Text-Oriented Software

Text-Oriented Software

Toward a Unifying Software Principle

Francesc Hervada-Sala

Cover photo © 2008 Bora Uçak, reproduced with permission.

Copyright © 2010 Francesc Hervada-Sala. All rights reserved.

Published by u-tx.net, Frankfurt am Main, Germany.

1st edition, March 2010

ISBN 978-1-4505-3873-2

Contents

Preface

Software technology has progressed a lot in the last fifty years. In the 1960s the development of time-sharing systems emerged to bring computer power and networking to the people, research flourished in the 1970s at the Xerox Palo Alto Research Center, where the ground elements that define computing today were set up. In the last decades there have been many advances toward humanizing computing, making it accessible and intuitive. This is good and must be further pursued. But one important aspect has not been cultivated: making software more powerful for the intellectual work. The developments from Doug Engelbart toward more intelligent computer systems have not yet caught on, the ideas of Ted Nelson about an electronic literature and his criticism about the current software landscape have not yet been understood. It is about time to work on getting more intelligence from computers. That is what we are trying here.

This book presents a new principle for understanding computing. I am convinced that the idea presented here is right and opens up a promising path, but the theory as formulated here is perhaps still defective. This idea is extremely simple but also extremely hard to communicate. The multiple details that are treated here should lead in the reader's mind to a single point of view that underlies it all. This book is not intended to be read sequentially from the first page to the last, you will probably want to jump from one part to another to get answers to your own questions. You will find the materials in a rather logical order. The first section, «Text,» presents a sketch of a text theory based on a general algebraic text formula. The section «Imagine» visualizes what kind of software could be built upon that theory. After that there are some case studies, including the description of an already existing implementation of the theory, the experimental software «Universaltext Interpreter.» The last section, «Background,» contains several considerations that might be useful as introductory notes.

The content of this book can be summarized with a single sentence: Computers are text machines. This does not mean that we can use computers for text among other purposes. It means that text is all computers are about, the only material that they store and manipulate. This book proposes a fundamental concept of text that reveals that documents, media, relational databases and source code are nothing but particular kinds of texts. This concept of text is not only a principle that can lead to a deeper understanding of computing, but it can also be directly implemented and produce computing systems that outdo the current ones.

Francesc Hervada-Sala

Frankfurt, January 27th, 2010

Part I

Text

1. Text Structure

I present here a candidate for the fundamental text structure. Let us introduce first a language to represent text and after that deal with the general text definition.

Plain Text Language

The simplest text is a single text unit having a name. We note it down this way:

```
^ person
```

This defines a text unit named «person.» A text unit can consist of some other text units. For example, a person has a name and a birth date.

```
^ person {
    ^ first-name
    ^ last-name
    ^ birth-date
}
```

That introduces three named text units that are part of the text unit «person.»

A text can also be of some type. We use for this a colon `:` as prefix.

```
= Ann : person
```

The prefix = identifies the unit name. If Ann is a person, then she has all characteristics of persons, according to the previous definition she has a name and a birth date:

```
= Ann : person {
    ~ first-name Ann
    ~ last-name Jones
    ~ birth-date 6/24/1954
}
```

The above introduces a text unit named «Ann» and records some data about it. The three subordinated text units are not being here defined, but mentioned, that is why we do not use ^ (for definition) but ~ (for role) instead.

A text ends with some data not being further analyzed, above two strings «Ann» and «Jones» and a date «6/24/1954.» For these we can define some unit types, too. The complete text would look like this:

```
^ string
^ date
^ person {
    ^ first-name : string
    ^ last-name : string
    ^ birth-date : date
}
= Ann ~ person {
    ~ first-name Ann
    ~ last-name Jones
    ~ birth-date 6/24/1954
}
```

If the text we are analyzing talks about some more people, we might want to distinguish between men and women. We can add the names «man» and «woman»:

```
^ man : person
^ woman : person
```

After that, each unit with the type «woman» or «man» is necessarily a person, too, and has therefore a name and a birth date:

```
= Ann ~ woman {
    ~ first-name Ann
    ~ last-name Jones
    ~ birth-date 6/24/1954
}
```

Text units can be grouped to produce higher text units. For example, a family is a compound unit consisting of persons.

```
^ family {
    ^ parent : person
    ^ child : person
}
```

The persons of a family are grouped above in two classes: the parents and the children. Each person that belongs to a family plays a particular role in it, either as parent or as child. When talking about some family and recording its members, one must determine the role of each of them.

```
= Jones ~ family {
    = Ann ~ parent : woman
    = John ~ parent : man
    = Lena ~ child : woman
}
```

Note the difference between a type and a role. Ann has herself a type, she is a woman, apart from that she belongs to family Jones and plays inside this family the role of a parent. Text units with different types (man or woman) can play the same role (parent or child).

Instead of a defining a unit, one can refer to a unit defined elsewhere with the sign ==. For example the following text produces exactly the same family as above:

```
=Ann : woman
=John : man
=Lena : woman
```

```
= Jones ~ family {
    ~parent ==Ann
    ~parent ==John
    ~child ==Lena
}
```

With the plain text language we have introduced a particular model of text. We record any text as a web of text units that relate to each other as component, as type or as role. Let us reduce this now to a single algebraic expression as a general definition of text.

Text Formula

The structure of every single text can be reduced to some particular text units that are related to each other this way: each text unit has a parent, a role and a type, these being again text units. One can express the general text formula this way:

```
<parent> {
    <child> ~<role> :<type>
}
```

The meanings of these relationships are these:

◇ A parent *consists of* some children. Referring to the parent is the same as referring to the ordered list of its children.

◇ A unit *plays* some role as a child of its parent. The role is a second grade relationship between children of a unit and children of this unit's type.

◇ A unit *instantiates* some type. That means the unit inherits its character from the type, from which it is seen as a particular occurrence. If the type has some attributes or functionality, the unit has these, too. A unit must have the same type as its role or a descendant type of it.

This one formula exhausts the structure of every text. An arbitrary text can be reduced to an expression based on it, getting its structure completely described.

Note that each finite text must have at least one completely self-related unit that is its own parent, type and role. Otherwise the above formula would not be true for all text units.

Note that there is a single root category, the unit. A type is not a separate entity: types are themselves units, too. Every unit can act as the type of another unit.

Note that the formal text definition does not include names. Unit names can be part of a language one uses to express texts, but they do not belong to the text structure itself.

Scope of Text

Let us now see what range of matters this concept of text applies to.

Books

Books are —not surprisingly— text. A particular book might be arranged in several fixed levels such as sections and chapters, this can be expressed explicitly «Section 1,» «Chapter Two» or implicitly through graphic presentation, for example: section titles appear at a separate page on the right side of the book, that only contains the title and is otherwise left blank, whereas chapter titles begin a new page, indifferently on the left or the right side and contain some prose paragraphs, too.

```
^ myBookOutline {
    ^ section {
        ^ title : string
        ^ chapter {
            ^ title : string
            ^ contents {
                [...]
            }
        }
    }
}
= myBook ~ myBookOutline {
    ~ section Introduction
        ~ chapter Motivation
        [...]
    ~ section Theory
        [...]
    ~ section Applications
        [...]
}
```

A book can also be arranged by generic levels instead, as usually for example in science as in «1.1 Motivation.»

```
^ myThesisOutline {
    ^ level {
        ^ title : string
        ^ no : cardinal
        ^ level : level
        ^ contents {
            [...]
        }
    }
}
= myThesis ~ myThesisOutline {
    ~ level Introduction {
        ~ no 1
        ~ level Motivation {
            ~ no 1
            [...]
        }
    }
    ~ level Theory [...]
}
```

Prose

Prose consists, apart from an arrangement in sections and chapters as we have already seen, of prose blocks, being a prose block a continuous flow of natural language. It can be a list of paragraphs:

```
^ prose {
    ^ paragraph : string
}
=chapter1 ~ prose
    ~ paragraph At first one must say...
    ~ paragraph Nonetheless this is tricky...
```

But it can also contain other structure elements, such as a list:

```
^ prose {
    ^ paragraph : string
    ^ list {
        ^ item : string
    }
}
=chapter3 ~ prose {
    ~ paragraph One must perform some steps:
    ~ list {
        ~ item First...
        ~ item After that, ...
        [...]
    }
}
```

The prose flow is not always a single level, there can be more than one, for example something included in parentheses or between dashes. Footnotes can be an embedded level, too.

```
^ paragraph {
    ^ sentence {
        ^ content : string
        ^ note : string
    }
}
~ paragraph {
    ~ sentence Something must be done. {
        ~ note Not everything is good, though.
    }
}
```

Natural Language

The prose structure ends with natural language sentences. But a sentence is not an unanalyzable character string, it has structure, too. One can perform a syntax analysis of a sentence to free its syntactic structure.

```
~ sentence : copulative-sentence "I am tired."
{
    ~ subject : personal-pronoun {
        ~ person 1
        ~ number singular
    }
    ~ copula "to be" {
        ~ tense present
        ~ mode indicative
    }
    ~ attribute tired
}
```

One will surely not want to make manually a syntax analysis of all the sentences that one writes, but in humanities one will want to analyze completely say Plato's Works, and a spell checker program will try to make an approximate one in order to check the grammar.

