Journal of Computer and System Sciences **58**, 453–482 (1999)
Article ID jcss.1999.1623, available online at http://www.idealibrary.com on IDEAL®

Cut and Paste

DIFA, Università della Basilicata, Via della Tecnica, 3, 85100 Potenza, Italy
E-mail: mecca@dia.uniroma3.it

and

Paolo Atzeni[†]

Dipartimento di Informatica e Automazione, Università di Roma Tre, Via della Vasca Navale, 84, 00146 Rome, Italy
E-mail: atzeni@dia.uniroma3.it

Received December 15, 1997; revised September 30, 1998

The paper develops EDITOR, a language for manipulating semistructured documents, such as those typically available on the Web. EDITOR programs are based on two simple ideas, taken from text editors: "search" instructions are used to select regions of interest in a document, and "cut & paste" instructions to restructure them. We study the expressive power and the complexity of these programs. We show that they are computationally complete, in the sense that any computable document restructuring can be expressed in EDITOR. We also study the complexity of a safe subclass of programs, showing that it captures exactly the class of polynomial-time restructurings. The language has been implemented in Java and is currently used in the ARANEUS project as a basis for a wrapper-generation toolkit.

1. INTRODUCTION

It is well known that databases provide a robust technology for querying highly structured data in a flexible and efficient way. Recently, the manipulation of less structured information has also become a field of great interest. This is especially due to the explosion of the World Wide Web [11], which is essentially a large collection of distributed documents organized as a hypertext. Extending database techniques to Web documents poses a number of new challenges. First, tools are needed to explore the huge graph of pages and locate data of interest [29, 30, 36]. Then, once documents have been downloaded, a key problem consists in identifying relevant pieces of information inside text and extracting them in order to build a database representation in some data model, which can then be manipulated

* http://www.difa.unibas.it/users/gmecca.
[†] http://www.dia.uniroma3.it/~atzeni.

0022-0000/99 $30.00

by a query language. This process is often referred to as a *wrapping* of a data source [15]. In this paper we focus on this latter aspect: we address the manipulation of textual data and propose a language, called EDITOR, for searching and extracting regions in a document. EDITOR programs are based on two simple ideas, borrowed from text editors. In fact, editors provide a natural way of interacting with a document: when the user needs to restructure a document, *search* primitives can be used to localize regions of interest inside text, and *cut & paste* operations to move regions around. Likewise, in EDITOR, search instructions are used to select regions in a document, and cut & paste to restructure them.

1.1. The Framework

The fact that the Web organizes information in documents has somehow caused a shift of perspective on the way data can be accessed and manipulated, so that traditional database query languages are not well suited to this new framework. Thus, the design and implementation of new languages capable of manipulating documents are becoming important research fields.

Indeed, incorporating text into databases is not a new idea [22, 32]. In fact, in many cases, textual documents are organized according to a precise structure, usually described using a formal grammar. For example, SGML [27] is a well-known formalism for describing structured documents. The fact that parsed documents closely resemble database structures has been used to propose extensions of query languages capable of manipulating text. However, all of these proposals are based on a form of grammatical preprocessing of textual information (see [4, 12, 16, 18, 23, 28, 40]). In essence, when the textual database is created, each document is preprocessed to generate suitable data structures, such as the parse tree associated with the document string or indexes used to access important regions in the text. Then, query processing is based on these structures.

When applied to the Web context, traditional query techniques for textual data show their limitations; in fact, in many cases, grammars are not flexible enough to capture the organization of semistructured documents. This depends on several factors: on the one hand, the structure is often incomplete or rather implicit; on the other hand, documents may present heterogeneities and exceptions or even *errors*. In fact, browsers do not parse the HTML sources they access, and they attempt to display the corresponding page even in the presence of errors. As a consequence, it is rather frequent that HTML pages do not fully comply with HTML grammar rules. For all these reasons, more flexible means of analyzing text are needed. Also, *restructuring* plays an important role here, whereas traditional approaches mainly focus on searches. Intuitively, restructuring means moving regions of the document around, adding new ones, and perhaps deleting some. However, it is not clear how the user should specify the restructuring to be performed and how new documents —or database structures—must be created starting from existing ones. In essence, document restructuring can be considered a view definition mechanism to define *derived* structures, which seem to be very important in this context.

Our goal in this paper is to introduce a flexible and expressive formalism for searching and restructuring documents. At the same time, we attempt to meet

another fundamental requirement, i.e., to study its computational properties. This is particularly interesting for two reasons: first, to gain a better understanding of what it means to compute on the Web, as attempted by [5, 37]; second, to efficiently implement document manipulation primitives, and tune the complexity of wrapping. There are, indeed, some procedural languages (such as Python [2] or Perl [41]) extended with powerful text management features; however, it would be difficult, if at all possible, to study the complexity of such languages. We therefore introduce a formal computational model for documents, based on an abstract EDITOR machine, and a language with a small and yet expressive number of primitives for managing text.

1.2. Overview of EDITOR

EDITOR is a language for searching and restructuring documents. It is based on a simple model of computation, which involves the basic operations of text editors. Each program can access a set of documents, considered simply as strings of symbols over a finite alphabet. *Regions*, i.e., contiguous substrings in documents, can be *selected* using *search instructions* and modified using *replace instructions*. A *clipboard* is associated with each program; restructurings can be performed using *cut*, *copy*, and *paste* instructions, which make use of the clipboard.

The search process is based on the use of simple *patterns*, made of constant symbols, such as `a`, `b`, `c`, ..., taken from the alphabet, plus a special symbol, `*`, called the *wild card*. Examples of patterns are `abc*d*e` and `<TITLE>*</TITLE>`. Patterns are matched against documents in a natural way: each alphabet symbol matches itself, and wild cards match any string. When a document is searched for strings matching a pattern, in the case of alternatives, the *leftmost match* is selected. For example, given a pattern `a*b` and a string `ccaabb`, the chosen match is `aab`.

An important concept in EDITOR is that of *current selection*. In fact, as is common in text editors, cut & paste operations on a document implicitly refer to the currently selected region in the text. In our approach, a *region* is a document portion delimited by two positions.[1] Regions in documents are selected using search instructions: whenever a search instruction is executed on a document and a pattern is specified, the leftmost substring of the document matching the pattern is *selected*. Then, clipboard operations on that document implicitly refer to the currently selected region. Thus, to cut or copy a specific region from some document, the user first has to select the region using a search instruction with the appropriate pattern.

EXAMPLE 1.1 [Title]. Given an HTML document `HMTLPage`, the following program is used to copy the title of `HTMLPage` onto a new document called `Title`:

```
search(HTMLPage, "<TITLE>*</TITLE>");
copy(HTMLPage);
paste(Title);
```

[1] Delimiting symbols are not included in the region. Thus, two consecutive positions delimit an empty region.

The first instruction searches the page, looking for the first region starting with the string <TITLE> and ending with the string </TITLE>. The wild card, *, is used to match any string contained between the tags. When the instruction is executed, HTMLPage is searched and the title region is selected. Then, the second instruction copies the selected region to the clipboard, and the third instruction pastes the clipboard content to Title.

Selections are very important in EDITOR, since they also act as place-holders; in fact, in the course of a computation, we keep track of the portions of documents already examined, and each search starts from the *current position*, i.e., the first position following the current selection. If the search succeeds, the new region is selected. If the end of the document is reached, the search fails and the empty region following the last document symbol is selected. Then, as it is common in text editors, subsequent searches start again from the first symbol, that is, the document is searched in a circular way.

The content of a document can be changed using replace instructions that replace occurrences of a constant string, S_1, with occurrences of another string, S_2; when a replacement has to be made, the first (leftmost) occurrence of S_1 is searched in the document, starting from the current position. If an occurrence is found, it is replaced with S_2, which becomes the new selected region. Otherwise, the replacement fails and the empty region following the last document symbol is selected. Note that, also in this case, the document is searched in a circular way; i.e., a subsequent search or replace will examine the document starting from the top. As a special case, a replace instruction in which S_1 is the empty string, ε, simply inserts the string S_2 into the document right before the current position.

EXAMPLE 1.2 [Changing the Title]. Suppose we are given an HTML document, HTMLPage, and want to change its title; we can use a combination of cut and replace instructions, as follows:

```
search(HTMLPage, "<TITLE>*</TITLE>");
cut(HTMLPage);
replace(HTMLPage, ε, "<TITLE>My Title</TITLE>");
```

The program searches the HTML source to locate the title and cuts it to the clipboard; in this way, the title is removed from the document and the current position coincides with the very next character. Then, a replace instruction is used to insert the new title, "My Title"; since the string to replace is the empty string, the new title is inserted right before the current position.

Searches in a document can be iterated using *loop* instructions, which have the form

```
loop search(D, Pat)
  Body
end loop
```

where *Body* is a sequence of instructions. Loops act in a natural way: when the loop is executed, the first (i.e., leftmost) occurrence of pattern *Pat* is searched in document *D*, starting from the top. If the search fails, that is, no occurrence of the pattern is found, the body is not executed. Otherwise, the body is executed as long as occurrences of the pattern are found in *D*. When, during the search, the document end is reached, the search fails and the loop terminates.

