance of the «open document» type along with the file «threatened.odt» in binary form.

A long-term archive should hold type definitions for every file format. For example, to store .mp3 files, one would map the file format into text units. An .mp3 file consists of frames, and each frame has a header and audio information. The header is built of segments, such as the frame sync (the first 11 bytes), the MPEG audio version ID (the next 2 bytes), and some more. Thus, the .mp3 type can be defined as follows:

```
~=mp3 :file {
  ~=frames {
    ~=frame {
      ~=header {
        ~=segment { ~=length :integer };
        ~=frame sync :segment { ~length "11" };
        ~=mpeg audio version id :segment { ~length "2" };
        ~=layer description :segment { ~length "2" };
        ~=protection bit :segment { ~length "1" };
        ~=bit rate index :segment { ~length "14" };
        ~=sampling rate :segment { ~length "2" };
        ~=padding bit :segment { ~length "1" };
        ~=private bit :segment { ~length "1" };
        ~=channel mode :segment { ~length "2" };
      };
      ~=audio { [...] };
    }
  }
};
```

This file format will be declared by a module that is programmed once and then distributed with the Text Engine, or by a third party. The mp3 module exposes the type definition along with a «save» transformation that serializes an instance of this type as a binary file, and a «read» transformation that deserializes a file into an instance of this type. To store an .mp3 file into the repository, the user loads the mp3 module and then executes:

```
~file :mp3 =home.mary.interviews.manager "./manager.mp3";
```

Given that the file to be read has the defined type «mp3,» the Text Engine invokes the reader included in the mp3 module in order to read the file.

Proceeding in this way, the archive contains each file not only as a byte sequence, but also semantically parsed. The fact that each file is parsed before being stored ensures that its format is known and correct. Small format variations and non-conformance with standards can now be detected, and with a custom derived type, documented.

The question then arises: what will be persisted in the repository? The file as a byte sequence, the unit definitions, or both? This is actually up to the module's programmer. The mp3 module could store the units when parsing the file and generate the byte sequence on demand, or it could store both of them in the repository. It could even store only the byte sequence and generate the units on demand. The best choice, however, would be to allow the user decide for each instance:

```
~file :mp3 =home.mary.interviews.manager "./manager.mp3" {
  ~binary storage: real;
  ~unit storage: virtual;
};
```

This way, the system's performance can be tweaked. If the module also supports subsequent changes to its parameters, then the user can change the preferred storage mechanism at any time according to their system's respective space and speed requirements. The module's behavior can be changed by entering the following:

```
#home.mary.interviews.manager {
  ~binary storage: virtual;
  ~unit storage :real;
};
```

This operation can be performed unassisted by a procedure that monitors the system's usage.

A long-term archive must deal with format conversions. For example, a user may want to read a document produced with an old word processing program no longer supported by current operating systems. The whole document, or parts of it, can be converted into another document format with a script. For example, to export the open document reporting on threatened species as a raw string file:

```
save "threats.txt" {
  println "*** [v =threats report.~title] ***";
  select =threats report.content (level #children) {
    if :heading1 { println "** [v] **" };
    if :heading2 { println "* [v] *" };
    if :paragraph { println "[v]" };
  }
};
```

Note that format conversion in such a system is very straightforward. Users handle conversion with the format's logic and not with file formats by utilizing general-purpose language to access unit definitions and avoid handling low-level binary sequences. Since unit definitions are contained in the repository and can be queried, and unit names (e.g. «heading» and «paragraph») are self-explanatory, the average user does not need to consult any further documentation. Any user with entry-level programming skills can therefore write conversion scripts spontaneously when required.