Apart from that, natural language sentences have a meaning, and this can be analyzed, too. For example, if a sentence quotes a work, the reference can be recorded.

```
^ sentence {
    ~ reference {
        ~ work : author.work
    }
}
```

One can set references on single words, too, for example to disambiguate them at a dictionary. One can set references to contemporary people or events that clarify text passages.

Knowledge

Some academic studies are mainly done in prose. Some of them have particular writings as object (as in humanities) or as mean (as in history), these writings can be recorded as prose and everything that the scholars find out can be tied with it together.

All knowledge can be expressed as text.

```
^ kingdom {
    ^ name : string
    ^ house {
        ^ name : string
        ^ head-of-state {
            ^ name : string
        }
        ^ king : head-of-state
        ^ queen : head-of-state
    }
}
~ kingdom "United Kingdom" {
    ~ house "House of Stuart"
        ~ queen "Anne"
    ~ house "House of Hanover"
        ~ king "Georg I"
        ~ king "Georg II"
        [...]
    ~ house "House of Windsor"
        [...]
}
```

Mathematics

Some sciences use the mathematical language. Mathematical statements can be analyzed syntactically and be reduced to a text, too.

```
~ statement : equation "3+4=7" {
    ~ part {
        ~ addition {
            ~ operand : integer 3
            ~ operand : integer 4
        }
    }
    ~ part {
        ~ value : integer 7
    }
}
```

Programming

Like natural language and mathematical language, every formal language can be recorded as text, too. One gets the text of a formal expression through syntax analysis, which also frees its semantics. This includes programming languages.

```
~ statement : for-loop {
    ~ source-string ""
for(int i=0; i<10; i++)
    dothis(i);
""
    ~ preoperation : assignment {
        ~ variable {
            ~ name i
            ~ type integer
            ~ scope block
        }
        ~ value {
            ~ constant : integer 0
        }
    }
    ~ condition : lesser-than {
        ~ left-operand : variable i
        ~ right-operand : constant 10
    }
    ~ postoperation : increment {
        ~ variable i
    }
    ~ statement : statement-block {
        ~ statement : function-call {
            ~ function dothis
            ~ parameter : variable i
            }
        }
    }
}
```

One can also express in terms of text a «make file» as commonly used to specify procedures for building executable files from source files.

```
^ make {
    ^ target {
        ^ filename : string
        ^ dependency : filename
        ^ command : string
    }
}
~ make {
    ~ target helloworld {
        ~ dependency helloworld.o
            ~ command cc -o $@ $<
     }
    ~ target helloworld.o {
        ~ dependency helloworld.c
            ~ command cc -c -o $@ $<
    }
}
```

Digital Media

Digital media files can be expressed as text. For example mp3:

```
^ mp3-file {
    ^ header {
        ^ segment {
            ^ bit-count : cardinal
            ^ meaning : string
        }
        ~ segment {
            ~ bit-count 12
            ~ meaning Sync Word
        }
        ~ segment {
            ~ bit-count 1
            ~ meaning Version
        }
        ~ segment {
            ~ bit-count 2
            ~ meaning Layer
        }
        [...]
    }
    ^ data {
        [...]
    }
}
```

Every media file can be expressed this way, one only needs to reproduce the so-called «file structure.»

We have seen that many different things can be reduced to this concept of text. Not only such different things as prose and digital media, mathematics and programming languages can be seen as text, but they can all be expressed by a single structure. This opens the door for computer assisted text management in all these fields.

2. Comparing Text to Other Structures

We defined the text as a four-way association between units:

```
<Parent> { <Child> ~ <Role> : <Type> }
```

with the restrictions: the role unit must be a child of the parent's type unit (or of some ancestor of it) and the type must be consistent with the role's type.

Let us now compare this structure with some software structures that are common today.

Relations

A relation is a two-way association between elements of two sets, `(a,b)`, being `a` element of A, `b` element of B, the relationship is a subset of the Cartesian product AxB. Operations are defined between relations: union, intersection, etc.

A text can be seen as a 2nd-order relationship between units:

```
((p,c),(t,r))
```

It is not a 4-way relationship, but a double parent-child relationship, between units and their types.

Transformations are defined between text units.

Instead of first defining a relation and then reducing text to it, one can first define text and then reduce a relation to it.

A relation is a trivial case of text.

```
P { C ~unit :unit}
```

Being *unit* the root unit that is defined as a 4-way self-reference:

```
unit { unit ~unit :unit}
```

The later approach, that defines text first and derives relations from it, is clearly the better one. A single root definition of *text unit* suffices, and it is not an abstract concept such as *set* but a real thing, because text is something that humans can patently experience.

Text is likely to provide a basis for a foundation of mathematics, because it is the only root concept that describes each phenomenon together with the possibility for humans to talk about it. If you begin with «set,» you must establish a correspondence between a set and the word «set»; you must explain what sets are, and what the word «set» is, and why they match. The same if you begin with another root concept such as «number» or «operation.» Only with «text» you don't have this problem, you just need to assume that humans can talk about things. And if you do not admit that humans can talk about things, then you are surely not going to do mathematics at all.

Data Structures

Variables can obviously be represented as text:

```
=variable1 ~ DATA-TYPE
```

For languages that support more than one data type, one instantiates `DATA-TYPE` for each of them.

```
^ DATA-TYPE
^ NUMBER : DATA-TYPE
^ INTEGER : NUMBER
^ REAL : NUMBER
^ STRING : DATA-TYPE
```

Many high-level programming languages support user-defined types:

```
TYPE sometype
BEGIN
    child1 : type1
    child2 : type2
    child3 : type3
END
```

Each variable of the defined type is a compound of multiple elements, each of those being a variable of a particular type. The corresponding text would be something as:

```
^ TYPE : DATA-TYPE {
    ^ DATA-ELEMENT : DATA-TYPE
}
= some-type ~TYPE {
    =child1 ~DATA-ELEMENT :type1
    =child2 ~DATA-ELEMENT :type2
    =child3 ~DATA-ELEMENT :type3
}
```

Some programming languages support abstract data types that have some functionality, called classes or packages.

```
^ TYPE {
    ^ DATA-ELEMENT : DATA-TYPE
    ^ CODE-ELEMENT : CODE
}
```

One can have multiple code elements:

```
^ CONSTRUCTOR : CODE-ELEMENT
^ DESTRUCTOR : CODE-ELEMENT
^ STATIC-METHOD : CODE-ELEMENT
^ METHOD : CODE-ELEMENT
^ PROPERTY-ACCESSOR : CODE-ELEMENT
```

Functions

Many programming languages allow defining functions such as:

```
void dothis(int a, char *b) {
    dothat(a);
    [...]
}
```

This can be reduced to the following text:

```
^ CODE-BLOCK {
    ^ PARAMETER : DATA-TYPE
    ^ STATEMENT
}
=dothis ~CODE-BLOCK {
    =a ~PARAMETER : INTEGER
    =b ~PARAMETER : CHAR-POINTER
    ~STATEMENT : FUNCTION-CALL {
        ~FUNCTION ==dothat
        ~PARAMETER : INTEGER {
            ~VARIABLE a
        }
    }
}
```

An implicit function such as a lambda expression is simply an embedded function definition instead of a reference `==dothat`.

Comparison

Both the relational database model and the programming languages can be reduced to text. For each programming language one defines a set of base text units reflecting the grammar of the language, and then every program can be reduced to it. This is the same that a conventional compiler does, but here the syntax trees are general texts instead of a language and implementation dependent ad hoc structure to which the programmer has no access at all.