EXAMPLE 1.3 [Table of Contents]. Suppose we are given an HTML document, `HTMLPage`, and want to generate its table of contents in a new document, `ToC`. We can do this by selecting each first-level header from `HTMLPage` and pasting it to `ToC`, in the following way:

```
loop search(HTMLPage, "<H1>*</H1>")
    copy(HTMLPage);
    paste(ToC);
end loop;
```

In this program, the loop is used to iterate the body until the end of document `HTMLPage` is reached. When the loop starts, `HTMLPage` is searched for the first region that begins with a tag of the form `<H1>` and ends with a tag of the form `</H1>`. The loop body simply copies the header (along with the tags) to the output document, `ToC`. Then, the loop end is reached and `HTMLPage` is searched again. Now, the search starts from the position following the selected header, so that the next header (if any) is selected and the body is executed again. After the last header has been found, the end of `HTMLPage` is reached, the search fails, and the loop terminates.

The previous examples illustrate the basic ideas of the formalism. We now give a more complex example, showing how EDITOR can be effectively used to wrap HTML pages and build database structures.

EXAMPLE 1.4 [Wrapping Pages]. Consider the page in Fig. 1. It shows a list of paintings in the Capodimonte Museum in Naples [1]. For each painting, the title and the painter, plus a link to the corresponding painting page, are reported. We now want to extract information from the HTML source and generate a table of tuples with two attributes, *title* and *author*, one for each painting in the page. To do this, we note that items in the list have the form

$$\texttt{<LI><A HREF="}\textit{ref}\texttt{">}\ \textit{title}\texttt{</A>(}\textit{author}\texttt{)},$$

where *ref* is the URL of the painting page, *title* is the painting title, and *author* is the name of the author; tag `<LI>` is used in HTML to denote one item in a list, whereas `<A HREF="`*ref*`">` *title* `</A>` indicates to the browser that, by clicking on the painting title, the page with URL *ref* is to be accessed. Our objective is to build a table of rows of the form `[`*title author*`]`, in which the two values are separated by a tabulator, denoted by `\t`. For example, given the input document `PaintList`

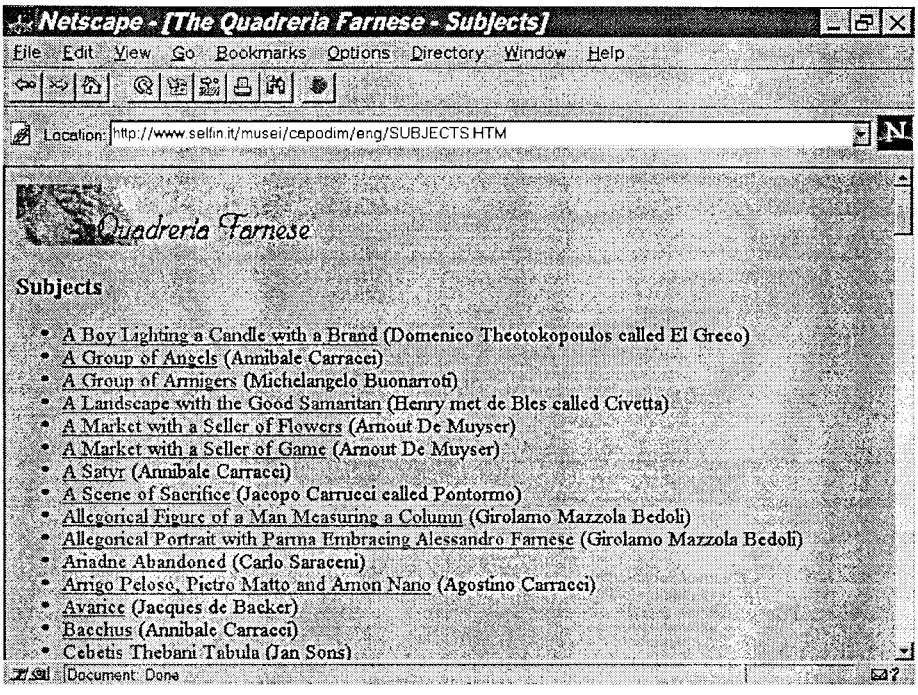

FIG. 1. A list of paintings in the Capodimonte Museum.

corresponding to the HTML source of the page in Fig. 1, we would like to generate the document partially shown in Fig. 2. In the document, square brackets, `[`, `]`, are used as tuple delimiters; the first row contains the attribute names. The program in Fig. 3 performs this task, returning the output in document `Table`.

The major part of the program consists in the loop, which searches `PaintList` for list items (lines `3`–`18`). Each item is processed as follows: (i) the whole item is copied to the clipboard and pasted to the temporary document, `Temp` (lines `4`, `5`); (ii) the title is extracted—by eliminating everything up to the symbol > (lines `7`, `8`), and then by selecting the string up to tag </A> (lines `9`, `10`)—and pasted to the output document `Table` (line `11`); the final tag is replaced with a tabulator (line `12`); (iii) then, with a similar management of details, the author is selected in `Temp` (lines `13`, `14`), pasted to `Table` (line `15`), and cleaned (lines `16`, `17`).

```
Title                                  Author
-----                                  ------
[A Boy Lighting a Candle with a Brand   Domenico Theotokopoulos called El Greco]
[A Group of Angels                     Annibale Carracci]
[A Group of Armigers                   Michelangelo Buonarroti]
[A Landscape with the Good Samaritan   Henry met de Bles called Civetta]
[A Market with a Seller of Flowers     Arnout De Muyser]
...                                    ...
[Bacchus                               Annibale Carracci]
[Cebetis Thebani Tabula                Jan Sons]
...                                    ...
[Wrath                                 Jacques de Backer]
```

FIG. 2. A document containing a relation.

```
1   replace(Table, ϵ, "Title \t Author \n"); Writes the first line in document Table;
2   replace(Table, ϵ, "----- \t ------ \n"); writes the second line in document Table;
3   loop search(PaintList, "<LI>*)" )        searches for list items iteratively;
4      copy(PaintList);                      copies the current item to the clipboard
5      paste(Temp);                          and pastes it to document Temp;
6      replace(Table, ϵ, "[" );              writes the left parenthesis in Table;
7      search(Temp, "<LI>*>");               eliminates from Temp
8      cut(Temp);                            the string preceding the title;
9      search(Temp, "*</A>");                selects the painting title,
10     cut(Temp);                            cuts it to the clipboard
11     paste(Table);                         and pastes it to Table;
12     replace(Table, "</A>", "\t");         replaces the final tag with a tabulator;
13     search(Temp, "*)");                   selects the rest of the item in Temp
14     cut(Temp);                            removes it from Temp
15     paste(Table);                         and pastes it to Table;
16     replace(Table, "(", ϵ);               removes the left parenthesis
17     replace(Table, ")", "]\n");           and replaces the right one with ]
18  end loop;                                followed by the new line and loops.
```

FIG. 3. EDITOR program for restructuring authors and titles.

We denote with `ListTable(PaintList)` the document produced by the program on input document `PaintList`. Since `ListTable(PaintList)` is essentially a relation with attributes `Title` and `Author`, we can think of importing it in a DBMS and querying it using SQL, for example, to know the titles of all paintings by Michelangelo, as follows:

```
SELECT  Title
FROM    ListTable(PaintList)
WHERE   Author = 'Michelangelo Buonarroti'
```

As can be seen from this example, the process of extracting a region of interest from a document usually starts with a search that selects some larger region and then progressively refines it by successive searches. This may in some cases yield rather involved programs containing long sequences of search instructions. It could be possible to shorten these pieces of code by introducing a syntactic sugar that allows us to "name" portions of a pattern inside a search using variables in a way which is common in pattern-matching languages; for example, with respect to this painting example, consider the following sequence of instructions:

```
search(PaintList, "<LI*)");
copy(PaintList);
paste(Temp);
search(Temp, "<LI*>");
copy(Temp);
```

```
paste(x);
search(Temp, "*/A>");
copy(Temp);
paste(y);
search(Temp, "(*)");
copy(Temp);
paste(z);
```

As can be seen, the result of the first search is further split into three parts pasted to documents `x`, `y`, `z` respectively. To considerably reduce the length of the code one might write

```
search(PaintList, "<LI*> : x "*</A>" : y "(*)" : z);
```

With this syntax we mean that as soon as a portion of the pattern is matched against a document region, the latter is immediately pasted into a document named as the corresponding variable.

Although very compact, these kinds of instructions introduce some subtleties into the definition of the syntax and semantics of the language. To simplify the treatment, in the following we will only use the base syntax without variables.

1.3. Contributions of the Paper

EDITOR is a new formalism for manipulating semistructured data. In this paper, we study its computational properties. Although very simple, the language has considerable expressive power. In fact, it can express a wide class of document restructurings. In our approach, a *document* is considered a string of symbols over a fixed alphabet, and a *restructuring* is a mapping that associates an output document with a set of input documents. In the paper we develop an automata-theoretic algorithm for searching occurrences of our patterns in documents and selecting the corresponding regions, based on the Aho–Corasick [7] algorithm for finding occurrences of a string in a text; unlike general regular expressions, the deterministic finite state automaton associated with the pattern can be generated in linear time, so that patterns can be efficiently matched against documents. Then, we study the expressive power and the computational complexity of EDITOR programs. We concentrate on a specific class of programs, those without nested loops, and show that they are very expressive; in fact, they are computationally complete, in the sense that any computable restructuring can be expressed using these programs. Note that we achieve completeness despite the fact that loops are not nested and that no explicit conditional instruction is present.

