*Natural Language Engineering* **14** (4): 457–469. © 2007 Cambridge University Press
doi:10.1017/S1351324907004676 Printed in the United Kingdom


# *Strengths and weaknesses of finite-state technology: a case study in morphological grammar development*

SHULY WINTNER

*Department of Computer Science, University of Haifa, 31905 Haifa, Israel*
*e-mail*: shuly@cs.haifa.ac.il

(*Received 12 January 2007; revised 31 May 2007; accepted 9 October 2007; first published online 6 December 2007*)

**Abstract**

Finite-state technology is considered the preferred model for representing the phonology and morphology of natural languages. The attractiveness of this technology for natural language processing stems from four sources: modularity of the design, due to the closure properties of regular languages and relations; the compact representation that is achieved through minimization; efficiency, which is a result of linear recognition time with finite-state devices; and reversibility, resulting from the declarative nature of such devices. However, when wide-coverage morphological grammars are considered, finite-state technology does not scale up well, and the benefits of this technology can be overshadowed by the limitations it imposes as a programming environment for language processing. This paper investigates the strengths and weaknesses of existing technology, focusing on various aspects of large-scale grammar development. Using a real-world case study, we compare a finite-state implementation with an equivalent Java program with respect to ease of development, modularity, maintainability of the code, and space and time efficiency. We identify two main problems, *abstraction* and *incremental development*, which are currently not addressed sufficiently well by finite-state technology, and which we believe should be the focus of future research and development.

## 1 Introduction

Finite-state technology (FST) denotes the use of finite-state devices, including automata and transducers, in natural language processing (NLP). Since the early works that demonstrated the applicability of this technology to linguistic representation (Johnson 1972; Koskenniemi 1983; Kaplan and Kay 1994), FST is considered adequate for describing the phonological and morphological processes of the world's languages (Roche and Schabes 1997; Beesley and Karttunen 2003). Even nonconcatenative processes such as circumfixation, root-and-pattern morphology, or reduplication, were shown to be in principle implementable in FST (Beesley 1998; Cohen-Sygal and Wintner 2006).

The utility of FST for NLP was emphasized by the implementation of several toolboxes that provide extended regular expression languages and compilers that convert expressions to finite-state automata and transducers. These include *INTEX*

(Silberztein 1993); *FSM* (Mohri, Pereira, and Riley 2000), which is a unix-based set of programs for manipulating automata and transducers; *FSA Utilities* (van Noord and Gerdemann 2001), which is a freely available, Prolog-implemented system; *XFST* (Beesley and Karttunen 2003), which is a commercial package assumed to be the most suitable for linguistic applications by providing the most expressive language; RWTH FSA (Kanthak and Ney 2004) and Carmel (`http://www.isi.edu/licensed-sw/carmel/`), which support both weighted and unweighted automata; and SFST (Schmid 2005), which is still work in progress.

Several properties of finite-state devices contribute to their utility for NLP applications:

*True representation:* Following the pioneering work of Johnson (1972), it is now clear that the kind of phonological and morphological rules that are common in linguistic theories can be directly implemented as finite-state relations. The implementation of linguistically motivated rules in FST is therefore straightforward and direct (Karttunen 1995).

*Modularity:* The closure properties of regular languages and relations provide various means for combining regular expressions, supporting a variety of operations these expressions denote on the languages. For example, closure under *union* facilitates a separate development of two grammar fragments which can then be directly combined in a single operation. The most useful operations under which transductions are closed is probably *composition*, which is the central vehicle for implementing *replace rules* (Kaplan and Kay 1994; Karttunen 1995).

*Compactness:* Finite-state automata can be minimized, guaranteeing that for a given language, an automaton with a minimal number of states can always be generated. Toolboxes can apply minimization either explicitly or implicitly to improve storage requirements.

*Efficiency:* When an automaton is deterministic, recognition is optimally efficient (linear in the length of the string to be recognized). Automata can always be determinized, and toolboxes can take advantage of this to improve time efficiency.

*Reversibility:* Finite-state automata and transducers are inherently declarative: it is the application program that implements either recognition or generation. In particular, transducers can be used to map strings from the upper language to the lower language or vice versa with no changes in the underlying finite-state device.