The fundamental structural difference between the text and today's languages is that, as we have seen, these languages are restricted to fixed types. You base upon some categories such as `CODE-BLOCK` and `DATA-TYPE`, these are established by the language and cannot be extended or replaced by the user. One can design text-oriented languages that allow the programmer to shape this categories. With such a language you can define say a `CLASS` and then you instantiate some classes, you can construct a special kind of functions that get the code by particular means or whatever. With a text-grounded approach the programmer is not restricted to use some predefined categories, she or he can construct some ones and change or extend them later on.

The second fundamental difference lies in usage. Of course you can use text-oriented languages as if they were conventional languages without changing your mind. But then you didn't get it. The implication of text-orientation is that the programmer specifies software and the compiler implements it. There is no more need for compilers to translate a `CODE-BLOCK` always as a particular sequence of assembly language instructions that put references to the parameters into the stack and transfer control to a particular memory address. There is no more need for compilers to translate a data type inflexibly as a particular sequence of bytes for storage purposes. Compilers are not restricted to linear translation of sentences into code any more.

3. Text Query

Query Languages

Formal languages are required to define, select and transform text units.

The relational database model is the great example. Its huge success shows how effective a sound theory can be. Think about how much work relational database management systems do every day all over the world. Think about how straightforward it is to query a large complex amount of data seeking for a particular information, how little work the programmer needs to do and how much of it is done by the system itself. There is a language available that has a tremendous expressive power, not because it is rich and complicated, but because it is simple and general and relies on a well-thought-out model that catches the deep structure of data in general.

We should follow the example set by the relational model and not be content with finding out a general text structure but providing it also with query languages. For this will be the key for its practical success.

How could text query languages look like?

Text is essentially hierarchical, therefore it suggests itself a multiple level notation each selecting particular nodes. Examples of such notations are CSS style sheets, that select HTML tags to apply style elements to, and XPATH, to select XML node lists.

While text selectors are handy and one can build compact simple expressions with them, they are not scalable and expressions get rapidly unclear as they grow. The full solution is a word-based query language allowing to build sentences and group them into parameterized functions and, in general, with all the means of fully developed higher-order programming languages. Examples of such languages are FLOWR for XML and SQL for the relational model, but both of them lack higher-order constructs and are much weaker than full programming languages.

A Text Selector Notation

Let us introduce a selector notation to query text.

The basic grammar of the selectors results from the text structure: selectors must choose text units having a particular name, role, or type. We can write simply a unit name with the usual prefix sign to indicate whether it is a unit name, a role name or a type name, and separate levels by a period.

To select a unit with a particular name, we write `=name`. If we select `:type` we get all instances of the given type or a descendant type of it. A tag `~role` selects all units that happen to play the given role. The role is the default selector, so that `role` is the same as `~role`.

Let us consider the example used above describing family Jones:

```
= Jones ~ family {
    = Ann ~ parent : woman
    = John ~ parent : man
    = Lena ~ child : woman
}
```

The selector `parent` returns `Ann` and `John`, `:woman` outputs `Ann` and `Lena`, and with `:person` we get `Ann`, `John` and `Lena`.

You can also restrict the results upon unstructured contents. If you have units such as:

```
~birth-date 6/24/1954
```

then you can select them with:

```
person . "6/24/1954" birth-date
```

This would return a unit playing the role `birth-date` and containing the given date string. You would usually rather want to get the person whose birth-date matches. For this purpose one defines the output level with the prefix `#`. If you select:

```
#person . "6/24/1954" birth-date
```

you get the unit `=Ann`.

To restrict the selection to more than one level, one writes more than one clause separated by a period. For example, to get all children of family Jones one selects:

```
=Jones . child
```

To get all first-level descendant units one uses a `?` for the level.

```
=Jones . ?
```

This matches all units under the unit named «Jones.» Using `??` instead matches the descendant units recursively, not restricting them to one level. If you want to get descendants up to the forth level, you select `??4`.

If a selector clause is prefixed with a number in parentheses, the results are limited to this count.

```
=Jones . (2):person
```

This outputs the second person from family Jones. The following retrieves the second and third ones:

```
=Jones . (2-3):person
```

If you do not specify a sort order, the results get the same order as the underlying text structure: first a unit and then its children ordered as they were fed.

With a prefix modifier – one gets the order reversed: first the last child until the first child, then the parent unit. If you have this text:

```
~books
~book Literary Machines
~book Augmenting Human Intellect
~book Software Pioneers
```

then with the selector `-book` you get:

```
~book Software Pioneers
~book Augmenting Human Intellect
~book Literary Machines
```

That refers to the order of the units, not to their contents. You can sort according to the binary contents of the returned units with < (ascending) and > (descending). For the text above with `<book` you get:

```
~book Augmenting Human Intellect
~book Literary Machines
~book Software Pioneers
```

The prefix modifier > is a shortcut for the modifiers `-<`.

This text selector notation can be extended in many ways in order to determine what results are to be retrieved.

4. Languages

Current Languages

Today, each programming language is an isolated kingdom with its own rules, infrastructure, and even culture. Text-orientation can lead to a new landscape of fully integrated programming languages, each with its own purpose and character but combining perfectly with each other.

The text-oriented conception of programming languages does not perceive them as means to code algorithms and data structures, but rather as means to code text. The code is seen not as consisting essentially in expressions in a particular language, but as consisting in parsed text structures that happen to have been entered by particular means and can be output at will in other terms.

With current languages one implements software. With text-oriented languages one specifies it, that's the basic principle. The programmer describes the software, the compiler constructs it.

In current programming languages there is already an incipient text-awareness, for example in macro expressions, in preprocessors such as PHP, in embedded languages such as SQL embedded in C, in general in language interpreters and evaluation functions. In each case the text is implemented as a character string which must be rewritten or interpreted, before it can be compiled or executed. Text-orientation does exactly the same thing, but not limiting text to a string expression, building upon parsed text instead. The current rudiments of text-awareness get fully developed by text-orientation, as they are replaced by a single principle that covers each of them and integrates them all.

Note on Lisp. Unfortunately I only know Lisp superficially. It is probably the most text-aware programming language that exists nowadays. See for example what the Lisp hacker Paul Graham says about it:

> Lisp looks strange not so much because it has a strange syntax as because it has no syntax; you express programs directly in the parse trees that get built behind the scenes when other languages are parsed, and these trees are made of lists, which are Lisp data structures.

Paul Graham: *Revenge of the Nerds*. In «Hackers & Painters,» O'Reilly, 1st ed., 2004, p. 188.

There is a close resemblance between this and my conception, with the difference that the underlying structure I propose for the parse trees is not a list, but the general text structure, and that I do not apply this structure to a single language, but to all of them. The main difference though is that Lisp remains an implementation language (it runs on an interpreter) and it is thus not so generally applicable as text as specification structure.

Concept of Language

Our theory does not consider as usual the language as a basic concept, and then define a particular text as its production. On the contrary, here the basic concept is the text, and a particular language can be recognized in an existing collection of texts. The text is universal, each language a particular one.

Language is a structural property of a text base. A language is a layer in a text corpus that provides some ground text units and ways to combine them. Language is urbanization: common things get handy, being expressed with minimal means. Language is infrastructure: it is a text factory that multiplies the productivity for general-used products. Language as a tool can be improved upon experience and transmitted between generations as heritage.

A language is not a closed set, there is always a margin in its bounds. If you analyze an existing text corpus, you can consider some text units to be part of the language or to be particular productions. If you create a language for producing new texts, there are many different limits you can set.

A language can have an own notation. It is a sign of maturity, but languages exist without one, and some have more than one. A notation is a user interface for a language, for both human and automated agents.

Language Multiplicity

The text theory states that languages are not intrinsically isolated from one another and their expressions can all be reduced to the general text structure. Languages are thus equivalent to each other as far as one can build the same text structures with them. What is the essence of a particular language? When talking about a particular language, one refers to particular semantics, or to a particular coding system, or to both.

The language semantics build a fundamental text layer all expressions are based upon. This does not get lost after parsing the code, but it has an effect afterwards. In fact the majority of expressions are small join sentences that just combine preexisting large amounts of assertions. In this respect languages are extremely important and have a direct impact on working systems.