Then, we study the complexity of these programs; we impose a natural restriction that guarantees safe computations, and we show that the resulting class of programs captures exactly the class of feasible restructurings, i.e. restructurings computable in polynomial time. The PTIME-expressibility result represents an interesting contribution.

In fact, EDITOR programs can be considered a string counterpart of *loop programs* [26, 38], a formalism for computing over integers. However, the complexity of loop programs jumps from linear time to exponential time, so that, unlike EDITOR programs, there is no known restriction of loop programs capturing polynomial time computations.

We have implemented an extension of EDITOR as a set of Java classes for managing documents.[2] We effectively use the formalism in the ARANEUS project [8, 9, 35] as a basis for a two-way view definition process: on one hand, we wrap HTML pages to build relational views over a Web site, which can then be queried using any database query language; on the other hand, we define new Web pages from data in the database, thus building *derived sites*. In our experiments, the formalism has proven to represent a natural and effective means to reason and compute about documents. Far from being a limitation, the procedural nature of the language makes it a flexible tool in analyzing semistructured data, capable of dealing with partial or implicit structures, and even coping with errors in the text. In fact, we believe that the cut & paste approach EDITOR is based upon can be extended to the manipulation of other kinds of data. For example, we plan to extend the language in order to manipulate multimedia data, in the spirit of [33].

The rest of the paper is organized as follows. Sections 2, 3, and 4 are devoted to the formal development of EDITOR; they formalize the computational model, the syntax, and the semantics, respectively. Section 5 studies the expressive power of the formalism and Section 6 establishes the complexity results. Finally, in Section 7 we show how the language can be used as a programming language for the Web.

2. EDITOR PROGRAMS AND EDITOR MACHINES

In this section we develop the syntax and semantics of EDITOR. As a preliminary step, we formalize the model of computation that EDITOR programs are based upon, called an *editor machine*.

We fix an *alphabet* of symbols, Σ, and some special symbols, not contained in Σ: $\triangleright$, the *left* (*or top*) *delimiter*; $\triangleleft$, the *right* (*or bottom*) *delimiter*; and $*$, the *wildcard*. We also fix a set of *document names*, used to denote documents. A *document* is a finite string over Σ; when manipulated in our framework, the document string is considered preceded by the top delimiter, $\triangleright$, and followed by the bottom delimiter, $\triangleleft$. Delimiters act essentially as start- and end-of-file characters. An *editor machine* works on a simple data structure: a finite set of documents, plus a special document, called the *clipboard*, initially empty. The computation of the machine consists of the execution of an EDITOR *program*, that is, a sequence of *editor instructions*. During the computation, each document has a state, consisting of its *selected region* (or *current selection*), which also determines the *current position* in the document, as follows.

DEFINITION 1 [Regions and Positions]. To each document $D = \triangleright a_1 a_2 \cdots a_n \triangleleft$ of length n there correspond numeric *positions* from 0 and $n+1$; the top delimiter,

[2] The implemented version is an extension of the language described in this paper; for example, it also provides an if–then–else instruction that can be used to test the result of a search.

$\triangleright$, has position 0; the bottom delimiter, $\triangleleft$, has position $n+1$. A *region* is a pair of positions (i, j) with $0 \leqslant i < j \leqslant n+1$; i and j are called the *left* and *right delimiters* of the region, respectively. To each region (i, j) we can associate a substring of the document, namely $a_{i+1} \cdots a_{j-1}$. Regions of the form $(i, i+1)$ denote empty substrings.

In the following, we blur the distinction between a region and the corresponding document substring. In each document at any step of the computation there is a *selected region*. Initially, the *selected region* of each document is $(0, 1)$. It then changes as an effect of executing instructions. The current selection also serves as a placeholder; in fact, if the current selection is (i, j), the *current position* coincides with j; that is, it corresponds to the first position following the selected region.

EXAMPLE 2.1 [Selections and Positions]. Consider document $\triangleright$`abcdef`$\triangleleft$. Here is a list of selected regions with the associated positions.

Curr. selection	Sel. region	Curr. position
$\triangleright$`a`⌈`bcde`⌉`f`$\triangleleft$	$(1, 6) =$ `bcde`	`6 (f)`
$\triangleright$`abcde`⌈`f`⌉$\triangleleft$	$(5, 7) =$ `f`	`7 (`$\triangleleft$`)`
$\triangleright$‖`abcdef`$\triangleleft$	$(0, 1) = \varepsilon$	`1 (a)`
$\triangleright$`abcd`‖`ef`$\triangleleft$	$(4, 5) = \varepsilon$	`5 (e)`

The following section introduces the class of instructions executable by an editor machine.

3. SYNTAX

In order to introduce the syntax of *editor instructions*, we first introduce the notion of *pattern*, i.e., a string of alphabet characters and wildcards, for example, `abc*d` or `*</h*>`.

DEFINITION 2 [Patterns]. Given an alphabet Σ, a *pattern* over Σ is a nonempty string over $\Sigma \cup \{*\}$ not ending with the wildcard, *.

Let us now devote our attention to *editor instructions*; for each instruction, we give the syntax and the intuitive semantics. The precise semantics will be developed in the next section.

DEFINITION 3 [Editor Instructions]. There are six *editor instructions*:

1. The *search instruction* has the form `search(`D`, `Pat`)`; intuitively, it is used to search document D for occurrences of *Pat*.

2. The *replace instruction* has the form `replace(`D`, `S_1`, `S_2`)`, where D is a document and S_1, S_2 are strings over Σ; it is used to replace occurrences of string S_1 with S_2 in document D.

3. The *cut instruction* has the form `cut(`D`)`; it removes the selected region from document D and overwrites it to the clipboard.

4. The *copy instruction* has the form `copy(`*D*`)`; it overwrites the selected region of *D* to the clipboard.

5. The *paste instruction* has the form `paste(`*D*`)`; it pastes the clipboard content to document *D*.

6. The *loop instruction* has the form

`loop search(`*D*, *Pat*`)`

Body

`end loop`

where *Body* is any finite sequence of editor instructions; loops are used to iterate the body as long as occurrences of pattern *Pat* are found in *D*.

DEFINITION 4 [EDITOR Programs]. An EDITOR *program* is a finite sequence of editor instructions.

Since loop bodies may contain any editor instruction, loops may be nested. In the following sections, we devote our attention to programs without nested loops.

4. SEMANTICS

We now want to define the semantics of EDITOR. In order to simplify the presentation, we concentrate on programs in which loops are not nested; we shall see that these programs are computationally complete, so that introducing nested loops does not increase the expressive power. The semantics of a program can be defined operationally based on the effect of its instructions on the associated machine. In order to define the latter, we first have to define the notion of *matching* a pattern.

4.1. Pattern Matching

Patterns are matched against document regions in a rather standard way. Given a pattern *Pat*: (i) each symbol of *Pat* in Σ matches itself; (ii) the wild card, $*$, matches any string of alphabet symbols, including the empty string.

When a document is searched, the document string is scanned from left to right and the *leftmost* matching region is chosen, defined as follows: Among all regions matching the pattern, the *leftmost* matching region is the shortest among those that have the minimum initial position, *i*. Consider, for example document $\triangleright$ `ddababacc` $\triangleleft$ and pattern "`a*ba*c`"; the matching regions are: (2, 9) = `ababac`, (2, 10) = `abababacc`, (4, 9) = `abac`, (4, 10) = `ababacc`. Among them, consider those starting at the earliest position, i.e., (2, 9), (2, 10); the leftmost one is (2, 9), corresponding to `ababac`.

We have developed a specific pattern-matching algorithm for searching a document. The algorithm is an evolution of the traditional automata-theoretic algorithm for searching (constant) strings in a text, due to Aho and Corasick [7]. We have extended the algorithm to manage our restricted form of regular expressions, as

follows. Given a document D and a pattern *Pat* to be searched for in D, the algorithm is as follows. First, we construct the (*deterministic*) *finite state automaton* [31, 21] associated with the regular expression *Pat*; each wildcard is considered a term of the form $(s_1 | s_2 | \cdots | s_n)^*$, where $s_1, s_2, \ldots$ are all the symbols in Σ. Then, the automaton is run on the document string starting from the current position in D; i.e., the first examined symbol coincides with the current position. If the final state of the automaton is reached, the matching process ends and the occurrence is selected. If the last position of D is reached, the search fails and we say that the *selection moves to the end* of D; that is, region $(n, n+1)$ is selected. The pattern-matching algorithm is detailed in the Appendix.

It is worth noting that our algorithm needs to inspect each document symbol only once while simulating the automaton. Moreover, the *size*—i.e., the number of states—of the finite state automaton is linear with respect to the size of the pattern. Note that this is not true for general regular expressions: in that case, the deterministic finite state automaton may have exponential size with respect to the size of the pattern [6, 21]. The linear size of the associated deterministic automaton is an important property of our patterns, which allows for efficient implementations. Although our patterns are less expressive than general regular expressions, we shall see in the following sections that this does not reduce the expressive power of the formalism. In fact, searches are "embedded" in our programming language, so that, by successive simple searches, arbitrarily complex structures can be recognized.

