A compiler for morphological analysers and generators based on finite-state transducers *

Alicia Garrido, Amaia Iturraspe,
Sandra Montserrat, Hermínia Pastor,
and Mikel L. Forcada

Departament de Llenguatges i Sistemes Informàtics,
Universitat d'Alacant,
E-03071 Alacant, Spain.
E-mail: {alicia,amaia,sandra,herminia}@torsimany.ua.es, mlf@dlsi.ua.es

Abstract

Morphological analyzers and generators are essential parts of many natural-language processing systems such as machine translation systems; they may be efficiently implemented as finite-state transducers. This paper describes a compiler that converts a morphological dictionary (a dictionary augmented with descriptions of the flexive paradigms) into a C program implementing a very compact finite-state transducer that performs the desired morphological analysis or generation task.

1 Introduction

Morphological analysis and generation are essential to many natural-language processing tasks such as machine translation. Morphological analysis reads the inflected *surface form* of each word in a text and writes its *lexical form*, consisting of a canonical form (or *lemma*) of the word and a set of tags showing its syntactical category and morphological characteristics. Generation is the inverse process. Both analysis and generation rely on two sources of information: a dictionary of the valid lemmas of the language and a set of inflection paradigms.

One of the most efficient approaches to morphological analysis and generation uses *finite-state transducers* (FST) (Mohri 1997; Oncina *et al.* 1993; for a review in Spanish see also Alegria 1996), a class of finite-state automata, that will be described in the following. FST may be used used as one-pass morphological analysers and generators and may be very efficiently implemented. Other analysers–generators, of which Pérez Aguiar's (1996) is a complete example for Spanish, use an intuitive pattern-matching approach which tries first to decompose the word in a number of stem–inflection pairs which are subsequently validated. There are a number of tools for the construction of FST-based morphological analysers available, the best known being those developed by Xerox (Karttunen 1994;

*Work supported by the Caja de Ahorros del Mediterráneo.

Karttunen 1993; Chanod 1994).

We have developed an aid to the construction and maintenance of FST-based morphological analysers and generators. It is a compiler that reads a *morphological dictionary* containing a static description of the lemmas and the inflection paradigms and writes a C program that implements a compact FST-based morphological analyser (or generator) performing the task. This allows the linguist to focus on describing the lexicon and morphology of the language in question in a simple format and frees him or her of having to think as a programmer.

2 Finite-state transducers

The morphological analysers and generators are based on finite-state transducers; in particular, we use *letter transducers* (Roche & Schabes 1997). Any finite-state transducer may always be turned into an equivalent letter transducer. A *letter transducer* is defined as $T = (Q, L, \delta, q_I, F)$, where Q is a finite set of states, L a set of transition labels, $q_I \in Q$ the initial state, $F \subseteq Q$ the set of final states, and $\delta : Q \times L \rightarrow 2^Q$ the transition function (where 2^Q represents the set of all finite sets of states).

The set of transition labels is $L = (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$ where Σ is the alphabet of input symbols, Γ the alphabet of output symbols, and ϵ represents the empty symbol. According to this definition, state transition labels may therefore be of four kinds: $(\sigma : \gamma)$, meaning that symbol $\sigma \in \Sigma$ is read and symbol $\gamma \in \Gamma$ is written; $(\sigma : \epsilon)$, meaning that a symbol is read but nothing is written; $(\epsilon : \gamma)$, meaning that nothing is read but a symbol is written; and $(\epsilon : \epsilon)$ means that a state transition occurs without reading or writing. The last kind of transitions are not necessary neither convenient in final FSTs, but may be useful during construction. It is customary to represent the empty symbol ϵ with a zero ("0"). A letter transducer is said to be *deterministic* when $\delta : Q \times L \rightarrow Q$. Note that a letter transducer which is deterministic with respect to the alphabet $L = (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$ may still be non-deterministic with respect to the input alphabet Σ.