These benefits encouraged the development of several large-scale morphological grammars for a variety of languages, including some with complex morphology such as Finnish (Koskenniemi 1983), German (Görz and Paulus 1988; Trost 1990), French (Silberztein 1993; Chanod and Tapanainen 1996), Turkish (Oflazer 1994), Arabic (Beesley 1996; Beesley 1998), and Hebrew (Yona and Wintner to appear).

While FST is instrumental in modeling the morphological, phonological, and orthographic phenomena of natural languages, as well as for rapid prototyping of implementations of these phenomena, the main claim of this paper is that

when the development of large-scale grammars is concerned, FST does not scale up well. This claim is supported by a realistic case study defining a sophisticated morphological task (Section 2), both using FST (Section 2.1) and with an alternative implementation in Java of the same grammar (Section 2.2). The two approaches are compared in Section 3 along several axes, focusing on engineering aspects of the grammar development process. The conclusion (Section 4) is the identification of two main Achilles' heels in contemporary technology: the lack of *abstraction* mechanisms and the computational burden of incremental changes. We believe that these two issues should be the focus of future research in FST.

## 2 A case study

To evaluate the scalability of FST we consider, as a benchmark, a large-scale task accounting for the morphological and orthographic phenomena of Hebrew, a natural language with nontrivial morphology. Clearly, languages with simple morphology (e.g., English) do not benefit from FST approaches, simply because it is so inexpensive to generate and store all the inflected forms. It is only when relatively complicated morphological processes are involved that the benefits of FST become apparent, and Hebrew is chosen here only as a particular example; the observations reported in Section 3 are valid in general for all similar tasks.

Hebrew, like other Semitic languages, has a rich and complex morphology. The major word formation machinery is root-and-pattern morphology, where roots are sequences of three (typically) or more consonants and patterns are sequences of vowels and, sometimes, also consonants, with 'slots' into which the root's consonants are inserted. After the root combines with the pattern, some morpho-phonological alterations take place, which may be nontrivial. The combination of a root with a pattern produces a *lexeme*, which can then be inflected in various forms. Inflectional morphology is highly productive and consists mostly of suffixes, but sometimes of prefixes or circumfixes. The morphological problems are amplified by issues of orthography. The standard Hebrew script leaves most of the vowels unspecified. Furthermore, many particles, including prepositions, conjunctions, and the definite article, attach to the words that immediately follow them. As a result, surface forms are highly ambiguous.

The finite-state grammar that we used as a benchmark here is HAMSAH (Yona and Wintner to appear), an XFST implementation of Hebrew morphology. XFST was chosen for this task because it is the most developed FST toolbox: it provides the largest set of operators developed specifically for linguistic applications, especially morphology and phonology; it represents years of development in optimizing the underlying representation and the finite-state algorithms that operate on them; and it is the best documented FST software package. Other implementations are limited in expressivity, too low level, or poorly scalable. While we compare a specific XFST grammar with a specific Java program, our observations are more general, and refer to the differences between finite-state grammars and alternative, direct implementations in a general-purpose programming language.

```
[+noun][+id 16236][+form xlwn][+gender masculine][+number singular]
[+definiteness false][+register formal][+plural wt]
```

Fig. 1. The XFST lexical representation of the noun *xlwn* 'window'.

### *2.1 An FST implementation*

The XFST grammar is obtained by composing a large-scale lexicon of Hebrew (>20,000 entries) with a large set of rules, implementing mostly morphological and orthographic processes in the language. As the lexicon is developed independently (Itai, Wintner, and Yona 2006), and is represented in SQL and XML, it must be converted to XFST before it can be incorporated in the grammar. This is done by a set of Perl scripts that had to be specifically written for this purpose. In other words, the system itself is not purely finite state, and we maintain that few large-scale systems for morphological analysis can be purely finite state, as such systems must interact with independently developed components such as lexicons, annotation tools, user interfaces, etc.