On the other hand languages provide a coding system, an arrangement of lexical baggage, syntactical rules and notation that one can use to produce new texts and to express existing ones. Although this vanishes after parsing and does not affect the running system, it is extremely important as human interface to text. This has a technological aspect: a compact, precise, clear language is a tool that results in effective, sound work. This has a mental aspect, too. The language interface creates the user experience, the world one lives in when writing and reading. This determines how we understand and imagine things, too. The language coding is therefore for us close to the semantic structures, thinking about coding ways can lead to semantic improvements, and coding facilities provide access to semantic functionality without requiring theoretical instruction.

To sum up, language diversity is desirable. Despite the fact that a general-purpose text language is possible and useful, it should be used for theoretical and implementation purposes and as an optional universal, well-known way of expression, but it cannot replace the multiplicity of languages, each of which is unique, apart from semantics, for the technological and human aspects.

5. Text-Orientation

Text-orientation is the principle of perceiving the computer as a text machine. A computer system stores and transforms text, being text the structure behind many phenomenons, such as natural language sentences, programming languages, mathematics, and many more, including graphics, music and in general all kinds of digital media. The previously introduced concept of text can probably theoretically be improved, but it already shows that it is possible to reduce many different things to a simple algebraic text structure. The fact that syntax analysis, knowledge, mathematics and digital media can also be recorded as text, shows that text is a far more general structure as commonly thought of and it is not confined to writings. That is why I mean that computers are text machines. You can apply computing to anything, as far as you can record it as text. The verb *digitalize* is used in particular for recording media, but the operation taking place is exactly the same that happens say when you put data about customers and orders into a database. You reduce a complex phenomenon to a logical expression consisting of some symbols and some relationships between them. That is, you reduce it to text.

This principle indicates some chances. If we find out a general text structure suitable for representing real-life software structures in every computing field, then we can construct computer systems where all data structures and procedures are perfectly integrated with each other, because a underlying text-engine ties all of them together. The user does not need to be aware of this inner structure. The occasional user of a user-friendly text-oriented system still sees different parts with different look and feel, but the system is grounded on a unified layer underneath some separate presentation layers. The more skilled the user is, the more she understands the underlying logic and the more means she has to interact with the system through distinct interfaces for querying and transforming data.

This applies to every field one uses computers for, including software development. Understanding the whole software as text leads to the definitive separation from specification and implementation the computing field has longed for. It makes also possible to construct far more flexible and scalable development systems than today's ones that put unprecedented power at the fingertips of highly skilled people.

Specification and Implementation

An implementation of text can certainly be useful, and it can provide services that otherwise do not exist. But the full power of text lies in its adoption for specification purposes. The key aspect of text-oriented programming is being aware that software is text, along with a fundamental, general concept of text that makes it possible to express every text in terms of it.

The other programming paradigms determine what the basic elements of «the world» are. For example the object-oriented paradigm states that there are objects that exchange messages, the relational model states that there are domains and relationships between them. Note that this is the base architecture of all software systems that are build on them. Not only this, they need run-time support: objects have to be allocated, messages must be passed, and the relational model needs even a dedicated monolithic server application.

The text-oriented paradigm is entirely different. It does not say anything about how «the world» is, but about our description of it. Text-oriented software development does not determine an architecture of the software to be build and it has as a matter of principle no run time effects at all. Text-orientation refers to the structure of the source code, neither to the structure of the executable binary files nor the structure of the running processes.

You cannot describe a running relational database in terms of object-orientation, it simply does not match the schema. You also cannot describe an object-oriented user interface in terms of tables and queries. But you can describe both of them and in fact every other system in terms of text-orientation.

All other software paradigms are implementation paradigms, on the contrary text-orientation is a specification paradigm. That means also: text-orientation does not replace the other programming paradigms; it furnishes a ground for each of them and integrates them all in a consistent whole.

Part II

Imagine

6. Text-Oriented IDE

Let us now imagine how a text-oriented Integrated Development Environment could look and feel.

Let us assume a graphical environment with pointing devices and movable, sizable windows containing some dockable, sizable frames, and concentrate on the text-based user interface and the system's functionality.

Source Selectors

One could proceed this way.

You open up a new window with an empty left sidebar and an empty main frame.

You bind the sidebar to a selector, say:

```
projects . =myProgram . :t
```

and set its presentation mode to `tree view`. Now you see the project's outline, containing for example these chapters: requirements, analysis, specification, design, code. You click on a chapter and see its subchapters. You set the main frame's source to show the contents of the element being currently selected in the sidebar. You can edit the project's outline at place, you can easily hide and show some items of it.

Suppose you are looking for a particular place, then you change the sidebar's source selector, say:

```
projects
   .=myProgram
      .#emails
         ."user interface" tag
            .>timestamp
```

to get the list of all electronic mail messages about the user interface sorted by time descending. You click on one of them and the message appears in the main frame. The workbench knows that this is an e-mail and renders it accordingly: One sees a header containing the sender name, subject and time, and below the message's contents as prose. The message contains a reference to a particular place in code. You can easily jump to it either in the main frame or in a new one or in a new window. The system knows, this is say «C» code and renders it as such.

You can write selector expressions in many languages. This example uses the selector notation introduced in «Text Query» (p. 30 ff.), but it could be a complete text-query language or even plain text language. This expression could be translated between languages by the workbench.

Browsing Code

Browsing code is not limited to fixed *files*. You can get a whole module in one window frame, but you can get say the whole code related to a particular functionality that can be dispersed throughout modules. You can also see the code for a particular transaction, e.g. what did I yesterday? You can also see all functions that invoke a particular function. You get all this by simply changing the source selector.

Browsing code is not bound to a particular *software system* such as a database management system or a development platform. The presentation is the same if you watch a list of tables and fields of a relational database or if it is a list of class members. You see it with the same user interface and you can apply the same operations to it, such as renaming, sorting and filtering. How this is implemented, does not make any difference.

So-called «code» and so-called «documentation» are seen and operated through the same user interface, too. You can see the whole documentation in more or less detail at will. When you point to a paragraph, you can get details about some aspects of it, such as a discussion between developers or with the management about it, work progress status or test results. When viewing a requirements document, you can select a particular sentence and see at a glance what parts of the specification, of the design and of code are affected by it.

When you view some code, you can control through the source selector if you want to see the related paragraphs at the user's guide, the specification or some implementation notes. Each function has natural language sentences describing its purpose, arguments, return values, exceptions, etc. You can see the code with some or all these informations embedded before each function's body, or at a synchronized frame, or overlay it on demand. If you are wondering why something was done this way and you don't know if it is a bug or it has some recondite reason, you can overlay the discussion between programmers about these particular lines.

The code can be alternatively rendered and updated as diagram, for example entity-relationship and class diagrams.

It is frequent in code to have functions that call other functions, all of them being defined in the program. One can choose to see this code parts top-down (first the caller and then the called functions) or bottom-up. One can define a custom order for a code view and hide some items. One can save code views and recall them later.

You can add your comments, to do tasks or questions to clarify. You can share your code views with your team, open discussion pages about particular issues, ask particular persons about particular parts, reply the questions you have been asked and overlay the context documentation. The discussions can be seen ordered by thread, time, people or with regard to content such as concerned classes, tables, requirements or emerged issues.

Code Transformation

Code is not just shown the way it was fed, it can be transformed at will. All you have to do is change the presentation mode through the source selector.

There are equivalent languages, for example those of the .NET environment, one can see the code in any of them and switch easily. If one enters a code part written in a completely unknown language or a particularly complex expression, one can always switch to the plain text language and get complete clearness about it.

Code is not bound to particular identifiers, either. A code one is familiar with can be shown with short identifiers consisting of a few letters, thus being compact expressed. If one dives into unfamiliar regions, perhaps something one wrote a long time ago or something written by others, one can get it expressed with long, significant identifiers. One can define and update name systems at will and establish correspondences between them. Some of them are volatile, just for a single session, some are for personal use, some team-wide or organization-wide and some worldwide.

