A Multilingual Natural-Language Interface to Regular Expressions

Aarne Ranta

Xerox Research Centre Europe (Grenoble Laboratory) and Academy of Finland.
XRCE, 6 chemin de Maupertuis, 38240 Meylan, France.

Abstract. This report explains a natural-language interface to the formalism of XFST (Xerox Finite State Tool), which is a rich language used for specifying finite state automata and transducers. By using the interface, it is possible to give input to XFST in English and French, as well as to translate formal XFST code into these languages. It is also possible to edit XFST source files and their natural-language equivalents interactively, in parallel.
The interface is based on an abstract syntax of the regular expression language and of a corresponding fragment of natural language. The relations between the different components are defined by compositional interpretation and generation functions, and by corresponding combinatory parsers. This design has been inspired by the logical grammar of Montague. The grammar-driven design makes it easy to extend and to modify the interface, and also to link it with other functionalities such as compiling and semantic reasoning. It is also easy to add new languages to the interface.
Both the grammatical theory and the interface facilities based on it have been implemented in the functional programming language Haskell, which supports a declarative and modular style of programming. Some of the modules developed for the interface have other uses as well: there is a type system of regular expressions, preventing some compiler errors, a denotational semantics in terms of lazy lists, and an extension of the XFST script language by definitions of functions.

1 Introduction

Regular expressions are a mathematical formalism widely used for many computational tasks, ranging from simple search-and-replace procedures to full-scale implementations of grammars of natural languages. While the basic system of regular expressions is simple and widely known, there are lots of variants designed for specific purposes. In this report, we shall consider one particular system of regural expressions, the formalism of XFST (Xerox Finite State Tool). It is a system mainly used for various tasks of linguistic processing. We shall use XFST as the object of a case study of programming and program documentation in natural language. XFST is simple enough to make a fairly complete interface easily manageable. At the same time, it is a system that has lots of occasional and non-programmer users, who might profit from a natural-language interface.

The users of XFST typically write *scripts*, which are programs consisting of regular expressions mixed with some instructions about what to do with them. In some cases, an XFST script is accompanied by some informal description of its content. For instance, the XFST programmer first writes an English text consisting of a sequence of grammatical rules, and then encodes this text rule by rule into the XFST formalism. The text then remains as a perspicuous and reliable document of the script, particularly useful for those who have not themselves written the program but just use it or want to modify it. It is partly in the very purpose of making various intuitive ways of expression possible that the rich formalism of XFST has been developed.

However, there is no systematic guarantee of the script's being accompanied by a corresponding informal text. Scripts are often created from scratch, without providing a full document even afterwards. And even if a document were written in parallel with the script, the two can differ because of human error. They can, of course, also be intentionally different, because the author wants to hide some details of the script.

But let us take it for granted that it is interesting to have natural language texts exactly corresponding to XFST scripts. This is at least an interesting theoretical problem, since it requires a very precise grammar of natural language—so precise that it would be possible to use natural language as a programming language instead of the formalism of XFST. Once such a grammar has been given to one language (to a small but sufficient fragment of it, of course), it is relatively easy to do it for other languages as well. This will immediately lead to the *simultaneous multilingual documentation* of XFST scripts.

Thus we are going to build up a system with the following functionalities:

translating of English and French texts into XFST scripts,
translating of XFST scripts into English and French texts,
translating between English and French texts (via XFST) so that equivalence of meaning is guaranteed,
stepwise editing of XFST scripts, of corresponding English and French texts, and of specialized technical lexica.

We shall argue that the last functionality, stepwise editing, is the most useful one. It produces more comprehensible results than direct translation, and it encourages a structured style of programming. Translating from English and French to XFST is probably not useful as such, because it is difficult for a human writer to stay within the recognized fragment. But it can be used for checking that a text once produced is unambiguous.