Since the lexicon was developed independently, the grammar had to be adapted to the format of the lexicon. In particular, the citation form of lexical items is the traditional one (e.g., singular masculine for nouns and adjectives and third person singular masculine past tense for verbs). Consequently, the XFST lexicon is based on such citation forms. For example, the XFST variable `noun` denotes the set of all lexical items whose part of speech is noun; by default, these nouns are singular masculine, and this is indicated by adding `number singular` to the lexical specification (some nouns are inherently plural, e.g., *mim* 'water', and they are specified as `number plural` in the lexicon). Hebrew has two plural suffixes, *im* and *wt*, and by default masculine nouns take the former and feminine nouns the latter. However, there are many exceptions to this rule, and hence idiosyncratic nouns are lexically specified as to which suffix they take. In the XFST version of the lexicon, this is indicated by concatenating `plural im` or `plural wt` to the lexical representation of (irregular) nouns. Figure 1 depicts the lexical representation of the noun *xlwn* 'window'.

A specialized set of rules implements the morphological processes that apply to each major part of speech. For example, Figure 2 depicts a somewhat simplified version of the rule that accounts for the *wt* suffix of Hebrew nouns. This rule makes extensive use of composition (denoted by '`.o.`') and replace rules ('`->`' and '`<-`'). The effect of this rule is dual: on the surface level, it accounts for alterations in the concatenation of the suffix with the stem (e.g., *iih* becomes *ih*, *wt* changes to *wi*, and a final *h* or *t* are elided); on the lexical level, it changes the specification of *number* from the default *singular* to *plural*.

The rule should be read from the center outwards. The variable `noun` denotes the set of all lexical items whose part of speech is noun. In XFST, a set of words is identified with the identity transduction that relates each word in the set with itself. The first composition on top of the `noun` transduction selects only those nouns whose `plural` attribute is lexically specified as `wt`. Of those, only the ones whose `number` attribute is `singular` are selected. Then, the value `singular` in the lexical

```
define pluralWTNoun [
 [
  [ plural <- singular || number _ ]
  .o. $[number singular]
  .o. $[plural wt]
  .o. noun
  .o. [ i i h -> i h || _ .#. ]
  .o. [ i t -> i || _ .#. ]
  .o. [ w t -> w i || _ .#. ]
  .o. [ [h|t] -> 0 || _ .#. ]
 ] [ 0 .x. [w t] ]
];
```

Fig. 2. XFST account of plural nouns.

(upper) string is replaced by `plural` in the context of immediately following the attribute `number`. In the surface (lower) language, meanwhile, a set of composition operators takes care of the necessary orthographic changes, and finally, the plural suffix *wt* is concatenated to the end of the surface string.

The organization of the rule as a sequence of compositions may be misleading at first blush; in actuality, the rule does not specify a sequence of replacements. Rather, this rule compiles into a single finite-state transducer that encodes all the possible relation pairs licensed by the rule. Specifically, this transducer will license pairs such as *xlwn–xlwnwt* (the default case); *tlwnh–tlwnwt* (final *h* elided); *clxt–clxwt* and *mpit–mpiwt* (final *t* elided); *kmwt–kmwiwt* (*wt* changes to *wi*); and *qniih–qniwt* (*iih* changes to *ih*).

This rule is a good example of how a single phenomenon is factored out and accounted for independently of other phenomena: the rule refers to lexical information, such as '`number`' or '`plural`', but completely ignores irrelevant information such as, say, gender. However, it also hints at how information is manipulated by regular expressions. Since finite-state networks have no memory, save for the state, all information is encoded by strings that are manipulated by the rules. Thus, a simple operation such as changing the value of the *number* feature from *singular* to *plural* must be carried out by the same type of replace rules that account for the changes to the surface form. There is no way to structure such information or encapsulate it, as is common in programming languages; specifically, there is no obvious way to group together features that form a natural bundle, such as agreement features.