7. Text-Oriented Programming Languages

Grammatical Elements

Programming languages should support grammatical elements. For example, instead of writing:

```
string condition;
if(_byID)
    condition = "WHERE ID ="+ToString(ID);
else if(_byName)
    condition = "WHERE name ="
        +CHAR(32)+name+CHAR(32);
EXEC("SELECT * FROM customers "+condition);
```

one should write something as:

```
SqlClause condition;
if(_byID)
    condition = WHERE ID = {ID};
else if(_byName)
    condition = WHERE name = {Name};
EXEC SELECT * FROM customers {condition};
```

Note that in the first case we have a main language with some embedded strings, whereas in the second case we use a blended expression in two languages, both of which get parsed from the beginning and can cause compile time errors due to incorrect syntax, both of which get syntax highlighting and context-sensitive help in the IDE, etc. A grammatical element looks like a variable, but it is not necessarily. Some of them are compiled as variable, some of them are replaced at compile time and get no runtime counterpart at all.

Grammatical elements can of course also be used inside a single language, for example the above code would look in a SQL stored procedure like that:

```
    DECLARE @condition CLAUSE;
    IF @byID = 1
        SELECT @condition =
            WHERE ID = {@ID};
    ELSE IF @byName = 1
        SELECT @condition =
            WHERE name = {@Name};
    SELECT * FROM customers {@condition};
```

Grammatical elements can be local to a function, but they can be defined at application or organization level, too. That will simplify the code a lot. For example, today in database queries one writes again and again things like that:

```
FROM customers
    INNER JOIN orders
        ON customers.ID = orders.customerID
```

This happens often not just with two tables, but with many: from customer to orders, from orders to products, to parts and prices, to availability and shipment history, etc. One should write such conditions only once and after that just mention them:

```
FROM {customer orders}
```

Not only this is shorter and easier to remember, but you can change the table structure and identifiers at any time.

Cut & Paste

Who has not done it? You are in a hurry, you want to get the job done, you remember you saw once something similar... And then you do it: you copy it and paste it and alter it a little bit. The consequence of this: after some years working, after some personal changes, the whole source code is just a big mess of similar chunks of code, whose differences of course nobody can tell. Text-oriented development offers two complementary approaches. On the one hand, it does not just copy and paste, but it records automatically the source of each code snippet. You can always see where some lines originated and compare both, you can always see all places that derive from a particular one and revise them. On the other hand, you have powerful means to unify them, from the beginning or later on. You can define similarities once and refer to them by name at multiple places. That applies not only to whole functions or classes as today usual, but also to segments of instructions or handling aspects, and does not need to have run time effects at all.

8. Files and Text

As Ted Nelson points out, the files are currently chunks of data owned by particular applications and the operating system has no access to their contents. This could be superseded by a text-oriented approach that would make the operating system responsible for managing a text structure including all software and data available. Imagine a text-engine that would store a text structure as presented in this book. Imagine that the entered text could be efficiently queried and navigated. The file system would be replaced by the text structure itself: the operating system would know not only about unit names, but also about roles and types. Instead of running `ls /usr/bin` one would use `ls :program` to list all units of type «program» or

```
ls =job.#document."Jane" modified
```

to get all documents under the name «job» that were modified by Jane. But one would also get the first paragraph of each article with `ls :article.(1)p` and all references to Xanadu inside them with

```
ls :article.#p."Xanadu" ref
```

An application such as a word processor would not *own* a chunk of data any more. To present a particular book the user is working on, the application would retrieve it from the operating system by getting

```
=my-new-book.:prose
```

that would return a list of all prose elements. The application would render it and let the user update it. But the same book could be operated by say a spreadsheet application that would be interested in

```
=my-new-book.:chapter.:status
```

that is in the titles of the book's chapters and their status: «work in progress,» «review» etc. and would show these in tabular form for the user to manage the project of writing the book. In this scenario there are still separate applications with particular purposes, but the information they work on is unique and perfectly integrated.

9. Programs and Text

In current operating systems there are some *executable files* called programs that the user invokes explicitly. In a fully developed text-oriented operating system there are no more separate closed files at all. The functionality is tied together with the text structure and is exposed as some unit types. Whenever a type is instantiated, the user gets all transformation possibilities exposed by all related software packages that are available on the system. When installing software, new units are appended to the text repository and some of them can be tied with executable images such as parsers, presentation managers and transformations that will be invoked by the operating system whenever required.

For example, after installing a word processor the system knows about a unit type `Prose.` Each time the user instantiates it, for example:

```
= myNextProject {
    = Vision : Prose
}
```

she or he can open `Vision` in a separate window for word processing purposes, or print it, or export it into an electronic book format. The user can also add own functionality to the unit `Prose`, for example through a script that automates certain manipulation steps or through a complete program. If one has installed more than one package that can process units of some type, one can work on each instance with any of them.

The types are not big bags comparable to today's «file extensions,» they can be very fine grained. For example, although there is a `Prose` type, it contains subtypes such as `Paragraph`, `List` or `Heading`, each of which can be attached to third party software packages or the user's own developments. For example, after installing a spell checker software you can check all your natural language sentences in all articles that you write with a word processor, all lists that you write through a spreadsheet software, all your notes, electronic mail messages, etc. In a fully developed text-oriented operating system that would be valid in general without exception out of the box.

Note that this presupposes either a standard ontology of common types or a standard equivalence system to map types from different software publishers to each other, in order for software pieces from different publishers to cooperate well together. This is certainly difficult to achieve, much more difficult than today's private agreements between two publishers or the pure imposition by the big ones. But if we manage to do it, the systems will be incomparably better integrated than now.

10. Text-Oriented Compiling

In text-oriented development, the process of producing binary executable files from source code is not the job of a stand-alone, closed program called «compiler.» There are no fixed, stand-alone strings called «source files,» either. Instead, there is a single parsed text structure that contains several layers describing the software product that is written in several languages. There are prose writings such as a goal description, requirements, discussion messages between team members, there are formal writings such as specification and design documents, there are one or more layers in programming languages, and at the bottom there is an assembly language layer.

Software Source Structure

The software source is a parsed text structure that consists of text units with relationships among them. An e-mail is a text unit consisting of a string text unit having the subject and a prose text unit having the message. A code function is a text unit consisting of a signature text unit and a body text unit. The function body text unit consists in turn of some text units as the result of its syntax analysis: an if-unit, a for-each-unit, etc. An assembly instruction text unit consists say of an operation plus a parameter, being both again text units.

There are dependencies between units. For example: one took a decision in an e-mail after a discussion, as a consequence a line of a function must be changed. This dependency is recorded at the repository. A dependency can be bound to a text transformation operation that describes how to get the target unit from the source one. Some of these transformations are executed automatically, others manually. They can be just a hint of what to do such as «test it,» a procedure description in prose or in a script language, or a complex program.

Compiling

The last layer of a software artifact is the assembly layer. That is the complete representation of the product by means of assembly language for a particular processor architecture. The assembly layer integrates with the rest of the text repository. It is not inflexibly generated by a «compiler,» some parts of it may be automatically generated and some other parts handcrafted. It can consist of more than one layer, for example a generic one that is architecture-independent and some architecture-specific modules.

In a text-oriented environment one can still call «compiler» a software piece than generates assembly code by transforming units written in a particular programming language, or, more accurately, units having particular semantics. Such compiler is not a program that runs on some «source files» and produces «executable files,» it is a program that receives a text unit as input, transforms it, and gives a text unit back. Generating a program is not calling one compiler once, but letting the system transform one text unit once, being the system responsible for invoking all needed compilers on the required units that compose it.

Such a compiler does not parse source code, of course. It does not get character strings as input, but already parsed text that it can navigate and query.

You may be asking: «but what about the actual executable files?» Well, executable files are just a particular representation of assembly text units. Once you get the assembly text, you can always export it as binary file.

Part III

Case Studies

11. Sample: Program Parameters

Let us consider an example of text-oriented programming.

Every program has constants or parameters that control its behavior. The programmer has many choices, among others: a) hard-code them as literals, b) define code constants, c) put them in a configuration file that is read at run time, d) define them as command line arguments, e) put them in a database and retrieve them at run time.

With today's means the programmer must make a commitment. The way she or he will refer to those values in code is tied with its implementation. The decision must take place at the beginning and to change it later on is time-consuming — if still possible at all.

The text-oriented approach is different. The programmer defines a symbol say `parameter` and instantiates it for each of the needed parameters. The whole code refers to those symbols by name. At the beginning of the development the values are perhaps hard-coded constants, one defines `parameter` to be of type `constant` and assigns each of them a value. Later on one changes the definition of `parameter` that gets a configuration file value. Note that here just one symbol was changed, the rest of the code remains unaltered. The compiler translates each read access to the parameters according to the current definition of `symbol`, and after a change the generated code looks very different in many places, but the source code changed only at one place. Note that there is no run time penalty at all, support for the generality of the `parameter` symbol is pure compile time support, and the programmer can determine the implementation details if desired with unlimited precision for a single case.