A string $w' \in \Gamma^*$ is considered to be a transduction of an input string $w \in \Sigma^*$ if there is at least one path from the initial state q_I to a final state in F whose transition labels form the pair $w : w'$ when concatenated. There may in principle be more than one of such paths for a given transduction; this should be avoided, and is partially eliminated by determinization (see below). On the other hand, there may be more than a valid transduction for a string w (in analysis, this would correspond to *lexical ambiguity*; in generation, this should be avoided). In analysis, the symbols in Σ are those found in texts, and the symbols in Γ are those necessary to form the lemmas and special symbols representing morphological information, such as `<noun>`, `<fem>`, `<2p>`, etc. In generation, Σ and Γ are exchanged.

The general definition of letter transducers is completely parallel to that of nondeterministic finite automata (NFA) and that of deterministic letter transducers, parallel to that of DFA; accordingly, letter transducers may be determinized and minimized (with respect to the alphabet L) using the existing algorithms for NFA and DFA (Hopcroft & Ullman 1979; Salomaa 1973; van de Snepscheut 1993). Transitions labeled $(\epsilon : \epsilon)$ may be eliminated during determinization using a technique parallel to ϵ-closure.

Unlike other compilers like Karttunen's (1993), the compiler described in this paper builds letter transducers having no cycles (transitions form a directed acyclic graph) which, in addition, have a unique final state. The absence of cycles is due to the fact that

only concatenations and alternations are allowed in the morphological dictionary (see section 3)[1]. To minimize the resulting transducer, we use an algorithm described by van de Snepscheut (1993), which has two identical steps which may be summarized as follows: in each step, the transition arrows in the letter transducer are reversed, so that the final state is initial and the initial state is final, and the resulting transducer is determinized with respect to L (that is, new states are formed with sets of old states so that the new δ is $\delta : Q \times L \rightarrow Q$). The transducer resulting from the double reversal–determinization process is minimal. This algorithm is particularly efficient in the case of acyclic letter transducers. Moreover, the two steps have a simple interpretation: the first step joins common endings (finds regularities in suffixes) and the second one joins common beginnings of transductions (finds regularities in prefixes).

FST-based analysers output all possible analyses for a homograph[2].

3 Morphological dictionary

The morphological dictionary is a text file where spaces, tabulators and newlines may be freely inserted for legibility. Any text between "`#`" and the end of line is ignored and may be used as a comment. The dictionary has the following three sections:

1. The symbol declaration section, where output symbols representing morphological features (such as `<fem>` or `<sg>`) are explicitly declared.

2. The paradigm section, where inflection *paradigms* are declared: when a set of lemmas in the dictionary share a common inflection pattern, this pattern may be given a name in square brackets (such as `[verbs_in_ducir]` for Spanish verbs such as *traducir*, *producir*, *inducir*, etc.) and declared in advance so that it may be used in the dictionary. Paradigms may be indefinitely nested, that is, the names of paradigms previously defined may be used to define other paradigms (paradigms are compiled into subtransducers that are then integrated to build the complete transducer). A paradigm is basically a name for a set of alternate *transductions*; the simplest *transduction* is a pair of strings (input and output), such as "`(com:comer<verb>)`", which we call a *couple*; a paradigm name, such as "`[pi1]`", is always a valid transduction; and the concatenation of one or more of transductions, such as "`(am:amar<verb>)[V1C]`" is also a valid transduction[3].

   Couples are treated as follows: if zeroes ("`0`") are used to align explicitly the input and output strings in a couple so that both have the same length, as in "`(am000:amar<verb>)`", zeroes are understood to represent the empty symbol and the alignment is preserved in the transducer; if both strings have different length and contain no zeroes, as in "`(am:amar<verb>)`", then the compiler will align them using a simple heuristic: the shortest string in the couple is completed with trailing zeroes[4]. We have experimentally found this heuristic to work

[1] We plan to add a simple repetition operator to allow for the recognition and analysis of numerical expressions or other variable-lengtht tokens which have a non-finite number of valid forms.

[2] A homograph is a surface form having two or more morphological analyses.

[3] The input (resp. output) string of the concatenation of two simple transductions is the concatenation of their input (resp. output) strings.