2 Background and related work

This study is an instance of the more general idea of an *editor of multilingual semantically precise documents*. This idea has its origin in some linguistic investigations based on the *constructive type theory* of Martin-Löf [8], extending the logical grammar of Montague [9] with some new logical and grammatical structures (see Ranta [11]). Constructive type theory provides a general framework for mathematics and programming, which has been implemented in various systems called *proof editors*, such as ALF from Gothenburg, LEGO from Edinburgh, and Coq from INRIA (see e.g. the volumes 860, 996, and 1158 of Springer *Lecture Notes in Computer Science*). Proof editors are used not only for editing proofs in pure mathematics, but also to construct computer programs and verify their validity with respect to given specifications.

One application of type-theoretical Montague grammar has been to build a natural language interface to a proof editor, making it possible to edit natural language texts in parallel with formalized mathematics, perhaps even without knowledge of the type-theoretical formalism. We did this first for English [12] and then separately for French [13], until it turned out that a suitable kind of *abstract syntax* could be easily made to support several languages in parallel. So there now exists an experimental proof text editor for six languages—English, Finnish, French, German, Italian, and Swedish—partly integrated in the new version of ALF.

The purpose of the present work on the regular expression formalism of XFST is to test the ideas of the proof text editor in a widely used programming environment, which is relatively simple and therefore easily permits a complete and efficient natural-language interface. This interface to XFST should be easy to integrate with the generic proof text editor so that, for instance, one could prove properties of regular expressions and edit XFST code by starting from mathematical specifications.

A natural language interface also exists for the proof editor Coq (see [1]). It is a documentation tool, translating formal proofs into English and French texts, and it is already a part of the official release of Coq. The very idea of using natural language as an interface to formal languages is certainly wide-spread, and extensively used, for instance, in database query dialogues. For regular expressions, there is an experimental translator from English into the formalism of Perl by Kinnunen [7].

Both the grammatical theory and the interface facilities have been implemented in *Haskell*, which is a purely functional programming language. Haskell code consists of definitions of data types and of functions between data types—one can think of it as LISP with types. (See [10] for the standard report and reference manual of Haskell.) A denotational semantics analogous to ours, for a standard system of regular language expressions, is presented in Thompson [14], using the language Miranda very similar to Haskell. He also defines a compiler, a determinizer, and an optimizer. As regular expressions themselves do not form a regular language, the task at hand could not be performed just by an application of finite-state methods. (For the whole task, we used ca. 3000 lines of Haskell code, of which 800 lines for a user interface and 350 lines for denotational semantics not used by the natural language interface.)

3 The regular expressions of XFST

The standard set of regular expressions, originally introduced by Kleene, has just six syntactic forms: the one-symbol expression `c`, the empty string expression `0`, the empty language expression `e`, the concatenation `A B`, the union `A | B`, and the Kleene closure `A*`. These forms have straightforward interpretations as *regular languages*, which are sets of strings, and they can be compiled into *finite state automata*, which are programs that decide whether a given string is an expression of a given regular language. The interpretation as regular languages could be called the *denotational semantics* of regular expressions, and the compilation their *operational semantics*. (See [2], chapters 2 and 3, for standard definitions, methods, and theorems concerning regular expressions.)

The formalism of XFST not only introduces a couple of dozens of new forms of expressions for regular languages, but also expressions for *regular relations*, which are sets not of strings but of pairs of strings. These expressions are compiled into *finite state transducers*, which not only accept and reject input strings, but transform them into other strings. (There is no complete specification of the XFST formalism available, but the tutorial report [5] gives enough background to the present report. Kaplan & Kay [3] and Karttunen [4] are the sources of many fundamental ideas in the use of regular relations.)

The simplest example of a relation expression is the symbol pair `a:b`, where `a` and `b` are symbols. It denotes the singleton relation `[(a,b)]`. Regular relations are closed under concatenation, union, and Kleene closure. So one can write, for instance,

```
[a:b a:b]* | [b:a b:a]*
```

which denotes the relation containing all pairs of strings consisting of equal even numbers of `a`'s and `b`'s. Seen as an operation, this relation transforms any such sequence of `a`'s into a sequence of `b`'s and vice versa.