With these ideas in mind, we now define the semantics of nonnested programs as follows.

4.2. Semantics of Nonnested Programs

The semantics of EDITOR programs can be defined in terms of the semantics of its instructions; first, when the execution of a program starts, the current selection in each document coincides with the region $(0, 1)$; then, instructions refer to the current selection and current position in a document. In fact, whenever a document is searched, the search starts from the current position; that is, the character corresponding to the current position is the first character to be examined. If the search succeeds, then the new region is selected; otherwise, the end of the document is reached and the selection moves to the end. In this case, region $(n, n+1)$ is selected. However, to allow further searches, documents are treated *circularly*; that is, the top delimiter, $\triangleright$, is considered as immediately following the bottom one. In this way, after the selection has moved to the end, the next search can start again from the beginning of the document. Note that, without this circular behavior, each document symbol could be examined only once, thus reducing the expressibility of the formalism.

Instructions are executed as follows.

1. When a search instruction `search(`D, *Pat*`)` is executed, document D is searched for a region matching pattern *Pat*; the search starts from the current position. If there are occurrences of the pattern following the current position, the leftmost match is selected; otherwise, the selection moves to the end.

2. When a replace instruction `replace(`D, S_1, S_2`)` is executed, (i) if S_1 is the empty string, then S_2 is inserted in D right before the current position, and this occurrence of S_2 is selected; (ii) otherwise, document D is searched for an occurrence of string S_1; the search starts from the current position. If there are occurrences of the string following the current position, then the leftmost one is replaced with S_2, and this occurrence of S_2 is selected; otherwise, the selection moves to the end.

3. When a cut instruction `cut(`D`)` is executed, the selected region of D is removed from document D and overwritten to the clipboard. If the selection is a nonempty region (i, j), the new selection is $(i, i+1)$. As a special case, if the selection is empty, the document is not changed and the instruction has the only effect of emptying the clipboard.

4. When a copy instruction `copy(`D`)` is executed, the selected region of D is overwritten to the clipboard; the document is not changed. As a special case, if the selection is empty, the clipboard becomes empty.

5. When a paste instruction `paste(`D`)` is executed (assume i is the current position), if the clipboard is nonempty, its content is pasted to document D right before position i; the pasted region becomes the current selection. If the clipboard is empty, the document is not changed and the new selection is the empty region $(i-1, i)$.

6. When a loop instruction `loop search(`D, Pat`)` *Body* `end loop` is executed,

- the current selection in D is moved to the top, that is, to region $(0, 1)$; in this way, when the loop starts, document D is searched starting from the beginning;
- the search instruction `search(`D, Pat`)` is evaluated; the search starts from the current position and may either *fail* or *succeed*; it *succeeds* if an occurrence of pattern *Pat* is found before reaching the bottom delimiter; it *fails* if (i) the current position in D is the bottom delimiter or (ii) no occurrence of pattern *Pat* is found before reaching the bottom delimiter;
- if the search succeeds, then the leftmost match is selected, the loop body is executed, and the search is evaluated again, starting from the current position;
- if the search fails, then the loop terminates.

Note that (i) the loop document, D, is examined from the beginning; in fact, when the loop starts, the current position in D is moved to the first document symbol; (ii) at every loop iteration, the search instruction is evaluated again; the loop terminates if and when the search reaches the end of D.

5. EXPRESSIVE POWER

We now study the expressive power of nonnested programs, in terms of *document restructurings.*

DEFINITION 5 [Document Restructuring]. Given an alphabet Σ and the set Σ^* of documents over Σ, a *document restructuring r* of *arity k* is a partial mapping from the k-fold cartesian product of Σ^* to Σ^*:

$$r: (\Sigma^*)^k \to \Sigma^*.$$

A restructuring r is *computable* if it is partial recursive.

EDITOR programs turn out to be very expressive even without nested loops. In fact, we have the following result:

THEOREM 1 (Expressive Power). *Nonnested* EDITOR *programs express the class of computable document restructurings.*

Proof. Based on the operational semantics developed in Section 4, it is easy to see that each restructuring expressible in EDITOR is a computable one. We shall now prove the converse, that is, that for each computable restructuring, there is an EDITOR Program without nested loops computing it.

In fact, each computable restructuring, r, is computed by a Turing machine, M, such that, on input D: (i) if $r(D)$ is defined, M terminates with output $r(D)$; (ii) if $r(D)$ is not defined, then M hangs. Without loss of generality, we can assume that M has only one tape.[3] We shall simulate the computations of M using a program P_r without nested loops. We encode a configuration of the Turing machine as a document as follows. In addition to symbols in the machine alphabet, Σ_M, we use some extra symbols. Suppose the machine has k states, $q_1, q_2, ..., q_k$; then we encode each state using a string of two symbols containing a special symbol, `q`, plus the state number, as follows: `q1, q2, ... qk`. Without loss of generality, we assume that symbol `q` does not belong to the machine alphabet. Square brackets, `[, ]`, are used to denote the tape left and right delimiters; `#` is used to denote the blank. Suppose that the content of the machine tape is $b_1 b_2 \cdots b_m$, that the tape head is currently scanning symbol b_i, and that the machine is in state q_j. We represent this configuration by the sequence $[b_1 b_2 \cdots b_{i-1}\, \mathtt{q}j\, b_i b_{i+1} \cdots b_m \#]$; a single blank symbol, `#`, follows the last character on the tape.

To compute a restructuring, our program performs three tasks: it constructs the initial configuration of the Turing machine, it simulates the Turing-machine computation, and it extracts the output from the final configuration. These tasks are carried out as follows.

Suppose `Tape` is the input document of size n. The first step of the simulation is the generation of the encoding of the initial configuration as follows:

```
replace(Tape, ε, "[q0");
search(Tape, "$");
replace(Tape, ε, "#]");
```

[3] In fact [39], given a k-tape Turing machine M operating within time $f(n)$ we can construct a Turing machine M' operating within time $\mathcal{O}(f(n)^2)$ such that, for any input x, $M'(x) = M(x)$. Note, however, that the proof could easily be adapted in order to directly simulate a k-tape machine.

Note how a special symbol, $, not used elsewhere, is used in the search instruction to force the selection to the end of `Tape`. After generating the initial encoding, we are ready to simulate the Turing machine. We use replace instructions to simulate machine moves, as follows:

Transition	Instructions
$(q_i, a) \rightarrow (q_j, b, stay)$	`replace(Tape, "qia", "qj! b");`
$(q_i, a) \rightarrow (q_j, b, right)$	`replace(Tape, "qia", "bqj!");`

$(q_i, a) \rightarrow (q_j, b, left)$	`replace(Tape, "`a_1`qia", "qj!` a_1`b");`
	`replace(Tape, "`a_2`qia", "qj!` a_2`b");`
	`...`
	`replace(Tape, "`a_n`qia", "qj!` a_n`b");`
	where $a_1, a_2 \cdots a_n$ *are the symbols in* Σ_M

$(q_i, \#) \rightarrow (q_j, b, right)$	`replace(Tape, "qi#", "bqj! #");`
$(q_i, a) \rightarrow (q_h, b, stay)$	`replace(Tape, "qia", "b");`

We use a special symbol, "!", not in Σ_M, to signal the fact that a replacement has been made; that is, a move has been simulated. Transitions in which the head does not move or moves to the right are straightforward; whenever the head moves to the left, we have to use a different replace instruction for each alphabet symbol a_i; this is because only constant strings can be replaced. Blanks require special attention; in fact, in order to simulate a right infinite tape, whenever the rightmost blank symbol is overwritten, a new blank is added to allow for further moves to the right. Finally, note that when the final state, q_h, has been reached, the string `qh` is not written to the tape.

The overall simulation is performed by a loop:

```
loop search(Tape, "q")
  replace(Tape, "$", "$");
  replace(Tape, "qia", "qj!b");
  replace(Tape, ..., ...);
  ...
  replace(Tape, ..., ...);
  replace(Tape, "$", "$");
  replace(Tape, "!", ε);
  replace(Tape, "$", "$");
  replace(Tape, "[", "[");
end loop;
```

The loop iteratively searches for the symbol q on the tape and so terminates as soon as the halting state, q_h, is reached, since at that point, the symbol q no longer appears on the tape, and so the search fails.

Each execution of the body of the loop performs exactly one move. The special symbol "!" is used to guarantee that at most one move is performed: it is inserted at each replacement, after the encoding of the state, and so prevents the execution of subsequent replace instructions; the "!" is then removed at the end of the body of the loop. At the same time, we can say that each execution of the body performs at least one move, since there is one replace instruction for each pair (nonfinal state, alphabet symbol), and the body is executed whenever the state is not the final one. Each replace examines the tape starting from the first symbol, because of the first instruction of the loop body, replace(Tape, "$", "$"), which has the only effect of moving the selection to the end, so that the next instruction can examine the tape from the beginning.

Therefore, we can claim that the execution of the loop completely simulates the computation of the Turing machine.