[4] An analogous choice is done in Karttunen's (1993) finite-state lexicon compiler.

very well for Spanish; the corresponding experiments are described in section 5. If, for some reason, one of the strings should be empty, a single zero may be used instead, as in "`(0:<3p><sg>)`". If zeroes are used, but the lengths of input and output strings do not match, as in "`(com00:comer<verb>)`", the compiler assumes that the designer of the morphological dictionary intended to supply an explicit alignment but has failed to give a correct one; accordingly, the compiler issues a warning to the effect, ignores the zeroes and uses the heuristic described above.

Note that paradigms do not necessarily have to describe endings (suffixes), because they may be placed anywhere in a transduction.

3. The dictionary section simply a large paradigm containing all the lexical units in the dictionary. Any valid transduction may be an entry in the dictionary; for example, the linguist may have chosen to form a single paradigm "`[tener]`" with all the forms of the irregular Spanish verb *tener* ("to have"); the dictionary entry for *tener* would then simply be the paradigm name, which could be in turn be used to define other entries sharing the same inflective pattern such as *detener* ("to arrest") as "`(de:de)[tener]`" or *contener* ("to contain") as "`(con:con)[tener]`".

Figure 1 shows an example of the format of the morphological dictionary.

4 The compiler

The compiler has been developed under Linux using `bison` (an evolved version of `yacc`, Johnson 1975) and `flex` (an evolved version of `lex`, Lesk 1975) to build a front-end which reads in

```
# Grammatical symbol declaration
%symbol <PresInd>; # Indicative present
%symbol <1p>;      # First person
%symbol <sg>;      # Singular
%symbol <2p>;      # Second person
%symbol <pl>;      # Plural
# ... etc.


# Paradigm definition:

['on] > ('on:<sg>)      #  nouns
       | (ones:<pl>) ;   # -'on

[pi1] > (s:<2p><sg>)  # indicative present
      | (0:<3p><sg>)
      | (mos:<1p><pl>)
      | (n:<3p><pl>) ;

[ii1] > (a:<1p><sg>)  # imperfect endings
      | (as:<2p><sg>)
      | (a:<3p><sg>)
      | (ais:<2p><pl>)
      | (an:<3p><pl>) ;

[pron1pl] # Enclitics after verb
       | (l:'el<pron><ac><3p>)[pl_fem]
       | (teme:
          tu<pron><2p><MF><sg>+yo<pron><1p><MF><sg>)
       | (le:'el<pron><dat><3p><MF>)[pl_vocal]
#      (...)
       | (te:tu<pron><2p><MF><sg>+)[pron_'el]
#      (...)


[V1C] > (o:<PresInd><1p><sg>)   # first conjugation
      | (a:<PresInd>)[pi1]
      | ('ais:<PresInd><2p><pl>)
      | (ab:<ImpInd>)[ii1]
      | ('abamos:<ImpInd><1p><pl>)
#        ... etc.
      | (emos:<Imper><1p><pl>)
      | ('emos:<Imper><1p><pl>+)[pron1pl]
#        ... etc.

[v1c] | (e:<Imper><3p><sg>+)[pron1pl]  # first conj.
      | (a:<Imper><2p><sg>+)[pron1pl]  # with accent
      | (en:<Imper><3p><pl>+)[pron1pl] ; # in root.


# Dictionary
%dic

(am:amar<verb>)[V1C];          # amar
('am:amar<verb>)[v1c];
(com:comer<verb>)[V2C];        # comer
(c'om:comer<verb>)[v2c];
(acci:acci'on<noun><fem>)['on]     # acci'on
```

Figure 1: Sample morphological dictionary

the morphological dictionary file and combines the partial transducers corresponding to the declared paradigms into a single transducer containing one initial and one final state using $(\epsilon : \epsilon)$ transitions as "glue" where convenient. Error messages are designed to help the linguist correct possible errors in the format of the morphological dictionary file being compiled (such as paradigms or symbols used but not defined, mismatched parentheses, etc.). The back-end minimizes the resulting transducer (as described in section 2) and combines the resulting code with a standard skeleton to produce a C program which is ready to be used on its own or included in a larger application such as a machine translation system.