The notation of XFST is *systematically ambiguous* between language and relation expressions, exploiting the fact that a regular language can be presented by its identity relation, that is, as the relation in which every expression of the language is paired with itself and with nothing else. For instance, the expression `a*` is ambiguous between the set `["","a","aa",...]` and the relation `[("",""),("a","a"),("aa","aa"),...]`.

The ambiguity is often resolved by the context in which an expression is used. For instance, the crossproduct expression `A .x. B` can only denote a relation, and its arguments `A` and `B` can only denote languages. (It also follows that it is not possible to iterate the crossproduct.)

In general, it can always be decided whether an expression is ambiguous. And whenever it is, the ambiguity is between a language and its identity relation. But there must be a level of syntactic description on which language and relation expressions are kept apart, so that one can say, for instance, that the arguments of the crossproduct are languages and the result is a relation, and see that `A .x. [B .x. C]` is not a well-formed expression. (What XFST gives at present is not a syntax error but the message "cannot compile the product" and, in spite of that, a result, which is a transducer consisting of one non-final state. This problem could obviously be avoided by means of a *type system* such as presented in Section 5 below. A type system prevents ill-typed expressions from being sent to the compiler.)

To define denotational semantics, a disambiguating syntax is needed. It will also be needed to produce natural language: we shall use *common nouns* to express regular languages but *instruction sentences* to express regular relations. Thus the expression `a*` is read in English as

optional sequence of 'a"s

when conceived as a language but

repeatedly accept 'a' as such, as long as applicable

when conceived as a relation.

4 Abstract syntax and operations on it

The central role is played by an *abstract syntax*. It is a system of *syntax trees*, whose relation to the *concrete syntax*—that is, the notation visible to the user of XFST or of natural language—is *linearization*, that is, the flattening of tree structure into the linear structure of a string. The inverse operation of linearization is *parsing*.

In the theory of programming languages, it is customary to think of compiling as an operation that is not applied to the visible code as such, but only after parsing. Generally speaking, it is the syntax tree and not the string that is *interpreted* in the denotational semantics of the language and *compiled* (in our case, into a transducer or an automaton).

The structure of the grammar of XFST notation is shown in Figure 1. The system of syntax trees is designed in such a way that they contain all information that is needed for linearization, interpretation, and compilation. Mathematically, each of these operations is a *function* from one system into another. As they usually suppress some of the information present in syntax trees, their inverses are not functions but *search procedures*. In the diagram of Figure 1, as well as in all diagrams to be displayed later, functions are represented by straight arrows and search procedures by bent arrows.

To give an example, the XFST syntax tree

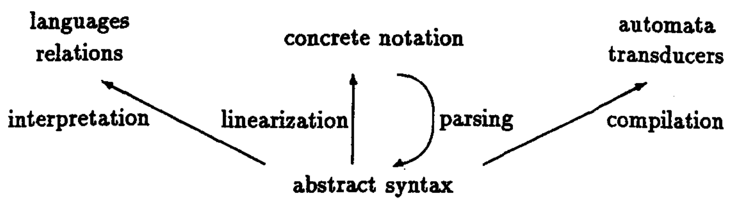

Figure 1. Grammar of XFST notation

```
RLEkleenestar (RLEsymbol (RSEonesymb "a"))
```

gets the following values under the main operations:

linearization: the concrete XFST expression `a*`,
interpretation: the regular set `["","a","aa",..]` (a lazy list of Haskell),
compilation: the automaton `([sb "a"],[0],[0],0,[(0, sb "a", 0)])` (written in the notation of Haskell).

Search procedures can be formalized as functions yielding lists as values, following a technique explained in [15]. For instance, the result of parsing is a list of syntax trees, which can be empty or have several distinct elements. Parsing the concrete regular expression `a*` gives, in addition to the syntax tree just mentioned, the tree

```
RREkleenestar (RREsymbol (RSEonesymb "a"))
```

that represents a relation expression and is compiled to a transducer. Parsing the ill-formed expression `A .x. [B .x. C]` gives an empty list.