Finally, we decode the output in document Output, as follows:

```
search(Tape, "[*]");
copy(Tape);
paste(Output);
replace(Output, "[", ε);
replace(Output, "#]", ε);
```

It is not hard to see that program P_r consisting of the instructions above correctly simulates the computation of machine M. Formally, we have the following result: for any input D, program P_r terminates on D if and only if M halts on input D; if P_r terminates on input D, then Output $= r(D)$. ▮

There are two important points to note about this result. First, EDITOR has in fact an iteration mechanism (the loop instruction), but no explicit if–then–else instruction. Thus, in the proof, we need to simulate conditional instructions using the rather implicit if–then–else computation associated with searches; nevertheless, completeness can be achieved. Second, nested loops do not increase the expressive power, so that the hierarchy of expressiveness due to nested loops collapses to the first level.

Since *accepting* a language L essentially means computing a partial mapping L from Σ^* to $\{yes, no\}$, such that, for each document D, $L(D) = yes$ if and only if $D \in L$, the following result is a consequence of Theorem 1.

COROLLARY 1. *Nonnested* EDITOR *programs accept the class of recursively enumerable languages.*

Theorem 1 and Corollary 1 show that nonnested programs have considerable expressive power, being able to simulate the computation of any Turing machine.

Thus, arbitrarily complex structures can be recognized using simple patterns. We now want to investigate the converse, that is, how an EDITOR program can be executed on a Turing machine, and study the complexity of this simulation. This is interesting since we are comparing what can be considered a *random access* computer over strings, the editor machine, with a Turing machine, whose only mode of accessing data is inherently sequential.

In order to discuss the computational complexity of the simulation, we introduce a natural cost model for editor machines, in which time costs are calculated on the basis of the number of editor instructions being executed. Despite the fact that searches or cut and paste instructions may access several document characters, we count each search, replace, cut, copy, and paste instruction as a single time unit. A loop costs as many units as the number of instructions in the body times the number of loop iterations. Time costs are defined in terms of *input sizes*, defined as the sum of the lengths of the input documents.

DEFINITION 6 [Time Cost of EDITOR Programs]. An EDITOR program, P, runs with *time cost* t on inputs $D_1, D_2, ..., D_n$, if t is the total cost of its instructions; the latter is defined as follows: (i) the cost of each search, replace, cut, copy, and paste instruction is one; (ii) the cost of a loop instruction is the total cost of the loop body (plus one) times the number of loop executions. An EDITOR program has *time cost* $f(n)$ if, on each input of size n, it runs with time cost at most $f(n)$.

Similarly, we define the *space cost* of a program as the space used to store documents during the simulation.

DEFINITION 7 [Space Cost of EDITOR Programs]. An EDITOR program, P, runs with *space cost* r on inputs $D_1, D_2, ..., D_n$, if r is the sum, for all used documents, of the maximum document lengths during the computation. An EDITOR program has *space cost* $f(n)$ if, on each input of size n, the program runs with space cost at most $f(n)$.

We can now prove the following fundamental result:

THEOREM 2 (Complexity). *For each nonnested* EDITOR *program of time cost* $\mathcal{O}(f(n))$ *and space cost* $\mathcal{O}(g(n))$, *there is a deterministic multitape Turing machine that executes the program in time* $\mathcal{O}(g(n) f(n))$.

Proof. In order to prove the claim, we shall describe the Turing machine and prove that it can execute search, replace, cut, copy, and paste instructions in linear time with respect to the size of the involved documents.

Suppose for the sake of simplicity that the EDITOR program manipulates k documents, $D_1, D_2, ..., D_k$. In this case, the Turing machine has $3k+2$ tapes.[4] Three tapes are used for each document D_i. The first tape, named T_i, contains the actual document string, plus the top and bottom delimiters. The second and third tapes are used to mark selections in the document; we use unary strings to encode

[4] For clarity's sake, we have adopted a simulation that uses a large number of tapes. However, the number of tapes might be easily reduced by changing the simulation.

positions in the document and to mark regions: The second tape, named $T_{i,\text{left}}$, indicates the left delimiting position, and the third tape, $T_{i,\text{right}}$, the right delimiting position. Two more tapes are needed: one for the clipboard, named T_C, and one to be used as a *service tape*, T_S.

Suppose the program P has m instructions. The Turing machine states are partitioned into m groups, each implementing a single instruction, as follows:

- To execute a cut instruction of the form `cut(`D_i`)`, (i) $T_{i,\text{left}}$, the left delimiter for the current selection in D_i, is used to locate the current selection; (ii) the selection is then overwritten to the clipboard tape, T_C; (iii) when the end of the selection is reached, the remaining part of T_i is overwritten to the service tape, T_S; (iv) then, the service tape content is overwritten to tape T_i following the left delimiter of the old selection; tapes $T_{i,\text{left}}$, $T_{i,\text{right}}$ are correspondingly updated to reflect the new selection. The execution clearly takes linear time with respect to the size of tape T_i.

- To execute a copy instruction of the form `copy(`D_i`)`, (i) the left delimiter for the current selection in D_i, $T_{i,\text{left}}$, is used to locate the current selection; (ii) the selection is then overwritten to the clipboard tape, T_C; tapes $T_{i,\text{left}}$, $T_{i,\text{right}}$ are correspondingly updated to reflect the new selection. The execution takes linear time with respect to the size of tape T_i.

- To execute a paste instruction of the form `paste(`D_i`)`, (i) $T_{i,\text{right}}$, the right delimiter for the current selection in D_i, is used to locate the current position; (ii) the part of T_i following the current position is overwritten to the service tape, T_S; (iii) tape $T_{i,\text{right}}$ is used to move the head of tape T_i to the current position; (iv) the content of tape T_C is overwritten to tape T_i; (v) the content of the service tape T_S is appended to T_i; tapes $T_{i,\text{left}}$, $T_{i,\text{right}}$ are correspondingly updated to reflect the new selection. The execution takes linear time with respect to the size of tape T_i.

- Replace instructions of the form `replace(`D_i, S_1, S_2`)` are executed as follows: If S_1 is not empty, there is first a set of states implementing the (deterministic) finite state automaton associated with S_1; it is used to identify the first occurrence, if any, of the string starting from the current position in tape T_i. If no occurrence is found, the selection is moved to the end, changing the content of tapes $T_{i,\text{left}}$, $T_{i,\text{right}}$; otherwise, as soon as an accepting state is reached, the search stops and the region is marked using tapes $T_{i,\text{left}}$, $T_{i,\text{right}}$. The remaining part of tape T_i is then copied to the service tape. Starting from the left delimiter, S_2 is then overwritten to tape T_i, its end is marked to become the new selection, and the service tape content is overwritten to tape T_i immediately following it. The execution clearly takes linear time with respect to the size of T_i. If S_1 is the empty string, the left delimiter in $T_{i,\text{left}}$ is moved up to the symbol immediately preceding the position in $T_{i,\text{right}}$ and S_2 is inserted using a similar technique.

- Search instructions of the form `search(`D_i`,` *Pat*`)` are executed by simulating the deterministic finite state automaton associated with pattern *Pat* (see the Appendix) and then proceeding in a way similar to that for replace instructions. This can be done in linear time with respect to the size of T_i.

Now, suppose that each loop instruction is implemented in the straightforward way, that is, by first searching the document and then executing the body if the search succeeds, until the search fails. It is apparent that the Turing machine has to simulate $\mathcal{O}(f(n))$ instructions; each instruction takes time at most $\mathcal{O}(g(n))$, and thus the whole simulation is carried on in time at most $\mathcal{O}(f(n)\, g(n))$. ▮

This theorem will be used in the following section to discuss the complexity of a family of programs based on their space and time cost.

6. TERMINATION AND COMPLEXITY

Due to the presence of loops, EDITOR programs may not terminate. This is shown in the following example.

EXAMPLE 6.1 [Nontermination]. Suppose we are given an empty document, `D`, and consider the following program:

```
replace(D, ε, "abc");
loop search(D, "b")
    search(D, "x");
    search(D, "a");
end loop;
```

The program does not terminate, since the loop execution goes on indefinitely. This is due to the interaction between loops and circularity. In fact, document `D` is initially empty. After the execution of the replace instruction, it contains the string `abc`. When the loop is started and `search(D, "b")` is evaluated, the second symbol in the document is selected. Then, the loop body is executed; instruction `search(D, "x")` fails, since `D` only contains the string `abc`. Thus, the selection moves to the end. Then, `search(D, "a")` is executed. Since the document is treated circularly, the search starts from the first symbol, and the `a` is selected. So, when the loop search, `search(D, "b")`, is evaluated again, the search succeeds, the `b` is selected, and the loop body is executed. In this way, the selection is "trapped" between `a` and `b` and the execution goes on indefinitely. With a similar technique it is possible to write programs that, using paste or replace instructions, generate documents of unbounded length.

In order to avoid these undesirable behaviors, we shall impose a natural condition on our programs. The condition consists in requiring that, inside the body of a loop, the status (i.e., the content and the current selection) of the loop document, D, cannot be modified. This corresponds to requiring that the only instructions involving D in the loop body be copy instructions; in fact, any other instruction would either modify the document or affect the current selection. Thus, we define the class of *safe programs* as follows:

DEFINITION 8 [Safe EDITOR Programs]. An EDITOR program P is *safe* if, for each loop instruction

```
loop search(D, Pat)
    Body
end loop
```

in P, the loop body does not contain any search, replace, cut, paste, or loop instruction involving document D.