### *2.2 A direct implementation*

The alternative to FST is a direct implementation of a morphological analyzer in some general purpose, high-level programming language. Recent proposals in this spirit include the Zen toolkit for morphological and phonological processing of natural languages (Huet 2005), which inspired the better known paradigm of Functional Morphology (Forsberg and Ranta 2004). In both paradigms, rules are

| When input ends in: | *iih* | *it* | *wt* | *h, t* | default |
|---|---|---|---|---|---|
| Replace it by: | *ih* | *i* | *wi* | $\epsilon$ | |
| Then add: | *wt* | *wt* | *wt* | *wt* | *wt* |

Fig. 3. Direct account of plural nouns.

expressed in a functional programming languages (Objective Caml and Haskell, respectively), and their compilation yields a decorated trie that can be efficiently traversed at run-time, resulting in highly efficient lookup.

We directly reimplemented the HAMSAH grammar as a Java program; this is conceptually a variant of the functional morphology paradigm. We used the more common object-oriented programming paradigm, rather than the functional one. The method we used was *analysis by generation*: we first generate all the inflected forms induced by the lexicon and store them in a database; then, analysis is simply a database lookup. It is common to think that for languages with rich morphology such a method is impractical. While this may have been the case in the past, contemporary computers can efficiently store and retrieve millions of inflected forms. Of course, this method would break in the face of an infinite lexicon (which can easily be represented with FST), but for most practical purposes, it is safe to assume that natural language lexicons are finite. We note in passing that the same approach was used for the first morphological analyzer of Hebrew (Shapira and Choueka 1964), and a similar idea (precompilation of all possible prefixes, stems, and suffixes, reducing analysis to a little more than lookup) is used in the current state-of-the-art analyzer of Arabic (Buckwalter 2004).

To separate linguistic knowledge from processing code as much as possible, our Java implementation uses a database of *rules*, which are simple string transductions intended to account for simple (mostly morpheme boundary) morphological and orthographic alterations. When generating inflected forms, the program identifies certain conditions (e.g., a plural suffix *wt* is to be attached to a noun). It then looks up this condition in the rule database and retrieves the action to apply, depending on the suffix of the input string. An example of the rule database, with alterations pertaining to the suffix *wt* (cf. Figure 2), is depicted in Figure 3. For many morphological processes, solutions such as this can accurately stand for linguistic rules of the form depicted in Figure 2.

Note that rules such as the one depicted in Figure 3 are *generation* rules, and must not be confused with the kind of *ad hoc* rules used at run-time, for example, stemming. They fully reflect the linguistic knowledge encoded in finite-state replace rules. Granted, the example rule is simplistic, and more complex phenomena require more complicated representations. For example, Hebrew inflectional morphology involves morpho-phonological alternations mostly along morpheme boundaries. In other languages, phenomena of vowel harmony that can spread across several morphemes may require more sophisticated solutions.

In addition, we preferred to formulate the grammar rules such that phonological (and orthographic) changes go hand in hand with morphological changes. It would have been possible to separate strictly phonological processes from affixation

processes, and such an approach might have yielded a more compact grammar; in our approach, different affixation processes that trigger the same phonological or orthographic process have to duplicate the stipulation of the phonological change. The main reason for this is that this was how the finite-state grammar was designed, and the direct implementation attempted to be as faithful as possible to the original grammar. For our purposes, then, the rule database solution is a reasonable representation. Grammars for other languages may resort to a more sophisticated representation of the rules, either along the lines of functional morphology or as (simple) regular expressions.

The morphological analyzer was obtained by directly implementing the rules and applying them to the lexicon. The number of inflected forms (before attaching prefixes) is 473,880 (>300,000 of those are inflected nouns and close to 150,000 are inflected verb forms). In addition to inflected forms, the analyzer also allows as many as 174 different sequences of prefix particles to be attached to words; separation of prefixes from inflected forms is done at analysis time, since such combinations do not involve any morpho-phonological alternations. In principle, we could have generated all the possible surface forms (including prefixes) off-line, in which case analysis would strictly amount to lookup. The number of forms would then have been approximately 50 million (not all the prefixes combine with all word categories). While this would still be easily manageable with contemporary memory sizes, we acknowledge that for some languages, especially agglutinative ones, such a solution may not be adequate at present, although it may become feasible in the near future with the rapid decrease in the cost of memory.

The direct implementation is equivalent to the finite-state grammar: this was verified by exhaustively generating all the inflected forms with each of the systems and analyzing them with the other system.

