Introduction

The engine implements a finite-state transducer (FST) - a special kind of finite-state automata where transition arcs are labeled by pairs of symbols. Such an automaton can be used to translate a sequence of symbols into another sequence. The basic idea is straightforward: we read symbols from input one by one and match each symbol against the first or the second element of a pair that labels an arc. If a match succeeds, we leap to a node at the other end of the arc, and add the other (unmatched) pair element to the output sequence. If we can reach the terminal state of the automaton when the input exhausts, then the result sequence collected so far contains the translation of the string.

It is natural to apply this technique to morphology. The advantages are:

- Efficiency of analysis: the search time is virtually linear in number of characters in the input, and presents only a slight dependence on the number of states/arcs in the automaton;
- Inherent reversibility: by changing the matching side (i.e. matching input to the second element of the label pair instead of first), we obtain an engine that performs a reverse conversion - i.e. generation instead of analysis.

There are also some drawbacks:

- The compilation of the automaton is a complex algorithmic task that requires a lot of memory space and is very computation-intensive. This may also lead to difficulties in maintaining the morphological descriptions.

This paper gives a general discussion algorithms used in implementing a morphological component based on FST technique, without relation to any specific programming language used for implementing it.

ALGORITHM DESCRIPTION

All processing (both in compilation and in analysis) can be clearly split into two parts:

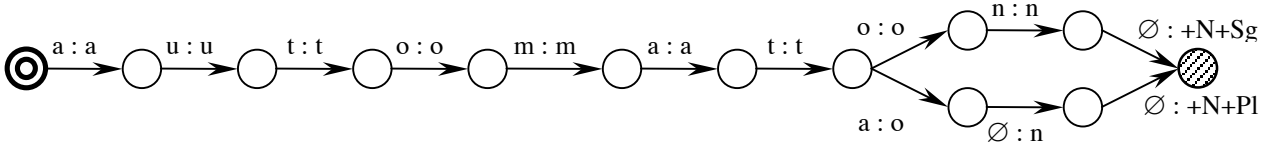
A. FST implementation. An FST subsystem operates on plain sequences of two-byte integers. The subsystem comprises a compiler engine for building an FST and saving it in a serialized form, and a walker engine for retrieving the serialized FST and performing lookup operations on it. This component is never accessed directly: it is wrapped into an external layer of morphological compiler interface.

B. Morphological compiler interface. This component handles conversion from string representation of words/features to integers understood by FST, providing all top-level interfaces.

A. FST Implementation

A.1. Lookup (analysis / synthesis)

Here's a graph representation of a sample automaton:



Nodes in the graph correspond to states of the automaton. In what follows, we often refer to states as nodes.

Arcs in the graph correspond to transition rules between states. Each arc has a label consisting of an ordered pair of symbols.

To analyse a word form using this graph, we trace a path from the *start node* (a double circle on a graph diagram) to the *end node* (a filled circle) in such a way that the first elements of labels of arcs belonging to the path yield the analysed word form. (Note that some labels may have null elements — these are denoted by ∅ on the sample diagram). A sequence of second elements from these arcs forms the result of analysis.