Apart from that, a text-oriented environment gives the programmer multiple ways of interacting with the parameter values. Today, when developing, one must put the parameters according to its implementation: either at a particular file, or at a particular database table, or at some source code files. The values can only be seen and updated at this place. In a text-oriented IDE the parameter values can be shown and updated independently of its implementation. When you are browsing some code functions that use some parameters, you can overlay the parameter values and change them at place. You can also get the code with parameter values expressed as literals, not showing the parameter names but their actual values instead, thus getting very readable code, and let the editor overlay some parameter attributes for single parameters on demand. You can also get a table with all parameters related to a particular functionality or a program. The text engine saves them only once, but your access is multiple. Note that this is implementation independent. You can get a single parameter list containing all of them: source code constants, configuration file values, database record values, etc. And you can switch the implementation of a single parameter at every time if you want to.

12. Unix: A Text-Aware Environment

Unix is not as many operating systems a mere conglomerate of services, but it is based upon a design principle that organizes all the software — not only the base system but also user defined commands and third party programs — as a consistent whole. A Unix system builds an integrated, powerful, clean development environment. What makes Unix different from other systems is its text-awareness. It is not really text-oriented, because it does not base upon an explicit unified concept of text, but it can be perceived as a predecessor of text-orientation.

At the heart of the system there is a text structure, the file system, that organizes and unifies all system resources. It defines a consistent name space for all entities the system is aware of.

The file is itself a variant of text, too, although it is restricted to a list of character strings. The file is used also as interface between programs, being standard input and output, pipes, etc., all just files. The file is also used as «run control file» to pass parameters to programs. This allows a simple text editor to be a universal tool for controlling all aspects of the system's functionality regardless of software packages. This facilitates the programmatic control over the system's behavior, too, because all filters and tools that edit text files can be used to change settings. Note the extreme efficiency of this approach and the high costs one would have to pay in non-text-based systems to get similar functionality. In fact non-text-based systems cannot afford such functionality and remain much more limited.

The shell is a language-based interface to the operating system. Its grammar consisting of a verb (command), prefixed options and arguments is simple and general, easy to learn and consistently supported by script and programming languages. The user can define custom verbs either through the shell linguistic mechanisms such as alias and user-defined functions or through shell scripts or programs.

The linguistic style is not restricted to a particular way of «interacting» with the system, its principle is throughout applied as design principle and conforms the overall software architecture. One does not think of software as operating system plus some «applications,» one thinks of it rather as a pool of available «commands.» An *application* is a stand-alone software unit that gets total control of a process — it determines its flow, data and interfaces — in order to perform a complex task. A *command* is a special-purpose software unit that receives a context and applies a specific transformation to it. To get a job done, the user builds a complex expression that combines multiple commands and lets the system execute it. Whereas in an application-oriented system you need a new application or at least a programmatic extension to an existing one for each new kind of job to be done, in a Unix system you just need to build a new sentence to get it done. As a result, a user of an application-oriented operating system is a prisoner whose possibilities are restricted to a closed set of functionality provided out of the box by the installed software packages, whereas a Unix user is a free citizen than can get all sorts of new functionality just combining predefined components at will.

Not only Unix as a product is text-aware, but also the culture that has grown around it. Its programmers share some good style norms that are all text-centered. They praise human readable text formats for control and inter-process communication, simplicity in the parts, the complexity being handled by intelligent combination, succinctness but completeness in documentation, transparency in source code. These are all values that refer to good qualities of text.

The text-awareness of Unix relates to the traditional core of the system. Unfortunately later extensions loose this principle and are in my opinion not progress, but decadence. The graphical desktop surfaces lack any text-awareness. The fashionable XML is a monster. With its unreadable appearance, these files destroy the legibility and compactness of the Unix configuration files and the richness of multiple notations, each of which was designed for a particular purpose. Further XML is not line-oriented and thus all system tools and mechanisms for querying and updating files are useless. In general the traditional values of the «old Unix school» vanish gradually, as the operating system becomes mainstream and inevitably bloats.

The right approach is followed by Unix's successor Plan 9, an interesting experimental operating system that is even more text-aware and more consistent.

13. Universaltext Interpreter

The Universaltext Interpreter is an experimental software written in Perl by me that implements the text structure presented in this book. It is a tool to produce written output (e.g. a prose book to be published, a dictionary in both electronic and paper format, a website) based upon source information in plain text files or in word processor documents. With a Perl script you can navigate and query the source text structure and generate some documents that extract all or some part of the contents.

This program implements a parser for the Universaltext Language (UTL) and supports alternate custom parsers. It implements text selectors for querying the text structure. It supports binding output processors at run time for the sake of generating different file formats out of the same source. It is open source and can be found on the Web at `http://u-tx.net`.

Universaltext Language

UTL is an implementation of the plain text language that is used throughout this book. It has a line-oriented format, each line defining one text unit and curly brackets enclosing text levels. Example:

```
^species {
    ^common-name :string
    ^scientific-name :string
}
^mammal : species
```

This defines a species as having a common and a scientific name and mammals being species.

The interpreter ensures the logical integrity of the entered text. Each symbol being mentioned must be previously defined, each role must be one of the type's children and have a coherent type. If one tries to define a text that violates the integrity rules, the parsing aborts with an error message.

UTL is designed to be embedded in prose and be non obtrusive. When entering a string you usually do not need to put it in quotation marks, the parser considers the first word without prefix (~, `:`, etc.) up to the end of the line to be binary data.

The parser knows about the text structure and infers which parent a child definition applies to without needing explicit curly brackets. For example the following is valid UTL (indentation is syntactically irrelevant):

```
^elephant : mammal
~elephant
  ~common-name savanna elephant
  ~scientific-name Loxodonta africana africana
^penguin : species
~penguin
  ~common-name Little Blue Penguin
  ~scientific-name Eudyptula minor
```

UTL considers the first child to be the default role, which is automatically assumed if a line consists just of binary data. For example, if you define prose as consisting of paragraphs and headings this way:

```
^ Prose {
    ^p : string
    ^h : string
}
```

A paragraph is assumed if the role is not explicit. You can enter prose with these lines:

```
~Prose
~h First Chapter
This is the first paragraph...
This is the second paragraph...
```

This way a prose document is very readable, having just occasional embedded tags that define its structure.

Alternate Parsers

UTL is a general-purpose text language appropriate for prose, that is sparse structured character strings. It becomes rather verbose for expressing dense structures such as a table. In order to embed in UTL portions of text in another syntax one can use alternate parsers. For example instead of defining a table this way:

```
~table
~line
    ~column 1.1
    ~column 1.2
~line
    ~column 2.1
    ~column 2.2
```

you could enter this:

```
~table [
1.1 1.2
2.1 2.2
]
```

The square brackets instead of curly brackets instruct the parser to call the alternate parser defined for the type `table` and pass to it the following two lines. To set up a parser for a type one declares the type this way:

```
^table {
    ~parser main::parseTable
    ^line {
        ^column : string
    }
}
```

The Perl function `parseTable` from name space `main` is called whenever a table has to be parsed. This function gets as parameter an object representing the interpreter that can be used to feed text and exposes a method `readline` that returns the next line to be parsed.

It is quite straightforward to define alternate parsers for dense structures that one uses throughout the source files and it makes the source documents more readable and maintainable.

Feeding Text

One can feed text either through UTL text files that are read and parsed by the interpreter or programmatically. With a Perl script one can generate units, for example:

```
$ut = new UText;
$ut->def({role=>"elephant"});
$ut->enter();
$ut->def({
    role=>"common-name",
    bin=>"savanna elephant"
});
$ut->leave();
```

One can also feed UTL expressions to be parsed:

```
$ut->read(<<"END");
~elephant
    ~common-name savanna elephant
END
```

It is very easy to feed UTL files that were written with a word processor instead of plain text files. A Perl script just needs to extract their contents and perform a `read` on them.

One can also write a Perl script that reads a regular word processor file without UTL tags and generates a text structure based upon the document's structure. For example, the headings are translated to `~h` units, the paragraphs to `~p`, etc. With some amount of programming one can feed any structured documents (including spreadsheets, outlines, etc.) into the interpreter.