We have the following result:

THEOREM 3 (Termination of Safe Programs). *Every safe program terminates.*

Proof. The proof is based on the observation that each instruction in a safe program can be executed in finite time. This is immediate for all instructions but loops. As far as loops are concerned, note that, in a safe program, neither the loop document nor the current selection can be modified by the loop body. At each loop iteration, the document is searched for a new occurrence of the specified pattern following the current position. Moreover, in a safe loop, each symbol of the loop document is examined exactly once and no circularity is possible. In fact, when the search reaches the document end, the loop terminates. Since the loop document has finite size, and thus contains a finite number of pattern occurrences, the loop terminates after a finite number of executions. ▮

We now want to study the computational complexity and the expressive power of safe programs without nested loops. We define the *complexity* of a restructuring r as the complexity of computing $r(D)$ on a multitape Turing machine, measured with respect to the length of document D. A language L is said to *express* a class of restructurings C if (i) each restructuring expressible in L has complexity in C and, conversely, (ii) each restructuring with complexity in C can be expressed in L.

Our objective is to characterize the class of *feasible* restructurings, i.e., the class of restructurings computable in polynomial time. Unfortunately, even without nested loops, safe programs may generate documents of exponential size; that is, they may have an exponential space cost. Based on Theorem 2, we know that their deterministic time complexity is therefore (at least) exponential. This is shown in the following example.

EXAMPLE 6.2 [Exponential Size]. Suppose we are given a document D containing n occurrences of the symbol #, plus two delimiters, [,], that is, D = [# # # # ⋯ #], and consider the following program:

```
search(D, "[*]");
copy(D);
paste(D1);
loop search(D, "#")
```

```
        search(D1, "[*]");
        copy(D1);
        paste(D1);
        replace(D1, "][", ε);
        search(D1, "x");
    end loop;
```

The program is safe and does not contain nested loops; it generates a document D1 of exponential size with respect to n. In fact, the first three instructions simply duplicate in D1 the content of D, so that, when the loop starts, D1 has size $n+2$. Then, each execution of the loop body doubles the length of D1 by copying the content of D1 to the clipboard and then pasting it to itself. The replace instruction removes extra delimiters; the final search has the only effect of moving the selection to the end. Since the loop body is executed exactly n times, one for each # in D, the final size of D1 is $2^n + 2$.

In order to further restrict our programs and capture polynomial time computations, we introduce another condition that avoids these exponential behaviors. In fact, the reason the size of D1 in Example 6.2 grows exponentially is that at each loop iteration the clipboard is used to double the size of the document generated at the previous iteration. In essence, the clipboard acts as a feedback device through which the content of document D1 can be pasted to itself. To disallow this feedback, we introduce the class of *simple* programs, as the class of programs in which loops do not contain search instructions. In this way, the clipboard size cannot grow during the loop execution.

DEFINITION 9 [Simple Programs]. A program is *simple* if loops do not contain search instructions.

Safe simple programs have polynomial time complexity. In fact, we have the following fundamental result:

THEOREM 4 (PTIME Expressibility). *The class of safe simple* EDITOR *programs expresses exactly* PTIME, *that is, the class of document restructurings computable in polynomial time.*

Proof. We shall first prove the complexity upper bound and then the expressibility lower bound.

- *Complexity.* We first show that simple safe programs can be executed in polynomial time. In order to do this, first we study the time and space cost of these programs, and then, based on Theorem 2, we show that they can be executed in polynomial time using a Turing machine.

In the following, we use D_0 to refer to the input document and $D_1, D_2, \ldots$ to refer to other documents; C denotes the clipboard. Given a document D, we refer to its size as SIZE(D). In order to study the time and space cost of programs, we partition them in *blocks* of instructions; a block is either (i) a sequence of instructions not

containing loops or (ii) a single loop. We are interested in the input–output behavior of these blocks, so that we shall indicate by $\text{SIZE}(D, j)$ the size of document D after the execution of block j and by $T(j)$ the time cost of the computation of block j.

— *Blocks without loops.* Consider first a block containing a single nonloop instruction. Suppose that, initially

$$\text{SIZE}(D_i) = \mathcal{O}(n) \qquad (i = 0, 1, 2, \ldots), \qquad \text{SIZE}(C) = \mathcal{O}(n).$$

After the block is executed, the size of the clipboard and of each input document is linear with respect to the input size. Now, a simple case by case analysis shows that, for a block j containing k instructions but no loops, we have

$$\text{SIZE}(D_i, j) = \mathcal{O}(n) \quad (i = 0, 1, 2, \ldots), \qquad \text{SIZE}(C, j) = \mathcal{O}(n), \qquad T(j) = k.$$

Thus, every finite sequence of editor instructions without loops has time and space cost that are linear with respect to the size of the original inputs.

— *Loop blocks.* Let us now consider a block j containing a single loop. Suppose we have initially that

$$\text{SIZE}(D_i) = \mathcal{O}(n) \quad (i = 0, 1, 2, \ldots), \qquad \text{SIZE}(C) = \mathcal{O}(n).$$

Let us consider the execution of a loop of the form `loop search(`D_x, *Pat*`)` *Body* `end loop`. We know that, since the program is safe, simple, and nonnested, the loop body is a block containing p instructions with no loops or searches; thus, any execution of the body requires time cost $p + 1$. Since the loop is iterated at most $\mathcal{O}(n)$ times, the total time cost is $\mathcal{O}(n)$. Let us now discuss how the space cost changes as an effect of executing the body. We know that the body cannot contain search instructions; thus, the clipboard size cannot increase during the loop execution. Copy and cut instructions do not make the size of documents grow, and replace instructions can make it grow at most of a constant size; thus, we need to consider only paste instructions. Each of these appends the clipboard content to some document, D_h. We know that, initially, $\text{SIZE}(D_h) = \mathcal{O}(n)$. After each execution of a paste instruction, the size grows by an additive term of $\mathcal{O}(n)$, so that, after the first body execution we have $\text{SIZE}(D_h) = \mathcal{O}(n) + cn$; after the second execution, we have $\text{SIZE}(D_h) = \mathcal{O}(n) + 2cn$; after the ith execution, we have $\text{SIZE}(D_h) = \mathcal{O}(n) + icn$. Since the loop can be executed at most $\mathcal{O}(n)$ times, when the loop execution terminates we have that $\text{SIZE}(D_h) = \mathcal{O}(n) + \mathcal{O}(n)n = \mathcal{O}(n^2)$; thus,

$$\text{SIZE}(D_i, j) = \mathcal{O}(n^2) \quad (i = 0, 1, 2, \ldots), \qquad \text{SIZE}(C, j) = \mathcal{O}(n) \qquad T(j) = \mathcal{O}(n).$$

Based on these considerations it is easy to study the behavior of a program. Suppose the program has l loops; then, the program has at most $2l + 1$ blocks. l blocks $\text{loop}_1, \text{loop}_2, \ldots, \text{loop}_l$ correspond to loops, and $l + 1$ blocks to instructions not containing loops. Based on the formulas developed above, we have that the

overall time cost is linear with respect to the size of the input document and the overall space cost is polynomial. In fact, for each document D_j, $j = 0, 1, 2, ...$, we have the following:

$$\begin{aligned}
&\text{SIZE}(D_j, \text{loop}_1) = \mathcal{O}(n^2), && T(\text{loop}_1) = \mathcal{O}(n)\\
&\text{SIZE}(D_j, \text{loop}_2) = \mathcal{O}(n^4), && T(\text{loop}_2) = \mathcal{O}(n^2)\\
&\text{SIZE}(D_j, \text{loop}_3) = \mathcal{O}(n^8), && T(\text{loop}_3) = \mathcal{O}(n^4)\\
&\cdots && \cdots\\
&\text{SIZE}(D_j, \text{loop}_l) = \mathcal{O}(n^{2^l}), && T(\text{loop}_l) = \mathcal{O}(n^{2^{l-1}}).
\end{aligned}$$

Thus, since l is a constant, the whole program has polynomial time and space cost. By Theorem 2, it has polynomial deterministic time complexity.

- *Expressibility*. We must now show that every restructuring in PTIME can be computed by a safe program without nested loops. Any such restructuring is computed by a Turing machine, M, that runs in polynomial time. Without loss of generality, we can assume that M has only one tape and runs in time n^k for some k, where n is the length of its input. We shall simulate the computations of M using an EDITOR program.

We encode a configuration of the Turing machine in the usual way (see proof of Theorem 1). To compute a restructuring, our program performs three tasks: It constructs the initial configuration of the Turing machine, it simulates n^k Turing-machine steps using a counter, and it extracts the output from the final configuration. These tasks are carried out as follows.