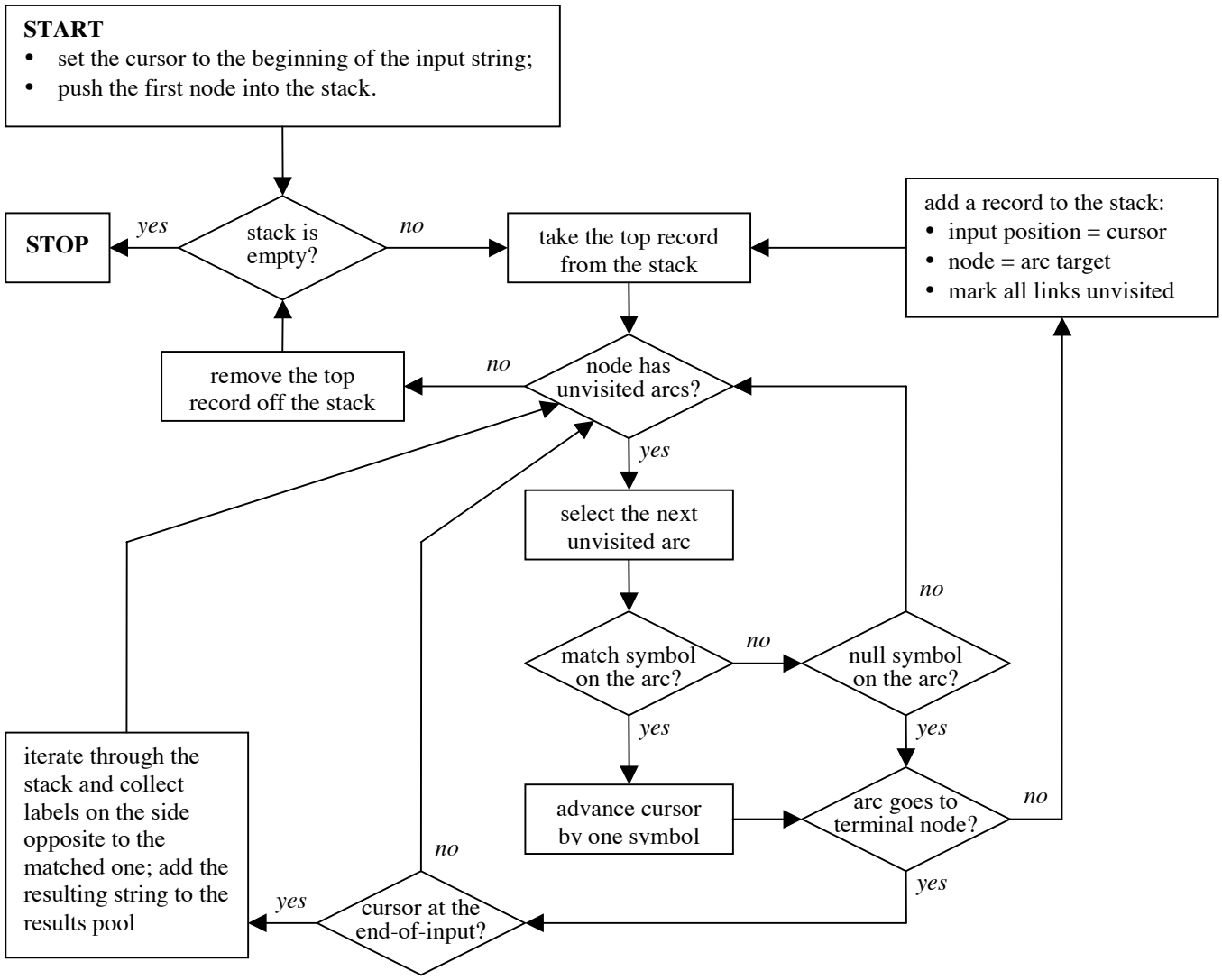
To generate a word form from a lemma and a set of features, we invert the procedure — match the input string against the second element of the label pair, and collect first elements to the output. The same data can be used for both analysis and generation.

Matching the input against arc labels can be done in a number of ways — it is a standard graph-walking problem, easily solvable in a computation-efficient manner. Described below is a specific algorithm implemented in the engine.

The algorithm uses an auxiliary data structure — a *configuration stack*. It is a stack of records, each record having the following fields:

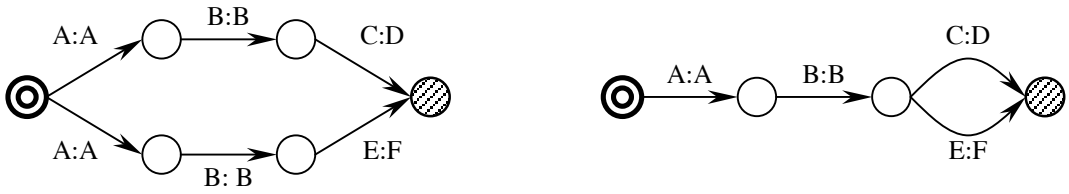
- current node in a graph;
- current arc going out from that node;
- current position in the input string (*cursor*).

The procedure is depicted by a block diagram below:



A.2. Compilation

It is easy to build an automaton that encodes an arbitrary mapping function between finite sets of strings: for every corresponding pair, we can simply trace a separate path from the start node to the terminal one. For instance, to encode a correspondence `{ABC ~ ABD; ABE ~ ABF}`, we may draw a simple graph shown on the left diagram below. However, it is clearly not an optimal one — the automaton shown on the right produces the equivalent matching, while being 50% smaller:



Building an optimal automaton is not a straightforward task. In out approach, the automaton is produced in two steps:

I. *Data acquisition* phase consists in building a loose automaton similar to the left one. We take the pairs to encode and add relative paths to the automaton. Naturally enough, we cannot afford building a totally unabridged automaton (for evident memory consumption reasons). Therefore, we try to add as few nodes as possible, merging the starting and the ending trait of the path:

   - if there is a path in the automaton that starts from the starting node and coincides with the initial trait of a newly added path, and if this path contains no nodes with two or more incoming arcs, then the initial trait of the new path can be merged with this path;
   - if there is a path in the automaton that ends in the terminal node and coincides with the final trait of a newly added path, and if this path contains no nodes with two or more outgoing arcs, then the final trait of the new path can be merged with this path.

   Conditions imposed on the incoming/outgoing links ensure that no extra pairings are produced by a newly added path: in each point of the automaton, possible continuations must either be already present in the system, or derive from the newly added path.