Personal Organizer

Another possible application of the Text Engine is as a personal organization utility that can manage an individual's documents, addresses, calendar dates, task lists, and other items. Begin with a simple to-do list, as follows:

```
~=to-do list {
  ~=task {
    ~=description :string;
    ~=due :date;
    ~=done :boolean;
  }
};
~to-do list =conference work {
  ~task { ~description "book airplane ticket"; ~due "09/14/2015" };
  ~task { ~description "book accommodation"; ~due "09/28/2015" };
  ~task { ~description "travel"; ~due "10/14/2015" };
};
```

To link a «prepare presentation» task to a report, write:

```
~=attachment container { ~=attachment :file };
#task :attachment container;
#conference work {
  ~task {
    ~description "prepare presentation";
    ~due "09/25/2015";
    ~attachment #"home.mary.panels.over-hunting.odp";
  }
};
```

To get a list of tasks due for the next conference that have yet to be completed, execute:

```
select #conference work.~task
  (having ~done (equal #false); order by ~due)
{
  println ~description;
};
```

To get all open tasks in general, instead use the following:

```
select :task
  (having ~done (equal #false); order by ~due)
{
  println ~description;
};
```

Suppose an email facility and a job scheduler agent are available. Users can make arrangements to receive daily emails with a list of open tasks by running:

```
schedule #daily {
  email "mary@example.com" (subject "open tasks") {
    select :task (having ~done (equal #false); order by ~due) {
      println ~description;
    }
  }
};
```

To manage a user's contacts, enter the following:

```
~=contact {
  ~first name :string;
  ~last name :string;
  ~email :string;
};
```

To look up an individual's email information, enter:

```
select ~contact
  (having ~last name (like "Brewster"))
{
  println "[v ~last name], [v ~first name]: [v ~email]";
};
```

A shortcut can be set for the above expression as follows:

```
define =look up :transformation
  (object =target name :string)
{
  select ~contact
    (having ~last name (like [v ~target name]))
  {
     println "[v ~last name], [v ~first name]: [v ~email]";
  }
};
```

Then, execute:

```
look up "Brewster";
```

This would print a result similar to the below:

```
Brewster, Emily: emily.brewster@example.com
Brewster, Mike: mbrewster@example.com
```

As with tasks and contacts, users can define collections of other entities, such as documents or calendar dates. This involves defining the data structure as a type once and then instantiating it for each item. The user can freely add fields to every entity and derive subtypes of the data, even if an external module defined it. In addition, these fields are not restricted to mere character strings, but can have arbitrary types, including those defined by the user.

Thus far, it has been discussed how applications would manage their data. Nothing about the user interface has been assumed. Although the Text Engine shell is a user interface that gives access to all data and operations, each application can supply additional views and interactive surfaces. For example, a Text Engine-based organizer application could present daily, weekly, and monthly calendar views; email messages; and to-do lists in a visually rich manner. Next, graphical user interfaces based on the Text Engine will be discussed.

Workbench

The Text Engine's workbench will be a graphical user interface designed for working with the Text Engine and its applications on a workstation or desktop computer. It is targeted at power users.

Windows

The workbench displays a collection of windows. There is no generic container for these windows and no dependency with any particular window manager. The first time the workbench launches, a single window appears, in which a Text Engine shell runs. The user can enter script instructions for the Text Engine to execute. To work on a particular word processor document, open it as follows:

```
open #over-hunting report;
```

This causes the workbench to create a new window, launch the word processor application in it, and call the application to load the document (which is a repository unit and not an operating system file). This assumes a word processor application designed to be compatible with the Text Engine and the workbench. The application fetches the document and renders it inside the window. When the user edits something, the application performs the required changes to the repository. This updating process is not a trivial issue, however; it can involve explicit or implicit transactions, moving data back and forth to temporary workspaces, and more.