The core of the natural-language interface consists of two abstract syntaxes, one for the XFST notation and another for natural language. The communication between the formalism and natural language takes place between the abstract syntaxes, so that all relevant information is preserved. However, natural language is richer than XFST in the sense that it may contain many different expressions for one and the same automaton or transducer (and we intend our fragment of natural language to be much richer in the future than it is now). Thus the operation of *phrasing* that takes an XFST expression into a natural-language expression is a search procudere, while the *interpretation* of XFST in natural language is a function.

Figure 2 shows the communication between XFST and natural languages. The user of the interface, who does not care about why it works, will only see the concrete notations on the top and on the bottom, and the translations between them, which are both search procedures, since at least one component in both of them is a search procedure.

5 Syntactic categories

The systematic ambiguity of XFST notation is resolved by introducing two distinct categories of regular expressions, which are the categories

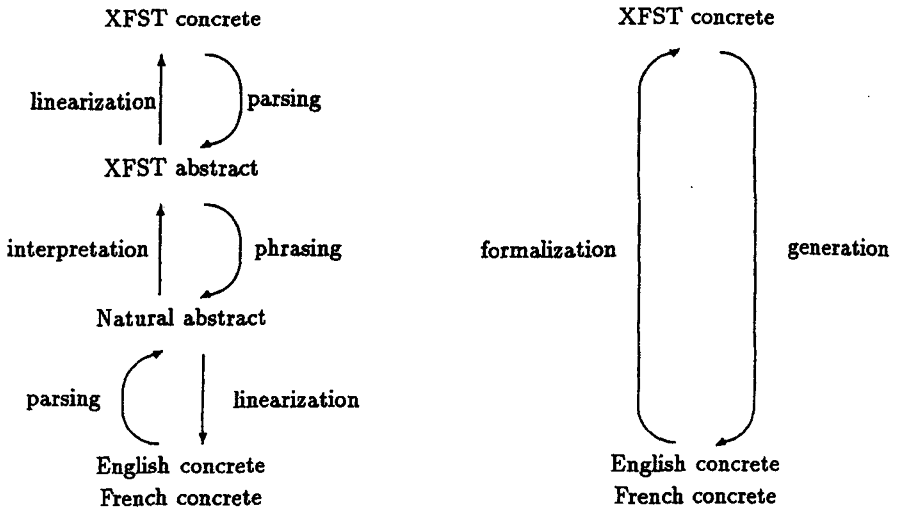

Figure 2. Connections between XFST and natural language (left), the system visible to user (right)

RLE of *regular language expressions*,
RRE of *regular relation expressions*.

In addition, there are some categories not directly visible to the user: RME of *regular match constraint expressions*, RCE of *regular context expressions*, ROE of *regular operation expressions*, and RSE of *regular symbol expressions*. Expressions of these categories occur as parts of language and relation expressions. For instance, match constraint expressions include the arrows -> and @-> denoting the "all matches" constraint and the "left to right longest match" constraint, respectively. What is important in recognizing these categories is that they can be given denotational semantics that works compositionally as a part of the semantics of larger expressions. Thus, for instance, match constraints can be interpreted as functions of pairs of lists of integers encoding segment lengths.

One level higher up than language and relation expressions, we have the category

RDE of *regular definition expressions*.

This category includes expressions of the form

```
define A B ;
```

used in XFST scripts. There are two syntactic structures of this category: definitions of regular languages and definitions of regular relations. Thus, as will be shown later, the categories RLE and RRE are open to the introduction of new expressions by the user of the interface.

In order to define a compositional interpretation of natural language in the XFST formalism, there must be at least one syntactic category of natural language for each category of XFST. We will have, in particular, the categories

RLECN of *regular language common nouns*,
RREI of *regular relation instructions*,
RDES of *regular definition sentences*.