## 3 Comparison and evaluation

Having described the XFST benchmark grammar and its direct Java implementation, we now compare the two approaches along several axes. It is important to emphasize that we do not wish to compare the two systems, but rather the methodology, given that our motivation is to identify the strengths and weaknesses of the technology. In particular, we chose XFST, as it is one of the most efficient and certainly the most expressive FST toolbox available. A recent comparison of XFST with the FSA Utilities package (Cohen-Sygal and Wintner 2005) shows that the latter simply cannot handle grammars of the scale of HAMSAH. All experiments were done on a dual 2-GHz processor Linux machine with 2.5 Gb of memory.

### *3.1 Faithfulness*

One of the assets of FST is that it allows for a very accurate implementation of linguistic rules. However, a good organization of the software can provide a clear separation between linguistic knowledge and processing in any programming environment, so that linguistic rules can be expressed concisely and declaratively,

as exemplified in Figure 3. Our conclusion is that while for a linguistically accurate modeling of natural language phenomena, FST remains superior, from an engineering point of view, the two approaches are comparable.

### *3.2 Reversibility*

A clear advantage of FST is that grammars are fully reversible. However, with the analysis by generation paradigm the same holds also for a direct implementation: the generator is directly implemented, and the analyzer is implemented as search in the database of generated forms.

### *3.3 Expressivity*

Here, the disadvantages of FST as a programming environment are clear. Programming with FST is very different from programming in ordinary languages, mainly due to the highly constrained expressive power of regular relations (programmers sometimes feel that they are working with their hands tied behind their backs). Sometimes the expression of morphological phenomena requires more than regular power (as in the case of reduplication); more frequently, regular relations suffice, but grammar designers trained as programmers wish they had access to operators that are not regular. A typical example is the rule that doubles the final consonant of English verbs before some suffixes: one would want to express such a rule as 'change a consonant $C$ to $CC$ when it is in the context of following a consonant and a vowel and preceding a vowel'. With regular expressions, this can only be done by stipulating all the consonants $C$ and their doubled form.

### *3.4 Portability*

XFST is a proprietary package with three versions available for three common operating systems. Other finite-state toolboxes are freer; FSA is open source, but as we noted earlier, it simply cannot cope with grammars the size of HAMSAH. FSM is available for a variety of Unix operating systems, as a binary only, whereas INTEX is distributed as a Windows executable. In contrast, a Java implementation can be delivered to users with all kinds of (contemporary) operating systems and hardware, and is optimally portable. The practical portability limitations directly hamper the utilization of FST in practical, commercial systems: this has been an issue both with the Hebrew HAMSAH and with a morphological grammar for Turkish (K. Oflazer, personal communication 2007)

### *3.5 Abstraction*

Large-scale morphological grammars tend to be extremely *non*-modular. Each surface string is associated, during its processing, with a lexical counterpart that describes its structure. The lexical string is constantly rewritten by the rules, as in Figure 2. There are very limited facilities for grouping together related features, abstracting away from the actual representation, modularizing the grammar, etc.

Since information cannot be encapsulated and the language provides no abstraction mechanisms, collaborative development of finite-state grammars is difficult. All grammar developers must be aware of how information is represented at all times. Furthermore, since the only data type is strings, debugging becomes problematic: very few errors can be detected at compile time. In contrast, a direct implementation benefits from all the advantages of developing in a high-level programming environment.

An additional limitation of XFST is that it only provides two levels of representation, surface and lexical. Consequently, information pertaining to various strata of linguistic representation are coerced into two levels, and cleanup rules must be used as part of the grammar to eliminate intermediate layers that are less important for the final analysis. Consider Figures 1 and 2, which depict orthographic information (*xlwn*), morpho-syntactic features (`number singular`), exception features (`plural wt`), and full semantic glosses, all represented in two levels of strings.

### *3.6 Maintenance*

A byproduct of the nonmodularity of FST grammars is that maintaining them is difficult and expensive. It is hard to find a single person who is knowledgeable in all aspects of the design, and any change in the grammar is painful. A direct implementation in a high-level language provides access to sophisticated debugging mechanisms that are missing altogether from FST frameworks, including XFST. This must be added to the poor compile-time performance (see below), which again hampers maintainability.