In a similar regard, and assuming a spreadsheet application designed for the workbench, users can open a window showing the spreadsheet chronicling threatened species by executing:

```
open #threats chronicle;
```

So-called «views» control the look and feel of documents in the workbench. A view is a unit in the repository that defines the appearance and user interaction capabilities of a particular type. Whenever the workbench opens an item, it determines the view to the unit's type and then opens it. The user can open not only documents, but also arbitrary units. For instance, opening a transformation results in a window showing its source code:

```
open #export penguin list;
```

The above would open a new shell window containing a script similar to the following:

```
define =export penguin list :transformation
  (object =file name :string)
{
  save [v ~file name] {
    select :penguin (order by ~common name) {
      println ~common name;
    }
  }
};
```

If there happens to be no other view available for the given type, the workbench opens a window that shows an expression in symbolic language.

```
open #little penguin;
```

The above operation would result in the following:

```
=little penguin ~penguin {
  ~common name "Little blue penguin";
  ~scientific name "Eudyptula minor";
  ~breeding pairs "500,000";
};
```

If there are several views available, the user can explicitly choose one:

```
open :turbo spreadsheet "#threats chronicle";
```

The above runs the spreadsheet application under the «turbo spreadsheet» view. Users can also set specific view parameters as follows:

```
=new view ~turbo spreadsheet "#threats chronicle" {
  mode #power user;
  caption "Yearly report";
};
open #new view;
```

The same can be accomplished with a single instruction:

```
open (object :turbo spreadsheet "#threats chronicle" {
  mode #power user;
  caption "Yearly report";
});
```

Any unit can be shown in symbolic language by invoking the symbol view:

```
open :symbol view "#threats chronicle";
```

To create a document, add a unit with the desired type. For example, a report is created by entering this in the shell window:

```
put ~=new report :report document;
open #new report;
```

Each window has a source that delivers data to it. The source is a transformation. The source for the window above would be:

```
select #new report;
```

The workbench shows the source of every window in a frame on the top or the bottom of the window. The source frame contains a script expression that the user can edit and rerun, thus changing the window's contents. For example, opening the spreadsheet «threats chronicle» returns the following source frame:

```
select #threats chronicle;
```

Users can then switch to the turbo spreadsheet view of the same document by changing the source as follows:

```
select :turbo spreadsheet "#threats chronicle";
```

Any transformation can act as source, including those involving on-the-fly unit generation:

```
select :threatened species.species (order by ~common name) {
  println "[v ~common name] ([v ~scientific name])";
};
```

A window with the above source would show:

```
African bush elephant (Loxodonta Africana)
African penguin (Spheniscus demersus)
```

The source of a window can be bound to an item focused at another window. For instance, additional information can be acquired about the single species currently the focus of the spreadsheet «threats chronicle» in another window by setting its source to the following:

```
select #threats chronicle window.current item :species {
  println "[v ~common name] ([v ~scientific name])";
};
```

The workbench can implement a dynamic update mechanism in order for the contents of a window to be automatically updated whenever the focused item in the spreadsheet changes. This requires the Text Engine to support triggers (i.e. automatic execution of custom code) whenever any instance of a particular type is updated.

Views

This section will go into detail regarding views. The window's content area is comprised of rectangular, non-overlapping areas called «frames.» Every frame is bound to an instance of the type «view,» which is a type exposed by the workbench that provides input/output mechanisms. Thus, it is possible for transformations to display something on the bound frame, as well as react to the user's keystrokes and mouse actions. Applications offer views for its visible types. When a view is instantiated, the workbench automatically launches the appropriate application, unless an instance is already running.

There are some comprehensive views, such as a word processor or a spreadsheet, that take up most of the space in a window while offering plenty of functionality. Yet there are also some very specific and much more limited views available. For instance, the «integer view» supports the standard user interface for the field type «integer» by providing transformations for displaying and editing integers on a given frame.

The workbench will furnish views for elementary data types (e.g. integer, strings) and some general container views such as list, table, and tree views. A table shows items in columns and rows, and a tree shows a hierarchy of items. Both will have the usual functionalities (e.g. resizing and moving table columns, collapsing or expanding tree nodes). Before considering the list view in more detail, the string view must be clarified.