It could be possible to have more categories than these so that, for instance regular languages could also be expressed by adjectives and not only by common nouns. But we shall here confine ourselves to this minimal system.

6 Translations between XFST and natural languages

Rather than showing the syntax trees, their interpretations, and their linearizations in formal detail, we shall just list the XFST operators and some ways of expressing them in English and in French. The list is shown in Table 1. The table presents the expressions grouped into the categories `RLE`, `RRE`, `RME`, `RCE`, and `RDE`. The category `RCE` is included in `RLE`, and the category `ROE` in `RRE`.

The table shows just one natural-language structure for each XFST form of expression. A few more are already implemented in the interface, and anyone who plays with it will almost immediately suggest some new ones. But there are some requirements that any new syntactic structure must fulfill—if some of the expressions included in the table looks more complicated than one would expect, this is usually explained by some of the following three principles:

> Expressions belonging to the same category must have the same syntactic behaviour. (This rules out having, say, adjectives in the same category as common nouns.)
> The constructions must be arbitrarily iterable. (Just as the operators of XFST are. The result is often hard to read but it should always be grammatical.)
> No expression should be ambiguous. (This is not necessary for an interface, but it makes it simpler. The language gets more complicated, though, because special words are needed to function as parentheses.)

In Section 9, we will explain an extension of the XFST script language that makes it possible to introduce new ways of expressing regular expression operators.

The presentation of natural language expressions in the table is schematic and does not make explicit the way in which various morphological features, such as those imposed by agreement, are controlled. There is a detailed discussion of this topic in [13]. All morphological features are introduced in linearization, and so is the order of words: they do not belong to the abstract syntax.

All of the structure captured by the abstract syntax is common to the different languages. Notice that there is more in common than just the semantical content, since the same content can be expressed in different ways. The tiny fragment presented here does not yet give a very good illustration of this phenomenon. But a little example can already be given. The regular language

```
a | b | c
```

is expressed, according to the table, by the English and French common nouns

string equal either to 'a' or to 'b' or to 'c',
chaîne égale soit à 'a' soit à 'b' soit à 'c'.

But the grammar also includes the more concise structure usable for a union all of whose members are single symbols:

XFST	English	French
c (symbol)	'c'	'c'
{abc} (word)	"abc"	"abc"
0	*empty string*	*chaîne vide*
.#.	*word boundary*	*limite de mots*
?	*symbol*	*symbole*
\A	*symbol other than an* A	*symbole autre qu'un* A
A B ... C	*string beginning with an* A *and continuing by a* B *...followed by a* C	*chaîne commençant par un* A *et continuant par un* B *... suivi d'un* C
A | B...C	*string equal either to an* A *or to a* B *... or to a* C	*chaîne égale soit à un* A *soit à un* B *... soit à un* C
A*	*optional sequence of* A*'s*	*séquence optionnelle de* A*'s*
A+	*nonempty sequence of* A*'s*	*séquence non vide de* A*'s*
$A	*string containing an* A	*chaîne contenant un* A
A - B	*other* A *than a* B	*autre* A *qu'un* B
A & B	*string equal both to an* A *and to a* B	*chaîne égale et à un* A *et à un* B
~A	*string other than an* A	*chaîne autre qu'un* A
A^n	*sequence of n* A*'s*	*séquence de n* A*'s*
(A)	*optional* A	A *optionnel*
A & B	*string resulting from an* A *by inserting* B*'s*	*chaîne résultant d'un* A *par l'insertion de* B*'s*
A => G	*string containing an* A *only* G	*chaîne ne contenant de* A *que* G
A (ident. rel.)	*accept* A *as such*	*accepter* A *tel quel*
c:d	*change* c *into* d	*changer* c *pour* d
R Q ... P	*in the beginning* R *then* Q *... then* P	*au début* R, *ensuite* Q, *... ensuite* P
R | Q...P	*not only* R *but also* Q	*non seulement* R *mais aussi* Q
R*	*repeatedly* R *as long as applicable*	*faisant répétition,* R *aussi longtemps qu'applicable*
R+	*repeatedly* R *as long as applicable but at least once*	*faisant répétition,* R *aussi longtemps qu'applicable mais au moins une fois*
R^n	*repeatedly* R *n times*	*faisant répétition,* R *n fois*
(R)	*optionally* R	*optionnellement,* R
A .x. B	*replace an* A *by a* B	*remplacer un* A *par un* B
R .o. Q...P	*first* R *and, in what results,* Q, *... and, in what results,* P	*d'abord* R *et, dans ce qui en résulte,* Q *... et, dans ce qui en résulte,* P
A x-> B	*replace every* A *by a* B, x->	*remplacer tout* A *par un* B, x->
A x-> L...R	*mark the beginning of every* A *by an* L *and the end by an* R, x->	*marquez le commencement de tout* A *par un* L *et la fin par un* R, x->
->	*choosing all possible matches*	*choisissant toutes les apparitions possibles*
@->	*choosing the longest matches from left to right*	*choisissant les apparitions les plus longues de gauche à droite*
L _	*if it is preceded by an* L	*s'il est précédé par un* L
_ R	*if it is followed by an* R	*s'il est suivi d'un* R
define A B	*An* A *is a* B.	*Un* A *est un* B.
define R Q	*To* R *is to* Q.	R, *c'est* Q.

