Automatic Creation of a Morphological Processor in Logic Programming Environment[1]

Ilyas Cicekli and Murat Temizsoy
Department of Computer Engineering and Information Science,
Bilkent University, 06533 Bilkent, Ankara, Turkey,
e-mail: ilyas@cs.bilkent.edu.tr,temizsoy@cs.bilkent.edu.tr

Abstract

In this paper, we describe a *two-level processor* which automatically creates a morphological processor from a given set of two-level phonological rules and morphotactic rules. The given two-level phonological and morphotactic rules are automatically converted into Prolog programs which represent a morphological processor for the language in concern. We propose new logical representations for two-level phonological rules, and two-level phonological rules are mapped into these new logical representations which are implemented in Prolog. Morphotactic rules are mapped into a finite state automaton (FA) representing those rules in Prolog. The mechanism described here is applied to Turkish and the results we get are compared with the results of the PC-KIMMO system which is developed for the same purpose. The experiments show that our system, which is in logic programming environment, is more efficient than the PC-KIMMO system.

Keywords: logic programming, program synthesis, morphological analysis, two-level phonology.

1 Introduction

Morphology is the study of word structures in a language, and phonology is the study of sound structure. The morphological analysis, which is a major branch of natural language processing, uses techniques of both morphology and phonology to analyze word structures of a language in a computational environment. There are two different representations for each word: the common representation known as *surface form* and the structural representation known as *lexical form*. For example, the surface form of the English word `cried` is itself, but its lexical form is `cry+ed` which describes that the word consists of the root word `cry` and the suffix `+ed`. Morphological analysis tries to model the relation between these two distinct representations of words.

There are a lot of studies on the morphological analysis of languages to be used in large scale natural language systems [4, 6, 8, 9]. The method proposed by Kimmo Koskenniemi [5] explains the morphological structure of a language using two-level phonological rules which give correspondences between lexical and surface representations of words. Although this method includes phonological constructs in addition to morphological ones, it is known as two-level morphology. Our system and the PC-KIMMO system [2] are two general purpose processors for two-level morphology.

Two-level morphology consists of two parts: *two-level phonological rules* which give correspondences between lexical and surface characters and *morphotactic rules* specifies the order of morphemes, minimal meaningful units, in word structures. For example, the realization of the lexical character `y` in the lexical form `cry+ed` by the surface character `i` in the surface

[1]This work was supported by NATO Science for stability Project Grant TU-LANGUAGE.

form `cried` is specified by a two-level phonological rule. A verb such as `cry` can be followed by the suffix `+ed` in lexical forms. This is specified by a morphotactic rule.

In the PC-KIMMO system, two-level phonological rules are represented by two-way transducers, two finite automatons working in parallel. They also represent morphotactic rules a single finite automaton (FA). In our processor for two-level morphology, phonological rules are converted into Prolog programs which represent logical meanings of those phonological rules. We also convert morphotactic rules into a Prolog program which simulates the FA for those morphotactic rules. These synthesized Prolog programs are used in *recognition* of the lexical forms of words from their surface forms, and *generation* of the surface forms of words from their lexical forms. Our two-level processor is applied to two-level morphological description of Turkish [9], and our results are compared with the results of the PC-KIMMO system in Section 5.

The rest of this paper is organized as follows. Section 2 and Section 3 describe structures of phonological and morphotactic rules in more detail, respectively. Section 4 explains how phonological and morphotactic rules are converted into Prolog programs. Section 5 compares our results with the results of the PC-KIMMO system, and Section 6 concludes the paper.

2 Two-Level Phonological Rules

A two-level phonological rule states a certain correspondence between a lexical character and a surface character. These rules are bi-directional, i.e. the representation in one level can be obtained from the representation in other level in both directions. The process of finding the surface level representation of a word from its lexical level representation is known as *generation*, and the process of finding the lexical level representation of a word from its surface form is known as *recognition*. These two processes can be exemplified by the example given in Figure 1. In that example, lexical characters `c`, `r`, `e` and `d` correspond to surface characters `c`, `r`, `e` and `d`, respectively. On the other hand, the lexical character `y` corresponds to the surface character `i`, and the lexical character `+` corresponds to a null surface character.