The raw string view is the standard view for the data type «string.» The definition can be written as follows:

```
~=string view :view {
  ~=style :style;
  ~=read only :boolean;
};
```

In the shell window, users can open a window with a string view that shows a string, like:

```
open "Hello, world!";
```

To set the string in italics, use the following:

```
open (object "Hello, world!" { style #italics });
```

In order to facilitate string representations of arbitrary types, the workbench offers an intermediate view:

```
~=stringifyable {
  ~=parse :transformation;
  ~=format :transformation;
  ~=style :style;
  ~=read only :boolean;
};
~=stringifyable view :string view;
```

The «stringifyable» view calls the format transformation, gets a string instance representing the value, and passes both the string instance and the style information to the string view, which then renders the string. After a user-made change, the «stringifyable» view calls the parse transformation with the new string, returning the new value.

This view eases the implementation of views for specific data types. For instance, the integer view is implemented as «stringifyable.» This requires to program only a parser and a formatter that convert between the binary value and its decimal representation with numeric characters.

Any unit can implement the «stringifyable» interface. Take the following example:

```
#species :stringifyable {
  ~format {
    [script/] println "[v ~common name] ([v ~scientific name])" [/script]
  };
  ~read only #true;
};
```

Next, perform the following:

```
open #little penguin;
```

Rather than having the symbolic language expression returned, the result is instead:

```
Little blue penguin (Eudyptula minor)
```

It is worth mentioning that any unit whose description in symbolic language is large or complex should implement the «stringifyable» interface, or at least the formatter transformation, so that the user can inspect their instances with minimal effort.

As an example of a general container view, take the list view into account.

```
~=list { ~=item }; ~=list view :view;
```

The list view shows a stack of items. Execute the following:

```
open (object :list { item :string "one"; item :integer "12" });
```

A window with two lines will appear:

```
one
12
```

The representation of each item is handled not by the list view, but by the workbench. The workbench chooses which view is most appropriate. In the above example, the first line is generated by the raw string view and the second line by the integer view.

No restriction exists regarding item types that a list can hold. For example, if several applications include views for word processing, images, and outlines, the user can create a new window and combine instances of these types within a single list.

To use a list view to represent arbitrary types, a mapping between the given type and a list item must be established. For example, in order to open a window with a list of species names, a user must first define a mapping from species to list.

```
define :mapping
  (source type #species (is sequence #true); target type #list)
{
  output ~list {
    select #source.~item { output ~item :string [v ~common name] };
  }
};
```

Then, a list of selected species displays by executing:

```
map =species list ~list {
  select :threatened species.species (order by ~common name);
};
open #species list;
```

The above generates and opens this unit in a new window:

```
=species list ~list {
  ~item :string "African bush elephant";
  ~item :string "African penguin";
};
```

The window shows two lines, each containing a common name.

In order for the view to be updateable, the mapping should also be updateable, achieved through the following:

```
map =species list ~list (update #on save) {
  select :threatened species.species (order by ~common name);
};
open #species list;
```

This requires a reciprocal mapping to exist in the repository from type «list» into a sequence of type «species.»

Another available view is the prose view. It is an editor for prose writing and based on a type as follows:

```
~=prose writing {
  ~=division {
    ~=heading :string;
    ~=paragraph :string;
    ~=division :division;
  }
};
```

For example, the beginning of this paper is represented as:

```
~prose writing {
  ~=code paragraph :paragraph;
  ~division {
    ~paragraph "The intention of this paper is to [...].";
    ~paragraph "I intend to develop and test this approach [...].";
    ~division {
      ~heading "Fundamentals";
      ~division {
        ~heading "Data: Text Structure";
        ~paragraph "The Text Engine supports a symbolic language [...]:";
        ~code paragraph "~=species;";
        ~paragraph "Subordinated symbols can then be added [...]:";
      }
    }
  }
};
```

The prose view derives from the list view and displays each division's contents, visiting each division in preorder. Headers and paragraphs are not rendered by the prose view, but instead by the view attached to their respective type. The type, in this case, is the raw string view, since both are defined as strings. In order to give a distinct appearance to headings and code paragraphs, run the following:

```
#code paragraph :string view { ~style #monospace };
#heading :string view { ~style #header1 };
```