II. *Minimization* consists in reducing the number of arcs and nodes in the automaton by applying a series of transducer-function-preserving transformations. In the current version, only the most straightforward optimization criteria is applied:

   - if two nodes have sets of outgoing arcs such that for each arc in one set, there is an arc in the other pointing to the same node and having the same label, then we can redirect all incoming arcs of the first node to the second one and erase the first node and all its outgoing links.
   - if two nodes have sets of incoming arcs such that for each arc in one set, there is an arc in the other starting from the same node and having the same label, then we can transfer all outgoing arcs of the first node to the second one and erase the first node and all its incoming links.

   These two criteria are applied to all nodes in the automaton. A recursive algorithm to apply the first criterion is described below. It uses an auxiliary flag attached to each node to mark it as either visited or unvisited.

   1. **Initial operations:** mark all nodes as unvisited except for the terminal one. Then start recursion, selecting the terminal node as current.

2. **Recursion step:**
    2.1. Mark the current node as visited;
    2.2. Collect a set of all nodes such that there exists an arc from it to the current node;
    2.3. Partition nodes in the set into groups by number of outgoing arcs. This partitioning is required only to speed up the procedure;
    2.4. For every two nodes in every group, try to apply the minimization criterion. If the minimization condition holds, merge the two nodes. When the start or terminal node are merged with some other node, the result of fusion should be designated as start or end node, respectively;
    2.5. For each node in the set, repeat the recursion step selecting this node as current.
3. End condition: when all nodes are visited, the procedure stops.

For the second criterion, the procedure is similar, but we depart from the start node and the direction of all arcs is inverted.

The procedure 1–3 is repeated several times for both criteria (in alternated order), until no more nodes can be further merged.

One can ideate more complicated minimization algorithms. However, even this simple one is quite efficient on the automata created for morphological analysis.

B. Morphology Compiler/Analyser

The morphological analyser is a specific application of a general FST engine to the task of morphological analysis. It uses an FST automaton as a low-level engine. The basic units of this level are character strings representing words and their morphological features. There are two distinct operations performed by the morphological component:

- *analysis* — convert a *word form* (represented as a string) to a *lemma* (the ground form of the word, a string) and a set of *features* (a list of character strings);
- *generation* is the reverse procedure: from a lemma and a set of features, reconstruct a word form.

B.1. Lookup (analysis / synthesis)

The lookup procedure comprises the following steps:

1. Convert the input data to a sequence of symbols used as labels in the underlying automaton. The rules are as follows:
    - each character in the word form and in the lemma corresponds to a separate symbol. All character encoding conversions are handled in this point;
    - features are treated as atomic entities: each feature corresponds to one symbol. The mapping between features and their correspondent symbols is managed by a special substitution table, called *feature directory*.
2. Look up the resulting symbolic sequence in the automaton, obtaining a set of possible translated sequences. These sequences are also made of automaton symbols.
3. For each of the translated sequences, convert symbols back to characters/features, using the same feature directory.

The only difference between analysis and generation consists in the choice of the matching side in the automaton: by matching on the other side, we reverse the transformation.

B.2. Compilation

The compilation procedure involves creating two separate objects:

- an automaton ;
- a feature map.

The engine proceeds by adding word form entries one by one. For each entry, the word form, the lemma, and the feature set are converted into sequences of symbols in the same way as described above. New features encountered during the processing are assigned a new code, and added to the feature directory. Then we build two sequences:

- a sequence corresponding to the word form, further referred to as *surface-side* sequence;
- a sequence corresponding to the concatenation of lemma and its associated features – *lexical-side* sequence.

After that, we build a sequence of symbol pairs by putting the surface and lexical sequences side by side, complementing the shorter sequence by padding zeros.

One important issue concerns the *relative alignment* of the lexical-side and surface-side sequences. It would be sufficient to append padding zeros to the end of sequence; however, in case of stem alternations this favors proliferation of bizarre symbol pairings. To increase chances of successful minimization, we have adopted a smarter alignment strategy that takes morphological reasons into consideration. The procedure is as follows:

- first, we try to find an optimal position of the lemma with respect to the word form, adding leading zeros to either the lemma or the word form, to maximize the overlaying segment. This turned out to be necessary for handling languages where prefixes are used to express morphological information. For example, to align a German pair *gesetzt : setzen*, we add two leading null characters at the beginning of the first string: [*g e s e t z t*] : [∅ ∅ *s e t z e n*]; this makes the relation between the two strings be similar to that in other pairs like *übersetzt : übersetzen*, so that these paths could be further merged in minimization;
- then, we split the segments past the last coinciding symbol in both strings, and align them by adding zeros to the beginning. In the above example, this yields [*g e s e t z* ∅ *t*] : [∅ ∅ *s e t z e n*];
- finally, we add a sequence of feature codes to the lexical side, and fill the correspondent part on the surface side by zeros: [*g e s e t z* ∅ *t* ∅ ∅] : [∅ ∅ *s e t z e n* +Verb +PP]

When all desired forms are merged into the automaton, the minimization starts. After the minimization, the automaton and the feature map are saved.