Table 1. Regular expressions in XFST, English, and French.

symbol from the list 'a', 'b', 'c'
symbole de la liste 'a', 'b', 'c'.

The distinction between these two grammatical structures is similar in English and French, but it is not reflected by any distinction on the semantic level, that is, in the XFST formalism.

7 Translating regular expressions

Given the theoretical framework of Figure 2 (left), it is possible to build several functionalities of translating between XFST, English, and French and of editing XFST scripts and English and French text files. For instance, if we have an interface implementing translation from XFST to natural language, we can type in the string

```
[a | b]+
```

and get the following output (actually as LaTeX code, here typeset):

English expressions for language :
nonempty sequence of symbols from the list 'a', 'b'
French expressions for language :
séquence non vide de symboles de la liste 'a', 'b'
English expressions for relation :
accept a nonempty sequence of symbols from the list 'a', 'b' as such
repeatedly accept a symbol from the list 'a', 'b' as such, as long as applicable but at least once
repeatedly not only accept 'a' as such but also accept 'b' as such, as long as applicable but at least once
French expressions for relation :
accepter une séquence non vide de symboles de la liste 'a', 'b' telle quelle
faisant répétition, accepter un symbole de la liste 'a', 'b' tel quel aussi longtemps qu'applicable mais au moins une fois
faisant répétition, non seulement accepter 'a' tel quel mais aussi accepter 'b' tel quel aussi longtemps qu'applicable mais au moins une fois

Because the input expression is ambiguous between a language and a relation, it can be expressed both as a common noun and as an instruction. The instruction, in turn, is either based on the identity relation of `[a | b]+` (the first sentence), on the Kleene closure of the identity relation of `[a | b]` (the second sentence), or on the disjunction of the identity relations of `a` and `b` (the third sentence). For the practical purpose of documentation, it seems that it is the user who knows best which alternative to choose.

Translation from English or French to XFST is unambiguous because our English and French fragments distinguish between language and relation expressions. The parser allows quite a lot of errors in natural language input: since morphology and orthography have been precisely defined in linearization, it is only easier, as well as more useful, to make the parser tolerant. Thus the French input

accepter un sequence optionnel de chaines vide tel quels

both yields the result `0*` and the corrected translation back to French,

accepter une séquence optionnelle de chaînes vides telle quelle

8 Editing scripts and text files

While many grammatical constructions of formal and natural languages can be arbitrarily iterated, even without ambiguities, the results can get hard to read. Thus the definition of a simplified Finnish hyphenation program reads in the XFST notation

```
define hyphenate
[ [d | g | h | j | k | l | m | n | p | r | s | t | v]*
[a | e | i | o | u | y]+
[d | g | h | j | k | l | m | n | p | r | s | t | v]*
@-> ... %- || _ [d | g | h | j | k | l | m | n | p | r | s | t | v]*
[a | e | i | o | u | y]+] ;
```

which is already hard to read, but probably easier than the English version produced by our interface (the French version is no better):