### *3.7 Compile-time efficiency*

A major obstacle in the development of XFST grammars is the speed of compilation. Let us first recall a few theoretical (worst case) results. As is well known, many of the finite-state operators result in huge networks: theoretically, composition of networks of $m$ and $n$ states yields a network with $O(m \times n)$ states, and replace rules are implemented using composition. While automata can always be minimized, this is not the case for transducers (Mohri 2000). Theoretically, it is very easy to come up with very small regular expressions whose compilation is intractable. For any integer $n > 2$, there exists an $n$-state automaton $A$, such that any automaton that accepts the complement of $L(A)$ needs at least $2^{n-2}$ states (Holzer and Kutrib 2002). An example of an XFST expression whose compilation time is exponential in $n$ is: `~[ [a|b]* a [a|b]^n b [a|b]* ]`.

Of course, this worst case behavior is not worse than that of a direct implementation in an arbitrary programming language, where programmers are free to write arbitrary code. A more meaningful comparison must be in terms of actual performance. As it turns out, large-scale grammars such as HAMSAH yield temporary networks that are sometimes larger than the available memory, requiring disk access and thereby slowing compilation down dramatically. The complete Hebrew grammar is represented, in XFST, by a network of approximately 2 million

Table 1. *Compilation/generation times (in minutes) when some lexical items change*

| | #items | | |
|---|---|---|---|
| | 360 (adverbs) | 1,648 (adjectives) | 21,400 (all) |
| FST | 13:47 | 13:55 | 48:12 |
| Java | 0:14 | 3:59 | 30:34 |

states and 2.2 million transitions. Compiling the entire network takes more than 48 minutes and requires 3 Gb of memory. To verify that this was not an issue of limited machine memory, we repeated the experiment on a similar machine with 16 Gb of memory (compared with the 2.5 Gb used for all the experiments). This reduced compilation time to 25 minutes.

Compilation time is usually considered a negligible criterion for evaluating system performance. However, when developing a large-scale system, the ability to make minor changes and quickly remake the system is crucial. With XFST, modification of even a single lexical entry requires at least an intersection of (the XFST representation of) this entry with the network representing the rules that apply to it. As a concrete example, adding a single two-character proper name (which does not inflect) to the lexicon increased the size of the network by nine states and ten arcs, but took almost 3 minutes to compile. Adding a two-character adjective resulted in the addition of 27 states and 30 arcs, and took about the same time. A major contribution to this poor performance may be the fact that the features and glosses used for the analyses were all multi-character symbols, which increase the memory requirements of the compiled network.

In the direct implementation, modification of a single lexical entry requires generation of all inflected forms of this entry, which takes a fraction of a second; the time it takes to generate $k$ lexical entries is proportional to $k$ and is independent of the size of the remainder of the system. The analysis program is not altered.

To summarize the differences, Table 1 shows the time it takes to compile a network when $k$ lexical entries are modified, for three values of $k$, corresponding to the number of adjectives, adverbs, and the size of the entire lexicon. This time is compared with the time it takes to generate all inflected forms of these sublexicons in the analysis by generation paradigm.[1]

### *3.8 Run-time efficiency*

While finite-state automata guarantee linear recognition time, this is not the case with transducers, which cannot always be determinized (Mohri 1997). Even when a device can be determinized, the determinization algorithm is inefficient (theoretically, the size of the deterministic automaton can be exponential in the size of its nondeterministic counterpart).

[1] This comparison is slightly biased, since the direct implementation does not produce inflected forms with prefixes, which are produced by XFST.

Table 2. *Time performance of both analyzers (in seconds)*

| | #Tokens | | | |
|---|---|---|---|---|
| | 10 | 100 | 1,000 | 10,000 |
| FST | 1.25 | 2.40 | 12.97 | 118.71 |
| Java + MySQL | 1.24 | 3.04 | 8.84 | 44.94 |
| Java + Hash | 5.00 | 5.15 | 5.59 | 7.64 |

As it turns out, storing a database of half a million inflected forms (along with their analyses) is inexpensive, and retrieving items from the database can be done very efficiently. We experimented with two versions: one uses MySQL as the database and the other loads the inflected forms into a hash table. In this latter version, most of the time is spent on loading the database, and retrieval time is negligible.