Text Selectors

Once the text has been fed, it can be exploited. The interpreter implements selectors as described in the chapter «Text Query» (p. 30 ff.).

The function `getVar` returns the binary contents of the first element returned by the given selector. To output the title of the current chapter, one writes:

```
print $ut -> getVar("title");
```

That outputs for example «First chapter.» To get the second paragraph, one puts:

```
print $ut -> getVar("(2)p");
```

The function `foreach` iterates over all items returned by a selector. For example, to get all known species that are elephants:

```
$ut -> foreach("elephant", \&doSomething);
```

This calls the function `doSomething` once for each species that is an elephant.

In UTL one can embed a tag `[v <selector>]` that gets expanded by the interpreter with `getVar`. For example, if you have this text:

```
^ book {
    ^ title : string
    ^ chapter {
        ^ title : string
        ^ p : string
    }
}
~book
~title Teach It Yourself
~chapter Preface
~p Reading "[v book.title]"
   will improve your skills.
```

A call to `getVar` for the first paragraph will return:

> Reading "Teach It Yourself" will improve your skills.

Note that the selector is `book.title`, if it were just `title` it would return «Preface.»

A loop over a selector can be included with a tag `[foreach]`. For example:

```
[foreach/ elephant]
[v common-name] (sc. [v scientific-name])
[/foreach]
```

This will get expanded say as:

> savanna elephant (sc. Loxodonta africana africana)
>
> Asian elephant (sc. Elephas maximus)

The selectors can be simply a role name but they can be complex query expressions, too. You can get for example the elephant list sorted by common name:

```
[foreach/ #elephant.<common-name]
```

And you can get all species described by Linnaeus with:

```
[foreach/ #species."Linnaeus" described-by]
```

provided you have a definition such as:

```
^species {
    ...
    ^described-by :string
}
```

Output Processors

It is possible to define custom tags and provide Perl code for expanding them. For example, one can define tags `[ul]` and `[li]` to declare item lists. The source text could look like that:

```
[ul/]
[foreach/ elephant]
    [li/][v common-name][/li]
[/foreach]
[/ul]
```

If you want a Perl script to generate a web page, you can bind both tags `ul` and `li` to a function that returns a HTML tag this way:

```
$ut->set_out_binding("HTML","ul",\&tag);
$ut->set_out_binding("HTML","li",\&tag);
sub tag
{
my ($self,$all,$op,$mod,$param,$str) = @_;
return "<$op>$str</$op>";
}
```

This tells the interpreter to call the function `tag` for rendering each found `ul` and `li` tags. (Actually a script does not need to provide these particular bindings since they are already set by default by the HTML module of the interpreter.)

The elephant list will then expand as:

```
<ul>
    <li>savanna elephant</li>
    <li>Asian elephant</li>
</ul>
```

Your Perl script can not only generate a web site but also a source LaTeX file for a printed book from the same source file. For this you would change the function bindings to something like this:

```
$ut->set_out_binding("tex","ul",\&ul);
$ut->set_out_binding("tex","li",\&li);
sub ul
{
my ($self,$all,$op,$mod,$param,$str) = @_;
return <<"END";
    \\begin{itemize}
    $str
    \\end{itemize}
END
}
sub li
{
my ($self,$all,$op,$mod,$param,$str) = @_;
return "\\item $str ";
}
```

The elephant list will then expand as:

```
\begin{itemize}
    \item savanna elephant
    \item Asian elephant
\end{itemize}
```

The strings with embedded tags `[...]` are parsed by the interpreter, that calls each time the bound function. The bound function gets as parameter the interpreter object, the tag name, the tag parameters and the contents:

```
[tag-name/ parameters]
content
[/tag-name]
```

The bound function must return the expanded text. The bound function must expand explicitly the parameters and the contents, if they could contain tags to be expanded. That makes it possible to define tags such as `[foreach]`, whose contents should never be expanded before calling the bound processor. If they were, this code:

```
[foreach/ chapter][v title][/foreach]
```

would not output the chapter titles but repeat the title of the book so many times as there are chapters.

To expand the parameters and the contents, the bound function calls the method `out` from the interpreter object:

```
$param = $self -> out($param);
$str = $self -> out($str);
```

Sample: Website generation

The combination of text selectors, predefined tags and custom tags makes it pretty easy to generate output documents. Let us see how one can generate for example a web site.

One can define a structure like this:

```
^ webpage {
    ^ title : string
    ^ content {
        ^ p : string
        ^ h1 : string
    }
```

One can populate the above structure with handmade sources:

```
~webpage =index {
~title Some Web Site
~content
~h1 [v title]
Welcome to [v title]!
}
~webpage =contact {
~title Contact
~content
~h1 [v title]
You can contact me at jane@example.com
}
```

But one can also populate the `webpage` units through the interpreter querying some other texts, for example you collect information about elephant species in a text structure as mentioned above and a Perl script generates UTL creating for each species a single web page describing it and a table of contents listing them all.

To generate the actual HTML files one can run a script that loops over the web pages and generates a file for each of them. The script could look like this:

```
$ut->foreach("webpage", <<"END");
[save/ [u].html]
<html>
<head>
    <title>[v title]</title>
</head>
<body>
[foreach/ content.?]
[if/]
    :h1     <h1>[v]</h1>
    :p      <p>[v]</p>
[/if]
[/foreach]
</body>
</html>
[/save]
END
```

The tag `[u]` expands as the current unit's name, `[save]` saves its contents as a file with the given name, `[if]` is a conditional operator that expands differently according to the current unit's type. The above generates two files:

```
index.html
contact.html
```

These are the contents of the index file:

```
<html>
<head>
    <title>Some Web Site</title>
</head>
<body>
<h1>Some Web Site</h1>
<p>Welcome to Some Web Site!</p>
</body>
</html>
```

Internals

Let us see now how the universaltext interpreter manages the text structure. The interpreter loads at boot time the following two text units:

```
^unit {
    ^unit
    ^binary
}
```

The symbol `unit` is the root text unit that is its own parent, type and role. All other text units in the system are descendants of it. This unit has the base functionality that applies to every text unit, for example parser support.

The unit named `binary` is a child from `unit` that holds text units that contain data to be stored and retrieved in binary form. Binary data is unstructured data, you cannot query and navigate it, you can just store it and retrieve it as a whole.

The implementation of binary units is hard-coded and cannot be overriden. If you need some unit to have binary data, you must define it as a having the type binary or, more commonly, as having the type of a descendant unit of binary. The interpreter defines additionally some common binary units such as `string` and `cardinal` that are available to be instantiated.

If you need a unit to contain strings, you define for example this:

```
^ title : string
```

Then you can enter binary data when defining a title:

```
~title Chapter One: Introduction
```

One can obviously build new types on the top of that. If one defines a subtype of a binary type, it becomes automatically a binary type, too.

```
^ file {
    ^ line : string
    ^ comment : line
    ^ code : line
}
=to-do ~file {
    ~comment That must be done:
    ~code int v(); //yields the current value
}
```

Here both `comment` and `code` are lines, thus strings, therefore they accept binary data.

The text structure is implemented as an array of scalars:

```
UNITS[ID] = (RID, TID, PID)
```

Each unit has an internal numerical identifier (a counter), for each unit one stores the identifiers of its role, its type and its parent. At boot time the units `unit` and `binary` are loaded with identifiers respectively 0 and 1.

```
UNITS[0] = (0, 0, 0)
UNITS[1] = (1, 1, 0)
```

Next Steps

The experimentation with the text structure has shown its usefulness for practical purposes, it is very expressive and flexible and one can make major semantic changes afterwards without having to manually update many places. The main limitation of the universaltext interpreter arises from the implementation of embedded tags. Now tags are parsed by the interpreter at run time and processed by special-purpose Perl code. They should be parsed by the interpreter at feed time instead and produce regular text units that can be queried and output by regular means. That would make the interpreter much more text-oriented and thus simpler and more powerful.