Suppose `Input` is the input document, of size n. The first step of the simulation is the generation of document `Counter`, of size n^k, to be used as a counter during the simulation; this document contains a single symbol, #, repeated n^k times. We do this in several steps. First, based on the input, we generate a document $\texttt{Counter}_1$, containing n #'s. To do this, we need to replace each alphabet symbol with a #. We use one loop for each alphabet symbol, `a`, `b`, ..., as follows:

```
replace(Input, ε, "[");
search(Input, "$");
replace(Input, ε, "]");
search(Input, "[*]");
copy(Input);
paste(Counter1);
loop search(Input, "a")
    replace(Counter1, "a", "#");
end loop;
```

```
loop search(Input, "b")
    replace(Counter_1, "b", "#");
end loop;
...
```

Then, after generating $\texttt{Counter}_1$, we use $p = \lceil \log_2(k) \rceil$ loops to generate a counter of length n^k, as follows:

```
search(Counter_1, "[*]");
copy(Counter_1);
loop search(Counter_1, "#")
    paste(Counter_2);
    replace(Counter_2, "][", ε);
end loop;
...
search(Counter_{p-1}, "[*]");
copy(Counter_{p-1});
loop search(Counter_{p-1}, "#")
    paste(Counter);
    replace(Counter, "][", ε);
end loop;
```

Once we have generated the counter document, we generate the initial encoding of the Turing machine in the usual way (see Theorem 1). Then, we are ready to simulate the Turing machine. However, in this case we only simulate n^k machine steps, using document `Counter` as a counter:

```
loop search(Counter, "#")
    replace(Tape, "$", "$");
    replace(Tape, "qia", "qj!b");
    replace(Tape, ..., ...);
    ...
    replace(Tape, ..., ...);
    replace(Tape, "$", "$");
    replace(Tape, "!", ε);
end loop;
```

Finally, we decode the output document in the usual way (see proof of Theorem 1). It is easy to see that the resulting program is safe and correctly computes the machine output. ▮

The result is interesting since it provides a characterization of the complexity of EDITOR programs based on PTIME. See [10, 19, 25, 34] for other PTIME expressibility results. Note that, although EDITOR programs can be considered a string counterpart of *loop programs* [26, 38], their complexity is considerably different. *Loop programs* are a formalism for computing on integers; they work on integer variables and are based on simple forms of assignment, such as $X = 0$, or $X = X + 1$, plus a loop instruction of the form `loop` *X Body* `end`, used to iterate the body as many times as the value of variable *X*. EDITOR programs are somehow similar to loop programs. However, there are some fundamental differences between the two formalisms: first, loop programs of arbitrary nesting do not express all computable functions, whereas EDITOR programs do, even without nested loops; second, loop programs without nested loops have very low complexity, essentially linear time, nested loops make the complexity jump to exponential time, and there is no known class of loop programs capturing polynomial time computations.

7. CONCLUSION

In the previous sections we have developed EDITOR essentially as a language for text search and restructuring, and studied its computational properties. We claim that such a language can be profitably used as a tool for wrapping documents such as HTML pages and building database representations of their content. Indeed, the language has been implemented as a set of Java classes and has been used as a basis for the ARANEUS wrapper toolkit called MINERVA, as described in [20].

We believe text restructuring to be a primary component of Web computing. In this respect, EDITOR itself can be seen as a programming language for the Web, provided that it is enriched with a simple mechanism for downloading HTML pages from the Internet. This can be easily done by augmenting the language with a download primitive of the form

`download(`*HREF*, *TargetDocument*`)`

which has two parameters, the first being a URL, *HREF*, and downloads from the network the (textual) document at address *HREF*, storing its content in the second parameter, *TargetDocument* (whose previous content is overwritten). If *HREF* is not a well-formed URL according to the HTTP protocol or the corresponding document is not a textual one, *TargetDocument* will be empty.

This new language, which we might call the WEB-EDITOR, allows expression of complex computations on the Web. As an example, consider the piece of code in Fig. 4, which computes a form of transitive closure of a portion of the Web: it starts from a given address, `http://www.unibas.it`, and downloads all documents that are reachable from it, storing their titles in a result document called `Closure`. For the sake of simplicity, in writing the program we make a number of hypotheses: First, we assume that the portion of the Web under consideration does

```
1  replace(Buffer, ϵ, "http://www.unibas.it\n");  Writes the starting URL in Buffer;
2  loop search(Buffer, "http://*\n" )             searches for all URLs in the buffer;
3     cut(Buffer); paste(URL);                    and pastes each to URL;
4     download(URL, Page);                        downloads page URL in Page;
5     search(Page, "<TITLE>*</TITLE>");           searches for the title
6     copy(Page); paste(Closure);                 and pastes it to Closure;
7     loop search(Page, "http://*.html" )         extracts all URLs from Page;
8        copy(Page); paste(Buffer);               and pastes them to the buffer,
9        replace(Buffer, ϵ, "\n");                adding a new line after each URL;
10    end loop;                                   end of the internal loop;
11    cut(URL);                                   empties document URL;
11    reset(Buffer);                              moves back to the beginning of Buffer;
12 end loop;                                      end of the external loop.
```

FIG. 4. WEB-EDITOR program for computing transitive closure.

not contain cycles; otherwise the program would loop forever. Second, we assume that all URLs encountered during the navigation start with `http://` and end with `.html`. This is not true in general. However, it is rather straightforward to write a longer program capable of handling cycles and URLs of any form.[5] In the program we also use an instruction of the form `reset(`D`)`, which simply resets the current selection and position to the beginning of a document D.

Although this is only a simple example, it shows the potential of coupling URL access with text manipulation. We consider this a promising starting point toward the definition of a theory of computability and complexity on the Web [5, 37].

APPENDIX: THE PATTERN-MATCHING ALGORITHM

In this section we discuss the algorithm used to match patterns against document regions. The algorithm is an extension of the Aho–Corasick algorithm [7] for searching occurrences of a constant string in a text and is based on the simulation of the deterministic finite state automaton associated with the pattern. We have extended the traditional algorithm, which only deals with constant strings, in order to manage our patterns. Note that we have chosen not to use the ordinary algorithm for regular expression matching [21], since in some cases the deterministic finite state automaton has an exponential number of states with respect to the length of the pattern. For our restricted patterns, however, our algorithm generates automata of linear size and runs in linear time with respect to the size of the pattern.

The algorithm has two steps: First, given a pattern, the corresponding deterministic finite state automaton is derived; then, the automaton is simulated on the document to search for occurrences of the pattern. The automaton is represented as a quadruple, (Σ, δ, s, f), where Σ is the alphabet, δ is the transition function, and s and f are the initial and final states, respectively. The construction of the automaton is performed using the algorithm `buildDFA` in Fig. 5. States are

[5] Indeed, we have tested a more sophisticated form of this program on real pages, to learn that transitive closure is to be approached with care when dealing with the Web, since the size of the results tends to explode very quickly. Even stopping the computation at distance 4 or 5 can yield very large results, on the order of the thousands of pages.

```
1   FUNCTION buildDFA (P: pattern) :  automaton;
2   VAR i :  integer;
3      state, temp :  integer;
4      wildCards :  boolean;
5   BEGIN
6      wildCards := false;
7      firstWcState := -1;
8      state := 0;
9      FOR each a_j ∈ Σ DO
10        δ(0,a_j) := 0;
11     FOR i := 0 to length(P)-1 DO
12     BEGIN
13        IF P[i] <> '*' THEN
14        BEGIN
15           temp := δ(state, P[i]);
16           δ(state, P[i]) := state+1;
17           FOR each a_j ∈ Σ DO
18              δ(state + 1, a_j) := δ(temp, a_j);
19           state := state+1;
20        END
21        ELSE /* P[i] is a wildcard */
22        BEGIN
23           IF NOT(wildCards) THEN
24           BEGIN
25              firstWcState := state;
26              wildCards := true;
27           END;
28           FOR each a_j ∈ Σ DO
29              δ(state, a_j) := state;
30        END
31     END;
32     finalState := state;
33     RETURN (Σ, δ, 0, finalState, firstWcState);
34  END;
```

FIG. 5. Algorithm buildDFA.

represented using numbers; if the automaton has k states, the starting state is 0, and the unique final state is $k-1$. The main point of the construction consists in assigning the transition function, δ. This can be seen as a matrix having states on the rows and alphabet symbols on the columns. The matrix is constructed by initializing the first row to 0 (lines 9–10) and then progressively unfolding the transitions associated with each pattern symbol p_i. The variable `state` denotes the current working state, to become the automaton final state at the end of the construction. Lines 15–19 are from the Aho–Corasick algorithm [7] and unfold transitions corresponding to constant characters in the pattern. Lines 23–29, in contrast, take care of wild cards. In this case, the state is simply treated as a "trap" state. In addition to elements in the quadruple above, the function returns an integer value, `firstWcState`. If the pattern contains wild cards, this is the state that corresponds to matching the first wild card; it is -1 otherwise. This value will be used when simulating the automaton in order to select the matched region.