It may prove useful to divide paragraphs and headings into segments in order for them to include structured content, such as links, automatically generated parts, or special formatting for selected words. To redefine these types as segment containers, apply the following:

```
~=prose writing {
  ~=division {
    ~=segmented string {
      ~=segment {
        ~=content :string;
        ~=view :string view;
      }
    };
    ~=heading :segmented string;
    ~=paragraph :segmented string;
    ~=division :division;
  }
};
~=segmented string view :view;
```

The segmented string view will now render each segment in order of appearance, each of which with its own string view instance. Different parts of a paragraph can be formatted, as well:

```
~paragraph {
  ~segment { ~content "It is important for" };
  ~segment { ~content "every"; ~view { ~style #emphasized }}};
  ~segment { ~content "species to be handled appropriately." };
};
```

The above results in a paragraph showing the word «every» in a bold font face.

After having discussed how views are implemented, the next step is to see how the user will operate with them.

User Interaction

Users control applications through the user interface provided by views. For example, a word processing view may have a menu bar with an option labeled «Insert Date» that inputs the present date at the cursor's current position. When the user applies this option, the view lets the Text Engine execute an instruction, such as:

```
select #selected paragraph { insert { now }};
```

This instruction injects the value returned by the transformation «now» into the current paragraph. A similar instruction could have been run by the user in the shell window instead and it would have had the same effect:

```
select #window1.selected paragraph { insert { now }};
```

Not only the view's data and operations, but also its user interface is mapped into repository units. For example, the aforementioned menu option is defined as:

```
=main menu ~menu bar {
  ~option {
    ~caption "Insert date";
    ~action [script/] select #selected paragraph { insert { now }} [/script];
  }
};
```

The user can list the available menu options in the shell window with the following:

```
select #window1.main menu.~option { println ~caption };
```

The user can also add a new option to the menu, binding it to an arbitrary instruction:

```
put #window1.main menu {
  option "To raw string file" {
    save "[v ~document name].txt" {
      select :paragraph (parent level #any) { println [v] };
    }
  }
};
```

After entering the above code in the shell window, the application window shows a new menu option labeled «To raw string file» that exports the document without headings, one paragraph at each line. The generated file has the same base name as the current document and the file extension «txt.»

The workbench itself is accessible through the shell window as well. For instance, the user can obtain a list of all open windows by executing the following:

```
select #workbench.layout.~window { println ~caption };
```

To induce the workbench to launch a particular document at start time, run the following:

```
put #workbench.start { open #threats report };
```

As a basic principle, everything that can be done in the workbench through the user interface—no matter which application it comes from—can also be done through scripting.

Conclusion

To recapitulate, it is now known what resources the Text Engine will seek when representing data and what coding procedures operate with it, as well as how applications will be programmed and used. Some example applications have been sketched in order to illustrate the system and to provide further evidence that it is both feasible and useful. In order of appearance, the examples became increasingly complex, and their design less clear; the undertaking progressively unsettled. While document generation is (due to a predecessor project) a well-known application to a degree, the workbench is still a personal dream. I intend to develop the Text Engine function by function in order to focus on each application in sequence, and then deploy and mature it, before moving on to the next.

What fascinates me is the experience of developing and the perspective of using such a well-integrated system. Users can enrich and detail all of an application's data and operations to adapt to their individual needs. Applications share data and operations, as well as their look and feel, despite the system's distributed architecture. Users can create interfaces between independently developed applications by simply writing sentences in familiar, generic language, and without having to deal with file formats. The same applies to user interfaces, both linguistic and graphical—they easily adjust to specific requirements and styles, and this occurs in an application-independent manner. Besides being flexible, the system is very consistent and achieves a plethora of goals by combining just a few basic features. That is because I have built the Text Engine on a single principle—software is text.

I believe that everything points to the Text Engine becoming an extremely powerful system.

—-oOo—-