ProNTo_Morph: Morphological Analysis Tool for use with ProNTo (Prolog Natural Language Toolkit)

Jason G. Schlachter
Artificial Intelligence Center
The University of Georgia
Athens, Georgia 30602-7415 U.S.A.
http://www.ai.uga.edu

May 8th 2003

1 Introduction

The paper describes ProNTo_Morph, a morphological parsing tool for English, written in ISO Prolog and fully compatible with SWI-Prolog version 5.0.10 (see site for download http://www.swi-prolog.org).

The program was developed by extending and modifying the program, Part Of English Morphology, developed by Dr. Covington at the University of Georgia. The source code for this program, poem.pl, is downloadable at his web site *http://www.ai.uga.edu/mc*.

Morphological parsing is the process of breaking a word into its smallest meaningful components, or morphemes. Our program, ProNTo_Morph, is concerned with breaking a word into its root and su□x.

For example, the word *harder* is made up of two morphemes, *hard* and *-er*. In the ideal situation, ProNTo_Morph would be given the input string:

```
harder
```

and the program would output a listing of morphemes such as:

```
[hard,-er]
```

Unfortunately, morphological parsing is not always this simple. There are many spelling rules that may break a given word into different morphemes. Since a computer algorithm can not know which of these is the correct way, it must have the ability to explore alternative solutions.

The following example help to will illustrate this idea. Let's assume that ProNTo_Morph is configured to output one alternative at a time, and it is given the input string

```
harder
```

then the output would be as follows:

```
[harder]
```

This is not the correct morphological parsing. If the program is forced to backtrack to the next untried alternative, it will return the following solution:

```
[harde,-er]
```

This is still incorrect. If the program is forced to backtrack a second time, it returns the following list:

```
[hard,-er]
```

This is the correct solution. If the program was forced to backtrack again it would fail because there are no more untried alternative.

The above example illustrates how ProNTo_Morph behaves in a non-deterministic way to return alternative solutions upon backtracking. If you would prefer that it returns all possible morphological parsings at once, it can also be called in a deterministic manner.

The next section of this paper will describe the design of the program and user configurable options. The third section of this paper documents the callable predicates of ProNTo_Morph and provides example input and output for clarity.

2 Program design

ProNTo_Morph can be executed in a deterministic or nondeterministic manner. If you ask the program for one parsing of a word, the program is nondeterministic and can backtrack to find untried alternatives. The only exception to this is when the word being parsed is in the irregular word list. In this case, the program will return the parsing specified by the irregular word list and will block backtracking. After all, if a word is irregular, there is no point in applying general spelling rules.

The following subsections describe the input and output types that ProNTo_Morph allows.

2.1 ProNTo_Morph accepts input in three forms

1. It can accept a list of tokens from *et.pl* (E□cient Tokenizer).

   ```
   [w([t,e,s,t,i,n,g]),w([t,h,e]),w([t,o,k,e,n]),w([l,i,s,t])]
   ```

   It also accepts any single word structure from that list.

   ```
   w([t,e,s,t,i,n,g])
   ```

2. It can accept character lists:

   ```
   [t,e,s,t,i,n,g]
   ```

   and lists of character lists:

   ```
   [[t,e,s,t,i,n,g],[m,o,r,e]]
   ```

3. It can accept atoms.

```
testing
```

and lists of atoms

```
[testing,more,words]
```

2.2 ProNTo_Morph returns solutions in two forms

1. It can output a list of lists where each inner list contains a morphological parsing of a word.

   See examples below:

   ```
   [[harder]]
   ```

   or

   ```
   [[work],[hard,-er]]
   ```

2. It can output a list that contains every backtracking alternative.

   See examples below:

   ```
   [[[harder]], [[harde, -er]], [[hard, -er]]]
   ```

   or

   ```
   [[[[work]]], [[[harder]], [[harde, -er]], [[hard, -er]]]]
   ```

3 User modifiable files

3.1 Spelling rules

There is a Prolog file, *pronto_morph_spelling_rules.pl* that contains all the spelling rules used by ProNTo_Morph to parse words. This can be modified.

You may want to comment out some of these rules to reduce the number of backtracking alternatives because some of them are only used by a small

number of words in the English language. You may also want to add spelling rules that are relevant to your lexicon.

The order of the spelling rules in the file is important. It determines the order in which ProNTo_Morph uses them to create morphological parsings of the input words. You may wish to rearrange them, so that the rules most used by your lexicon are tried first. This may help to reduce the amount of backtracking.

3.2 Irregular word list

What's an irregular word list? It is a list of irregular words that can not be parsed by regular spelling rules. Each entry contains an irregular word and the correct parsing for that word. When the program tries to parse a word, it first looks to the irregular word list, and if it finds the word in that list it returns the correct parsing and blocks backtracking (i.e. it becomes deterministic).

Many of the words in the irregular word list were taken from the irregular word list files of WordNet, a lexical database for the English Language. This program is downloadable at *http://www.cogsci.princeton.edu/ wn/*.