5 Experiments

We have performed experiments to evaluate the heuristic used by the compiler to align string couples when the linguist who has constructed the morphological dictionary decides not to supply an explicit alignment. To that effect, a linguist constructed a morphological dictionary D_1 for 3 099 of the most frequent Spanish words which included all the nominal, pronominal, adjectival and verbal flexive paradigms of the Spanish language. She used her linguistic knowledge to align explicitly all couples in the dictionary using zeroes where necessary. A second morphological dictionary D_2 was created from D_1 by removing all zeroes. Each one of these morphological dictionaries was compiled into an analyser (A_1 and A_2 respectively). The number of states of A_1 (5 574) and A_2 (5 589) was almost indistinguishable. We also used a small corpus of Spanish legal texts containing 798 801 words[5] to study the *number of live hypotheses per letter*, that is, the number of partial transductions, or, equivalently, the number of states alive per input letter. This is a measure of non-determinism with respect to input symbols and therefore of time complexity. The results for A_1 and A_2 are almost indistinguishable: average 4.0 and standard deviation 2.5 for A_1 (linguistically motivated alignment) and average 3.9 and standard deviation 2.5 for A_2 (automatic alignment). Speeds observed lie in the range of 20 000 words per second on a Pentium running at 400 MHz.

[5] The coverage of the 3 099-word dictionary on this corpus was 66.8%.

6 Concluding remarks

A compiler to automatically build finite-state-transducer-based morphological analysers and generators from morphological dictionaries has been described. This tool may be of great interest when building natural-language processing systems such as machine translation programs. When the linguist does not supply an explicit alignment between surface forms and lexical forms, the compiler uses a simple heuristic to produce an alignment that has been experimentally shown to be equally efficient. We are currently testing an extended version the program which determinizes and minimizes the complete transducer into a p-subsequential letter transducer (Mohri 1997) which behaves exactly as the minimal deterministic finite automaton with respect to the input alphabet Σ; according to the experimental results presented in the previous section, we expect to obtain speeds on the order of 100 000 words per second on state-of-the-art desktop workstations.

Acknowledgements: We thank Francisco Moreno-Seco and Rafael C. Carrasco for their suggestions and help.

References

Alegria, Iñaki. 1996. Morfología de estados finitos. *Procesamiento del Lenguaje Natural* 18:1–26.

Chanod, Jean-Pierre. 1994. Finite-state composition of French verb morphology. Technical Report Technical Report MLTT-005, Xerox Research Centre Europe, Meylan, France.

Hopcroft, J. E., & J. D. Ullman. 1979. *Introduction to automata theory, languages, and computation*. Reading, MA: Addison–Wesley.

Johnson, S.C. 1975. Yacc – yet another compiler compiler. Technical Report Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J.

Karttunen, Lauri. 1993. Finite-state lexicon compiler. Technical Report Technical Report ISTL-NLTT-1993-04-02, Xerox Palo Alto Research Center, Palo Alto, California.

——. 1994. Constructing lexical transducers. In *Proceedings of COLING-94*, volume 1, 406–411, Kyoto, Japan.

Lesk, M.E. 1975. Lex — a lexical analyzer generator. Technical Report Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J.

Mohri, Mehryar. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics* 23(2):269–311.

Oncina, Jose, Pedro García, & Enrique Vidal. 1993. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15:448–458.

Pérez Aguiar, José R., 1996. *Reconocimiento y generación integrada de la morfología del español: una aplicación a la gestión de un diccionario de sinónimos y antónimos*. Facultad de Informática, Universidad de Las Palmas de Gran Canaria dissertation.

Roche, E., & Y. Schabes. 1997. Introduction. In *Finite-State Language Processing*, ed. by E. Roche & Y. Schabes, 1–65. Cambridge, Mass.: MIT Press.

Salomaa, Arto. 1973. *Formal Languages*. New York, NY: Academic Press.

van de Snepscheut, J.L.A. 1993. *What computing is all about*. New York: Springer-Verlag.