> To hyphenate is to mark the end of every string that begins with an optional sequence of symbols from the list 'd', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'r', 's', 't', 'v' and continues by a nonempty sequence of symbols from the list 'a', 'e', 'i', 'o', 'u', 'y' followed by an optional sequence of symbols from the list 'd', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'r', 's', 't', 'v' by '-' if it is followed by a string that begins with an optional sequence of symbols from the list 'd', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'r', 's', 't', 'v' and continues by a nonempty sequence of symbols from the list 'a', 'e', 'i', 'o', 'u', 'y', choosing the longest matches from left to right.

Both the formal code and the corresponding English and French texts are easier to understand if organized in sequences of shorter definitions:

```
define vowel a | e | i | o | u | y ;
define consonant d | g | h | j | k | l | m | n | p | r | s | t | v ;
define syllable consonant* vowel+ consonant* ;
define hyphenate syllable @-> ... %- || _ consonant vowel ;
```

> A vowel is a symbol from the list 'a', 'e', 'i', 'o', 'u', 'y'.
> A consonant is a symbol from the list 'd', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'r', 's', 't', 'v'.
> A syllable is a string that begins with an optional sequence of consonants and continues by a nonempty sequence of vowels followed by an optional sequence of consonants.
> To hyphenate is to mark the end of every syllable by '-' if it is followed by a string that begins with a consonant and continues by a vowel, choosing the longest matches from left to right.

> Une voyelle est un symbole de la liste 'a', 'e', 'i', 'o', 'u', 'y'.
> Une consonne est un symbole de la liste 'd', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'r', 's', 't', 'v'.
> Un syllabe est une chaîne qui commence par une séquence optionnelle de consonnes et continue par une séquence non vide de voyelles suivie d'une séquence optionnelle de consonnes.
> Marquer les syllabes, c'est marquer la fin de tout syllabe par '-' s'il est suivi d'une chaîne qui commence par une consonne et continue par une voyelle, choisissant les apparitions les plus longues de gauche à droite.

(The example is from [6], structured in a slightly different way.)

When editing a natural language text in parallel with an XFST script, some lexical information has to be given: the word or words used for each new defined concept, the plural form (if irregular), and the gender (in French). The meaning of each lexical entry is given by the definition itself. Thus the words that are introduced are used in very precise technical meanings.

9 Function definitions

Standard XFST scripts have a format for defining macros for *constant* regular expressions, but we can get much more structure by defining *functions*. For function definitions, we use the format

```
define F(.X,...,Y.) C,
```

where C is an already defined regular expression possibly containing the variable symbols X, ..., Y (these symbols of course cannot be used as names of these letters in C—but we need not reserve a special class of variable symbols). A file containing function definitions can be translated into a file without them by replacing all applications of functions by their definienda. Those functions that are not used in definitions of constants are then simply ignored.

Function definitions can be used for introducing new operators. For example, the task of shallow parsing uses the same kind of mark up operation over and over again: a segment of a string is put between parentheses and the closing parenthesis is marked by a category label. Using a function definition, we can write

```
define labelWith(.C,c.) C @-> %( ... %) c ;
define markNP labelWith(.NP,%+np.) ;
define markVP labelWith(.VP,%+vp.) ;
define markS  labelWith(.S, %+s .) ;
```

where NP, VP, and S are some previously defined sets of noun phrases, verb phrases, and sentences, respectively. Now, the natural-language structure corresponding to functions is an expression with *complements*, and it is easy to include user-defined information on the complements in the grammar. This information includes the prepositions (possibly none) required by each argument place, as well as the question whether the complement takes the plural or the singular form. The English text corresponding to the above piece of script looks as follows:

> To label *C*'s with a *c* is to mark the beginning of every *C* by '(' and the end by a string that begins with ')' and continues by a *c*, choosing the longest matches from left to right.
> To mark noun phrases is to label noun phrases with '+np'.
> To mark verb phrases is to label verb phrases with '+vp'.
> To mark sentences is to label sentences with '+s'.