You can modify the irregular list for your purposes (i.e. animal or plant taxonomy) by modifying one of the four irregular word files. There is one for nouns, verbs, adjectives, and adverbs, and they are named as such.

For example, the following is an entry from *pronto_morph_irreg_noun.pl*:

```
irregular_form( children,X,[child,-pl| X ]).
```

In this case,

```
children
```

is the irregular word and

```
[child,-pl]
```

is the list of morphemes. You can add new entries in the same format or remove entries that are not part of your lexicon.

4 User-callable predicates

4.1 morph_tokens(+Tokens,-List)

Converts the output of *et.pl* to a list of morphemes. *Tokens* should be instantiated to a list of tokens from *et.pl* (E□cient Tokenizer) or a single token from that list. The predicate will unify *List* with a list of lists where each inner list contains a morphological parsing of a word.

The predicate is non-deterministic and will backtrack to alternative morphological parsings upon failure. If there are no alternatives the predicate will fail.

Example One

input:

```
w([h,a,r,d,e,r])
```

output:

```
[[harder]]
```

Example Two

input:

```
[w([w,o,r,k]),w([h,a,r,d,e,r])]
```

output:

```
[[work],[hard,-er]]
```

4.2 morph_chars(+Chars,-List)

Converts character lists to a list of morphemes. *Chars* should be instantiated to a character list or to a list of character lists. The predicate will unify *List* with a list of lists where each inner list contains a morphological parsing of a word.

The predicate is non-deterministic and will backtrack to alternative morphological parsings upon failure. If there are no alternatives the predicate will fail.

Example One

input:

```
[h,a,r,d,e,r]
```

output:

```
[[harder]]
```

Example Two

input:

```
[[w,o,r,k],[h,a,r,d,e,r]]
```

output:

```
[[work],[hard,-er]]
```

Although this required backtracking to find the correct parsing.

4.3 morph_atoms(+Atoms,-List)

Converts atoms to a list of morphemes. *Atoms* should be instantiated to an atom or a list of atoms. The predicate will unify *List* with a list of lists where each inner list contains a morphological parsing of a word.

The predicate is non-deterministic and will backtrack to alternative morphological parsings upon failure. If there are no alternatives the predicate will fail.

Example One

input:

```
harder
```

output:

```
[[harder]]
```

Example Two

input:

```
[work,harder]
```

output:

```
[[work],[hard,-er]]
```

4.4 morph_tokens_bag(+Tokens,-List)

This predicate takes the same input as morph_tokens/2, but it will unify *List* with a list that contains every backtracking alternative.

The predicate is deterministic and can not backtrack. If asked to backtrack the predicate will fail.

Example One

input:

```
w([h,a,r,d,e,r])
```

output:

```
[[[harder]], [[harde, -er]], [[hard, -er]]]
```

Example Two

input:

```
[w([w,o,r,k]),w([h,a,r,d,e,r])]
```

output:

```
[[[[work]]], [[[harder]], [[harde, -er]], [[hard, -er]]]]
```

4.5 morph_chars_bag(+Chars,-List)

This predicate takes the same input as morph_chars/2, but it will unify *List* with a list that contains every backtracking alternative.

The predicate is deterministic and can not backtrack. If asked to backtrack the predicate will fail.

Example One

input:

```
[h,a,r,d,e,r]
```

output:

```
[[[harder]], [[harde, -er]], [[hard, -er]]]
```

Example Two

input:

```
[[w,o,r,k],[h,a,r,d,e,r]]
```

output:

```
[[[[work]]], [[[harder]], [[harde, -er]], [[hard, -er]]]]
```

4.6 morph_atoms_bag(+Atoms,-List)

This predicate takes the same input as morph_atoms/2, but it will unify *List* with a list that contains every backtracking alternative.

The predicate is deterministic and can not backtrack. If asked to backtrack the predicate will fail.

Example One

input:

```
harder
```

output:

```
[[[harder]], [[harde, -er]], [[hard, -er]]]
```

Example Two

input:

```
[work,harder]
```

output:

```
[[[[work]]], [[[harder]], [[harde, -er]], [[hard, -er]]]]
```

5 Using ProNTo_Morph with other ProNTo modules

This program has been developed to work with the other modules of ProNTo. It has been fully tested with ProNTo's WordNet program and the E□cient Tokenizer (ET) and there are no known bugs.

The full integration of all components of ProNTo will require some programming in Prolog, however this should be straight forward and require

minimal effort. Of course if you are adding to the system's complexity then it may require a good bit of programming.

Please mention the author of this program and other ProNTo packages if you use them in your work.

References

Covington, Micheal (2003) *Part Of English Morphology* http://www.ai.uga.edu/mc

Fellbaum, Christiane; Langone, Helen; Miller, George A.; Tengi, Randee; Wakefield, Pamela; and Wol□, Susanne (2003e). *WordNet 1.7.1. - a lexical database for the English language.* http://www.cogsci.princeton.edu/~wn.

Wielemaker, Jan (2003). *SWI 5.1 Reference Manual.* University of Amsterdam. Dept. of Social Science Informatics (SWI).