Once the automaton has been derived, it is simulated on the document starting from the current position, according to the algorithm `simulaDFA` in Fig. 6. The code needed to run the automaton on the document is rather straightforward. We use integers to denote positions in the document. Two variables are used to mark

```
1   PROCEDURE simulaDFA (D: document, P: pattern, DFA: automaton,
2                        VAR start, position:  integer; VAR found :  boolean)
3   VAR state, firstPrefixLength :  integer;
4     found, firstWcReached :  boolean;
5   BEGIN
6     start := position-1;
7     IF (P <> "*") THEN
8     BEGIN
9       state := 0;
10      firstWcReached := false;
11      found := false;
12      firstPrefixLength := indexOf(P, '*');
13      if (firstPrefixLength = -1)
14        firstPrefixLength := length(P);
15      WHILE (position < length(D)) AND NOT(found) DO
16      BEGIN
17        state := δ(state, D[position]);
18        IF (P[0] <> '*') AND (state = firstWcState) AND NOT(firstWcReached) THEN
19        BEGIN
20          start:=position - firstPrefixLength +1;
21          firstWcReached := true;
22        END;
23        IF (state=finalstate) THEN
24          found:=true;
25        position:=position+1;
26      END
27      IF (firstWcState = -1) THEN
28        start := position - firstPrefixLength;
29    END
30    END;
```

FIG. 6. Algorithm simulaDFA.

the selection in case the matching succeeds: `start` is used to denote the left delimiter for the current selection, and `position` to denote the current position. Special care must be taken in recording where matches start, in order to correctly select the matching region after the simulation has been performed. To do this, we use two variables: `firstWcState`, derived above, and `firstPrefixLength`. The latter is the number of constant symbols preceding the first wild card in the pattern. If the pattern does not contain wild cards, `firstPrefixLength` equals the length of the pattern itself. The strategy used to record where a matching starts consists in checking when state `firstWcState` is entered for the first time. At this point, the first constant prefix has been matched and therefore we set `start` to `position` minus `firstPrefixLength` +1.

ACKNOWLEDGMENTS

We thank all people in the ARANEUS project at the Università di Roma Tre, including Paolo Merialdo, Alessandro Masci, Giuseppe Sindoni, and Valter Crescenzi, for stimulating discussions on the subject of this paper. In particular, Alessandro and Valter provided many insightful comments and significantly contributed to the implementation of the system. Thanks also go to Alberto Mendelzon for interesting preliminary discussions on document query languages. Finally, we thank the anonymous reviewers for comments that helped us to improve the presentation. This work was supported by the Università di Roma Tre, MURST, and the Consiglio Nazionale delle Ricerche.

REFERENCES

1. The Capodimonte Museum Web site (http://capodimonte.selfin.net).
2. The Python Language Home Page (http://www.python.org).
3. S. Abiteboul, Querying semi-structured data, *in* "Sixth International Conference on Data Base Theory (ICDT'97), Delphi (Greece)," Lecture Notes in Computer Science, Springer-Verlag, Berlin/New York, 1997.
4. S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Siméon, Querying documents in object databases, *J. Digital Libraries* **1**, No. 1 (April 1997), 5–19.
5. S. Abiteboul and V. Vianu, Queries and computation on the Web, *in* "Sixth International Conference on Data Base Theory, (ICDT'97), Delphi (Greece)," Lecture Notes in Computer Science, Springer-Verlag, Berlin/New York, 1997.
6. A. V. Aho, Algorithms for finding patterns in strings, *in* "Handbook of Theoretical Computer Science" (J. Van Leeuwen, Ed.), Elsevier Science Publishers (North Holland), Amsterdam, 1990.
7. A. V. Aho and M. Corasick, Efficient string matching: An aid to bibliographic search, *Comm. ACM* **18**, No. 6 (1975), 333–340.
8. P. Atzeni, G. Mecca, and P. Merialdo, To weave the web, *in* "International Conference on Very Large Data Bases (VLDB'97), Athens, Greece, August 26–29, 1997," pp. 206–215.
9. P. Atzeni, G. Mecca, and P. Merialdo, Design and maintenance of data-intensive web sites, *in* "Sixth International Conference on Extending Database Technology (EDBT'98), Valencia, Spain, March 23–27, 1998."
10. S. Bellantoni and S. Cook, A new recursion-theorethic characterization of the polytime functions, *in* "ACM International Symposium on Theory of Computing," pp. 283–293, 1992.
11. T. Berners-Lee, R. Cailliau, A. Lautonen, H. F. Nielsen, and A. Secret, The World Wide Web, *Comm. ACM* **37**, No. 8 (August 1994), 76–82.
12. G. E. Blake, M. P. Consens, P. Kilpeläinen, P. Larson, T. Snider, and F. W. Tompa, Text/relational database management systems: Harmonizing SQL and SGML, *in* "First Intern. Conf. on Applications of Databases, (ADB'94), Vadstena, Sweden," Lecture Notes in Computer Science, Vol. 819, pp. 267–280, Springer-Verlag, Berlin/New York, 1994.
13. A. J. Bonner and G. Mecca, Sequences, datalog and transducers, *J. Comput. System Sci.* **57** (1998), 234–260.
14. P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu, A query language and optimization techniques for unstructured data, *in* "ACM SIGMOD International Conference on Management of Data (SIGMOD'96), Montreal, Canada," pp. 505–516, 1996.
15. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom, The TSIMMIS project: Integration of heterogenous information sources, *in* "IPSJ Conference, Tokyo, 1994."
16. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl, From structured documents to novel query facilities, *in* "ACM SIGMOD International Conference on Management of Data (SIGMOD'94), Minneapolis," pp. 313–323, 1994.
17. C. L. A. Clarke and G. V. Cormack, On the use of regular expressions for searching text, Technical Report No. CS-95-07, University of Waterloo, 1995.
18. C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski, An algebra for structured text search and a framework for its implementation, *Comput. J.* **38**, No. 1 (1995), 43–56. [Also available as Technical Report No. CS-94-30, University of Waterloo.]
19. L. S. Colby, E. L. Robertson, L. V. Saxton, and D. Van Gucht, A query language for list-based complex objects, *in* "Thirteenth ACM SIGMOD International Symposium on Principles of Database Systems (PODS'94)," pp. 179–189, 1994.
20. V. Crescenzi and G. Mecca, Grammars have exceptions, *Inform. Systems* **23**(8) (1998), 539–565.
21. M. Crochemore and W. Rytter, "Text Algorithms," Oxford Univ. Press, London, 1994.

22. G. H. Gonnet, Text dominated databases: Theory, practice and experience, Tutorial presented at PODS, 1994.
23. G. H. Gonnet and F. W. Tompa, Mind your grammar: A new approach to modelling text, *in* "Thirteenth International Conference on Very Large Data Bases, Brighton (VLDB'87)," pp. 339–346, 1987.
24. I. S. Graham, "HTML Sourcebook," Wiley, New York, 1995.
25. S. Grumbach and T. Milo, An algebra for POMSETS, *in* "Fifth International Conference on Data Base Theory (ICDT'95), Prague," Lecture Notes in Computer Science, pp. 191–207, Springer-Verlag, Berlin/New York, 1995.
26. J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages and Computation," Addison–Wesley, Reading, MA, 1979.
27. ISO, International Organization for Standardization, "ISO-8879: Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)," October 1986.
28. P. Kilpeläinen, G. Lindén, H. Mannila, and E. Nikunen, A structured document database system, *in* "International Conference on Electronic Publishing, Document Manipulation and Typography (EP'90)," pp. 139–151, 1990.
29. D. Konopnicki and O. Shmueli, W3QS: A query system for the World-Wide Web, *in* "International Conference on Very Large Data Bases (VLDB'95), Zurich," pp. 54–65, 1995.
30. L. Lakshmanan, F. Sadri, and I. N. Subramanian, A declarative language for querying and restructuring the web, *in* "6th International Workshop on Research Issues in Data Engineering: Interoperability of Nontraditional Database Systems (RIDE–NDS'96)," 1996.
31. H. Lewis and C. H. Papadimitriou, "Elements of the Theory of Computation," Prentice–Hall, New York, 1981.
32. A. Loeffer, Text databases: A survey of text models and systems, *SIGMOD Record* **23**, No. 1 (March 1994), 97–106.
33. S. Marcus and V. S. Subrahmanian, Foundations of multimedia database systems, *J. Assoc. Comput. Mach.* **43**, No. 3 (May 1996), 474–523.
34. G. Mecca, "From Datalog to Sequence Datalog: Languages and Techniques for Querying Sequence Databases," Ph.D. thesis, Università di Roma "La Sapienza," 1996.
35. G. Mecca, A. Mendelzon, and P. Merialdo, Efficient queries over web views, *in* "Sixth International Conference on Extending Database Technology (EDBT'98), Valencia, Spain, March 23–27, 1998."
36. A. Mendelzon, G. Mihaila, and T. Milo, Querying the World Wide Web, *in* "First International Conference on Parallel and Distributed Information Systems (PDIS'96), 1996."
37. A. O. Mendelzon and T. Milo, Formal models of Web queries, *in* "Sixteenth ACM SIGMOD International Symposium on Principles of Database Systems (PODS'97), Tuscon, Arizona, 1997."
38. A. R. Meyer and D. M. Ritchie, Computational complexity and program structure, *I.B.M. Res. Rep.* (1967), 1817.
39. C. H. Papadimitriou, "Computational Complexity," Addison–Wesley, Reading, MA, 1994.
40. F. Salminen and F. W. Tompa, "PAT Expressions: An Algebra for Text Search," Technical Report OED-92-02, University of Waterloo, 1992.
41. R. L. Schwartz, "Learning Perl," O'Reilly & Associates, 1993.