We compared the performance of the two systems on four tasks, analyzing text files of 10, 100, 1,000, and 10,000 tokens. The results are summarized in Table 2, and clearly demonstrate the superiority of the direct implementation. In terms of memory requirements, XFST requires approximately 57 Mb of memory, whereas the Java implementation uses no more than 10 Mb. This is not a significant issue with contemporary hardware.

## 4 Discussion

We compared the process of developing a large-scale morphological grammar for Hebrew with FST with a direct implementation of the morphological rules in Java. Our conclusion is that FST remains superior to its alternatives with respect to the true representation of linguistic knowledge, and is therefore more adequate for smaller scale grammars, especially those whose goal is to demonstrate specific linguistic phenomena rather than form the basis of large practical systems. However, viewed as a programming environment, FST suffers from severe limitations, the most significant of which are lack of abstraction and difficulties in incremental processing.

Abstraction is the essence of computer science and the key to software development. Working with regular expressions and developing rules that use strings as the only data structure does not leave much space for sophisticated abstraction. Several works attempt to remedy this problem. XFST itself provides a limited solution in the form of *flag diacritics* (Beesley and Karttunen 2003). These are feature-value pairs that can be added to the underlying machines in order to add limited memory to networks; a similar solution, which is fully worked out mathematically, is provided by *finite-state registered automata* (Cohen-Sygal and Wintner 2006). These approaches are too low level to provide the kind of abstraction that programmers have become used to. A step in the right direction is the incorporation of feature structures and unification into finite-state transducers (Zajac 1998), and in particular the recent proposal to use typed feature structures as the entities on which such transducers operate (Amtrup 2003). More research is needed in order to fully

develop this direction and incorporate its consequences into a finite-state-based grammar development framework.

The problem of incremental grammar development, exemplified in Table 1, can also be remedied by incorporating some recent theoretical results, in particular in incremental construction of lexicons (Daciuk *et al.* 2000; Carrasco and Forcada 2002), into an existing framework. Ordinary programming languages benefit from decades of research and innovation in compilation theory and optimization. In order for FST to become a viable programming environment for natural language morphology applications, more research is needed along the lines suggested here.

## Acknowledgments

This paper extends and revises an earlier version (Wintner 2007). This research was supported by The Israel Science Foundation (grant No. 137/06), the Israel Internet Association, the Knowledge Center for Processing Hebrew, and the Caesarea Rothschild Institute for Interdisciplinary Application of Computer Science at the University of Haifa. I am very grateful to Shlomo Yona for implementing the XFST grammar and to Dalia Bojan for implementing the Java system. I also thank Galia Givaty, Alon Itai, Nurit Melnik, Kemal Oflazer, Yael Sygal, and four anonymous reviewers for useful and constructive comments. The views expressed in this paper as well as all remaining errors are, of course, my own.

## References

Amtrup, J. W. (2003) Morphology in machine translation systems: efficient integration of finite state transducers and feature structure descriptions. *Machine Translation* **18**(3): 217–238.

Beesley, K. R. (1996) Arabic finite-state morphological analysis and generation. In *Proceedings of COLING-96, the 16th International Conference on Computational Linguistics*, Copenhagen.

Beesley, K. R. (1998) Arabic morphology using only finite-state operations. In M. Rosner (eds.), *Proceedings of the Workshop on Computational Approaches to Semitic languages*, pp. 50–57, Montreal, Quebec. COLING-ACL'98.

Beesley, K. R. and Karttunen, L. (2003) *Finite-State Morphology: Xerox Tools and Techniques.* Stanford: CSLI.

Buckwalter, T. (2004) *Buckwalter Arabic Morphological Analyzer Version 2.0.* Philadelphia: Linguistic Data Consortium.

Carrasco, R. C. and Forcada, M. L. (2002) Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics* **28**(2): 207–216.

Chanod, J.-P. and Tapanainen, P. (1996). A robust finite-state grammar for French. In *ESSLLI'96 Workshop on Robust Parsing*, pp. 16–25, Prague.