Without the initial function definition, we would need three sentences of the same length and complexity as the function definition.

10 The importance of structured writing

Organizing a program into a sequence of definitions is an example of *structured programming*, whose benefits are well known among programmers, but which is even more important if programs are systematically translated into texts. The impact of structuration on readability is, so to say, magnified when formal code is translated into the less perspicuous syntax of natural language. In natural language, syntactically complex expressions *must* be avoided by careful planning of the text.

An obvious question arises whether it is possible to take a messy text and make it more readable by some automatic structuration software. The answer suggested by the analogy between text-production and programming is negative: as there is no algorithm that turns messy programs into structured ones, there is no algorithm that turns messy texts into readable ones. But programmers and writers can be encouraged to structured thinking.

Readability is not so much a function of the language that is chosen but of the way in which the chosen language is used. There may be programming languages in which it is impossible to produce readable code, but unreadable code can be produced in any language, be it formal or natural. Thus any natural-language interface should be judged, not by the worst expressions it includes (because every language includes bad expressions), nor by its coverage of natural language (which is surely limited), but by its ability to provide clear and natural ways of expression whenever properly used.

References

1. Yann Coscoy, Gilles Kahn, and Laurent Théry, 1995. "Extracting text from proofs". In *Typed Lambda Calculus and Applications, Lecture Notes in Computer Science* 902, Springer, Heidelberg.
2. John E. Hopcroft & Jeffrey D. Ullman, 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading (Ma.).
3. Ronald M. Kaplan & Martin Kay, 1994. "Regular Models of Phonological Rule Systems", *Computational Linguistics* 20, pp. 331–380.
4. Lauri Karttunen, 1996. "Directed Replacement", In *Proceedings of ACL-96*, Santa Cruz, California.
5. Lauri Karttunen, 1997. "Syntax and Semantics of Regular Expressions", Web documentation, Xerox Corporation.
6. Lauri Karttunen, 1997b. "Examples of Networks and Regular Expressions". Web documentation, Xerox Corporation.
7. Matti Kinnunen, 1997. "A natural-language interface to regular expressions". Talk given at the Second Conference on Logical Aspects of Computational Linguistics in Nancy.
8. Per Martin-Löf, 1984. *Intuitionistic Type Theory*, Bibliopolis, Naples.
9. Richard Montague, 1974. *Formal Philosophy*, Yale U.P., New Haven.
10. John Peterson & Kevin Hammond *eds.*, 1997. *Report on the Programming Language Haskell. A Non-strict, Purely Functional Language*, Yale University. Available through `http://haskell.org/`.
11. Aarne Ranta, 1994. *Type Theoretical Grammar*, Oxford University Press, Oxford, 1994.
12. Aarne Ranta, 1996. "Context-relative syntactic categories and the formalization of mathematical text". In S. Berardi and M. Coppo, eds., *Types for Proofs and Programs*, pp. 231–248, *Lecture Notes in Computer Science* 1158. Springer, Heidelberg.
13. Aarne Ranta, 1997. "Structures grammaticales dans le français mathématique". In *Mathématiques, informatique et Sciences Humaines.*, vol. 138 pp. 5–56 & vol. 139 pp. 5–36.
14. Simon Thompson, 1995. "Regular Expressions and Automata using Miranda". University of Kent. Article available in `http://www.cs.ukc.ac.uk/pubs/1995/212/index.html`.
15. Philip Wadler, 1985. "How to replace failure by a list of successes". In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, pp. 113–128, *Lecture Notes in Computer Science* 201. Springer, Heidelberg.