The next project can be a text-engine that not only parses and transforms text as the interpreter, but it can store it persistently. This way one could maintain all documents and media files in a text repository. A client would check in and check out files, although these files would not be stored as binaries, but as parsed text. What you check in is not necessarily the same that you check out. For example one can check in the website definition above and from the server check out the HTML pages, the text-engine generates them on request according to the current contents. If you are writing a book, you can check out a single chapter or the whole book as a word processor document, update it and check it in. Or get it as a LaTeX source file or as PDF. And you can also check out the book's outline as a spreadsheet document and update it. The text transformations are stored in the text repository and invoked automatically when required. The text engine knows about the dependencies between text units and updates them whenever necessary.

Basing on the text-engine one can develop a text workbench. A text workbench is essentially a text editor that knows about the text structure. Instead of checking out a document as word processor or spread sheet document and use the correspondent application to edit it, at the text workbench you can query the text repository and view the results as prose, list or tree and change the view at will. If you edit for example a prose writing that has `~h` headings and `~p` paragraphs, you can see the prefixed signs before the units, or you can hide them and just use another font size for headings, or both, and you can switch between presentation modes interactively. You can define your own views for particular unit types or single units, this begins customizing appearance and ends with programmed add-ins that provide specialized rendering and editing capabilities, for example a programmer's plug-in that supports development.

Part IV

Background

14. What is Text?

There are commonalities between different fields that can be led back to text usage. For example, in humanities one has a text corpus recording a set of writings, one has a concordance showing all occurrences of each particular word in the corpus, and one has a dictionary recalling what meanings the words were used in the corpus with. And in software engineering one has a source code repository, the IDE facilitates passing through all instances of a type or all times a particular function is invoked and jumping to the place in the source code where they are defined. In both cases the involved structures are identical. The people that work in humanities and software development are of course thinking about very different things, but the logical structure they use is the same. I call this structure «text.» A text is a symbolic structure that is expressed by means of language. There are properties of the text by itself: it can be ordered as a hierarchy, as a list, etc., some parts of it refer to others as in cross-references, quotations, hyperlinks, etc., and there are properties of the language itself: it consists of comparatively few words — the *lexis*— that can be chained up in different ways to produce a huge amount of different sentences, they combine flexibly, but not completely free —the *grammatic*— and for a particular text or text group they can be given a particular meaning. The usage of text is not restricted to both fields mentioned above. Practically all human activities are infused with text. Consider just law, politics, science, business. Everywhere there are prose writings, there are dictionaries, there are lists of things to do, to evaluate or to talk about. If we analyze and understand the general text structure we will be able to construct computer systems that assist us to handle with text in general. Computer aided text management will benefit many fields.

15. What is Text-Orientation?

One becomes text-aware when one realizes that everything one does with a computer is manipulating text. For example in software development. In a relational database system you create a table, you insert some records, you perform a query. But what are you then *really* doing? You are *writing* SQL statements and letting the computer run them. You write some scripts, too, and you write some sentences in programming languages called *source code*. Working does not dirty your hands and your muscles do not get tired from it. Why? Because you are writing.

The current technologies hide this fact by offering for each task a completely different tool. The operating system's shell is one thing, then you have a separate surface to manage the relational database, and you have also a stand-alone program that is not very compatible with them and is paradoxically called *integrated development environment*. The look and feel of each of those is completely different, they are not tied together, and therefore you experience each action and think of it as if it were a completely different thing.

One adopts a text-oriented attitude when one arrives to this decision: it *is* text, let us handle it *as* text, too! What is programming? Programming is saying things. You say «do this,» «show me that,» «modify it this way,» that's all. When you speak you don't repeat long sentences again and again, you just say «as I already said.» You introduce once a name for a complex state of facts, and then just repeatedly point to it by name. Programming can also be so easy and it must be. It is unnecessary complication, to coerce you to say some kind of things in a particular window with particular keyboard shortcuts while some other kind of things must be entered by clicking on a tree view with the mouse or typing a sentence with a particular syntax. It should be possible to say the same things by different means, it should be possible for the user to pick up the form of expression as she or he sees fit each time. It is unnecessary complication, to have to say the same sentences again and again, instead of using a pronoun and saying just «now *that* again.» It is unnecessary complication to have to code the same algorithm over and over, just because you happen to have written it in a particular language and you need it now in another one or because this time there is a slight variation in it. It should be possible to say just «same but instead of this that.» Programming is speaking to the computer. It can be so easy as speaking. And it must be.

16. Just Once: A Programming Ideal

One should be able to open the development environment and whisper: «Now listen carefully, I won't say it twice.»

What are languages? They are means for expressing things.

What are programming languages? They are means for expressing software.

If you have to formulate the same thing twice, it is bad designed. You did express it once, right? You did take care of the details once, right? Now it's the computer's turn.

Let the computer do what it can do and do yourself what is reserved to humans. Is there a pattern? Can it be generalized? If so, then specify the logic once and let the machine apply the logical consequences each single time. If you do yourself the routine work, not only have you an overhead when writing new parts and there are necessarily more bugs, but you must also from now on retain the details in memory — and you can retain just a small amount of things in your memory, and you do not share your memory with your team —, furthermore, if you decide to change something in it, you will have to manually update many places. That requires not only your time and your energy, but becomes with today's technology frequently impossible. For example: it is nowadays not feasible to change a field name in a large organization's database because that would break the applications that query it. This is not a bagatelle, it forces uncontrolled growth that leads to intermingled, obscure software the programmers can no longer manage.

Source code does not *construct* software but *describe* it. When you write some sentences, you are deciding how something must be done — you are not doing it yourself. This applies to the most general business rule you write in the analysis as well as to an assembly language instruction you write in a device driver. You are not the one who puts a value in a register, it is the processor, you just determine what has to be done.

Programming is ideally taking each decision at exactly one place and extracting consequences from it at many places. The source code must be tight organized as a web of dependencies, in order for the programmers to retain control over it. Insufficient structure sets scalability limits. The place where a particular decision is taken must be to a programmer obvious and easy to access. The definition must be clear and concise, susceptible to be extended or changed. To sum up, programming is specificating, ideally every single thing just once and at a traceable place. The motto of this approach is: if it can be said, it can be programmed; it should be explained once and mentioned oft-times.

This ideal of programming is both a question of mentality and of technology. You can begin right now to think this way, you can use current technologies trying to specify things instead of implementing them and get the best of languages and systems at your disposal. For this you just need an attitude change. But this ideal shows a direction for further technological development, too. Programming languages must be enhanced to support specifying and not coercing its users to implement. They must be designed seeking the compact, general and precise expression of rules and with its user, the programmer, and not the compiler in mind. Dependencies between code parts must be tracked by the system, not just automatic dependencies but also dependencies the programmer sets manually where the used technology comes to its limits. These developments are of course not restricted to syntax and compilers, run time support is needed and new technologies must arise, such as sound name systems and standard ontologies.

17. Why is Computing Important?

We just thought at first we could use electronic devices to compute numerical expressions, and then we thought we could invent languages to formulate programs, and then we realized that computers are not only good with numbers but they can also record all sorts of information and control all sorts of machines. Then we delved into the information's structure with the relational model and last we generalized the communication with the Internet. Seventy years after the construction of the first electronic computers, computing is ubiquitous and regulates most of our affairs all over the world. If computing has spread so fast, if it has introduced so far-reaching changes in all processes, it must have got the heart of the matter. But of what *matter* actually? Of *civilization*.

The human being is able to speak — that is biology. But once we began writing. That was not an innovation in language, because we wrote at first the same things that we had already been speaking for a long time. But it was an innovation in text. We began constructing larger texts, and these were not blown away by the time, they persisted instead. They were revised and enhanced, and they got better, our knowledge grew. They got fixed, this being useful for regulating society through contracts. Inventing writing, civilization made a great leap forward.

Another step was achieved with the introduction of movable type printing in the 15th century. That was not an innovation in language, that was an innovation in text, too. The hand-writings that already existed were printed. With the use of printing, ideas and knowledge were wide spread, and that changed the world. It led to a new form of civilization based upon open discussion and science.

Computing is an innovation in text, too. Electronic devices store text and carry it. Texts get to be immediately available all over the world. Texts become much larger, moreover the publishing costs shrink. Texts get much sounder, because its coherence is granted without gap. Not only this, computers let us query the text and get some parts of it represented in other ways as they were introduced. That was not possible with hand-writings and printed books. The automated handling of text opens up a lot of possibilities for its evaluation and manipulation.

Computing is not just important, it is crucial. What we make of it will determine the future of the human being. It is not just a tool, it touches the roots of the civilization.