Cohen-Sygal, Y. and Wintner, S. (2005) XFST2FSA: comparing two finite-state toolboxes. In *Proceedings of the ACL-2005 Workshop on Software*, Ann Arbor, MI.

Cohen-Sygal, Y. and Wintner, S. (2006) Finite-state registered automata for non-concatenative morphology. *Computational Linguistics* **32**(1): 49–82.

Daciuk, J., Mihov, S., Watson, B. W. and Watson, R. E. (2000) Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics* **26**(1): 3–16.

Forsberg, M. and Ranta, A. (2004) Functional morphology. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pp. 213–223, New York: AACM Press.

Görz, G. and Paulus, D. (1988) A finite state approach to German verb morphology. In *Proceedings of the 12th Conference on Computational Linguistics (COLING-88)*, pp. 212–215, Budapest.

Holzer, M. and Kutrib, M. (2002) State complexity of basic operations on nondeterministic finite automata. In *Implementation and Application of Automata (CIAA '02)*, pp. 151–160.

Huet, G. (2005). A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *Journal of Functional Programming* **15**(4): 573–614.

Itai, A., Wintner, S. and Yona, S. (2006) A computational lexicon of contemporary Hebrew. In *Proceedings of The Fifth International Conference on Language Resources and Evaluation (LREC-2006)*, Genoa, Italy.

Johnson, C. D. (1972) *Formal Aspects of Phonological Description.* Mouton, The Hague.

Kanthak, S. and Ney, H. (2004) FSA: an efficient and flexible C++ toolkit for finite state automata using on-demand computation. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL 2004)*, pp. 510–517.

Kaplan, R. M. and Kay, M. (1994) Regular models of phonological rule systems. *Computational Linguistics* **20**(3): 331–378.

Karttunen, L. (1995). The replace operator. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pp. 16–23.

Koskenniemi, K. (1983). *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production.* The Department of General Linguistics, University of Helsinki.

Mohri, M. (1997) Finite-state transducers in language and speech processing. *Computational Linguistics* **23**(2): 269–312.

Mohri, M. (2000) Minimization algorithms for sequential transducers. *Theoretical Computer Science* **234**: 177–201.

Mohri, M., Pereira, F., and Riley, M. (2000) The design principles of a weighted finite-state transducer library. *Theoretical Computer Science* **231**(1): 17–32.

Oflazer, K. (1994) Two-level description of Turkish morphology. *Literary and Linguistic Computing* **9**(2): 137–48.

Roche, E. and Schabes, Y. (eds.) (1997) *Finite-State Language Processing*. Language, Speech and Communication. Cambridge, MA: MIT Press.

Schmid, H. (2005) A programming language for finite state transducers. In *Proceedings of the 5th Workshop on Finite State Methods in Natural Language Processing*, Helsinki, Finland. University of Helsinki.

Shapira, M. and Choueka, Y. (1964) Mechanographic analysis of Hebrew morphology: possibilities and achievements. *Leshonenu* **28**(4): 354–372. In Hebrew.

Silberztein, M. (1993) *Dictionnaires électroniques et analyse automatique de textes : le système INTEX* Paris: Masson.

Trost, H. (1990) The application of two-level morphology to non-concatenative German morphology. In *COLING-90*, pp. 371–376.

van Noord, G. and Gerdemann, D. (2001) An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt and H. Jürgensen (eds.), *Automata Implementation*, number 2214. Lecture Notes in Computer Science. Springer.

Wintner, S. (2007) Finite-state technology as a programming environment. In A. Gelbukh (eds.), *Proceedings of the Conference on Computational Linguistics and Intelligent Text Processing (CICLing-2007)*, vol. 4394. Lecture Notes in Computer Science, pp. 97–106. Berlin and Heidelberg: Springer.

Yona, S. and Wintner, S. (2008). A finite-state morphological grammar of Hebrew. *Natural Language Engineering*.

Zajac, R. (1998) Feature structures, unification and finite-state transducers. In *FSMNLP'98: The International Workshop on Finite-state Methods in Natural Language Processing*, Ankara, Turkey.