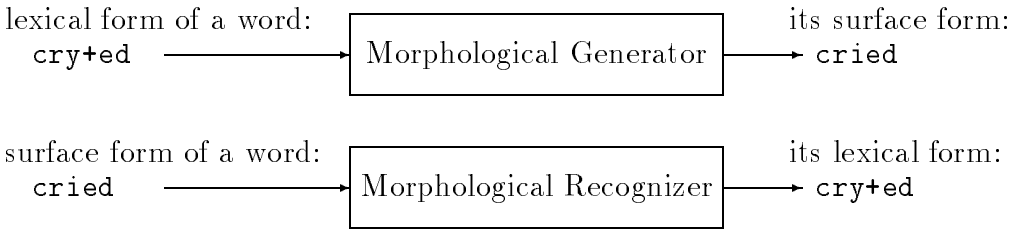


Figure 1: Morphological Generation and Recognition

A two-level rule can be defined by three components: a *correspondence* between a lexical and a surface character, an *environment* (or context) in which the correspondence is valid, and a *rule operator* that defines the relation between the correspondence and the environment. It is denoted as:

$$LexicalCharacter : SurfaceCharacter\ RuleOperator\ LeftContext\ \_\_\ RightContext$$

Some of two-level rules have only a correspondence between a lexical and surface character, and they are known as *default rules*. For example, `y:y` is a default rule which defines a normal correspondence between the lexical character `y` and the surface character `y`, and the

lexical character `y` can be realized by the surface character `y` if other rules with the lexical character `y` do not block this realization. The *environment* is separated into two parts: *left-context* defines the constraints that must be satisfied by correspondences before this correspondence, and *right-context* defines the constraints that must be satisfied after this correspondence. The semantics of the rule operators are defined as follows:

$\Rightarrow$ The correspondence only, but not always occurs in that environment. For example, the rule `t:c` $\Rightarrow$ `__ i:i` states that the lexical character `t` corresponds to the surface character `c` only preceding the correspondence `i:i`, but not necessarily always in that environment. Thus, other realizations of `t` may be found in that environment such as the default realization of `t` by `t`.

$\Leftarrow$ The correspondence always, but not only occurs in that environment. For example, the rule `t:c` $\Leftarrow$ `__ i:i` states that the lexical character `t` has to correspond to the surface character `c` preceding the correspondence `i:i`, but not necessarily only in that environment. Thus, the correspondence `t:c` may occur in other environments but the lexical character `t` must always be realized by the surface character `c` in that environment.

$\Leftrightarrow$ The correspondence always occurs in that environment, and it only occurs in that environment. For example, the rule `t:c` $\Leftrightarrow$ `__ i:i` states that the lexical character `t` has to correspond to the surface character `c` preceding the correspondence `i:i`, this correspondence is not possible in other environments, and other realizations of the lexical character `t` are invalid in that environment.

$/\Leftarrow$ The correspondence never occurs in that environment, and generally used to cover exceptions in a more general rule. For example, the rule `t:c` $/\Leftarrow$ `__ i:i` states that the lexical character `t` cannot correspond to the surface character `c` preceding the correspondence `i:i`.

3 Morphotactic Rules

The second component of two-level morphology for a language contains morphotactic rules. One of the important observations about natural languages is that the morphotactic rules of a language can be represented as a FA, and this is exploited in two-level morphology. A morphologically complex word can be obtained from its morphemes by following legal translations in the FA representing morphotactic rules, and attaching the current morpheme to the end of string of morphemes. So, if we have the morpheme stream `en+rich` in a state S_i, then there must be a transition with the morpheme `+ment` to another state S_j to get the noun `en+rich+ment` in a FA representing morphotactics of English. Another observation that can be made is that the same state transitions can be used for `en+large+ment`, so lexicals (morphemes including root words) can be divided into classes that morphotactically behave alike to reduce the complexity of the FA for a language. Often, these subclasses reflect major word classes such as nouns, verbs and adjectives. The last observation is that there are some exceptional cases in any language which should be handled explicitly. For example, there is no word like `en+small+ment` in English.

The morphotactic rules of a language determine the structure and the ordering of morphemes. A state of the FA for morphotactic rules represents a subclass of words in that language, and we can move from one state to another with an arc labeled with a morpheme. For example, Figure 2 gives a FA to describe a small subset of the morphotactic rules for

adjectives in English. Here, we have two special states *BEGIN* and *END* to mark initial and final states of that FA. We can move from state *BEGIN* to state *ADJROOT* by consuming an adjective root. In order to make a move from the state *ADJROOT* to state *END*, we have to consume one of morphemes `+er` or `+est`, or we have to make an empty transition. This means that we can recognize adjective roots or adjectives ending with morphemes `+er` and `+est`.

Figure 2: A Simple FA for English Adjectives

4 Logical Representation of Rules

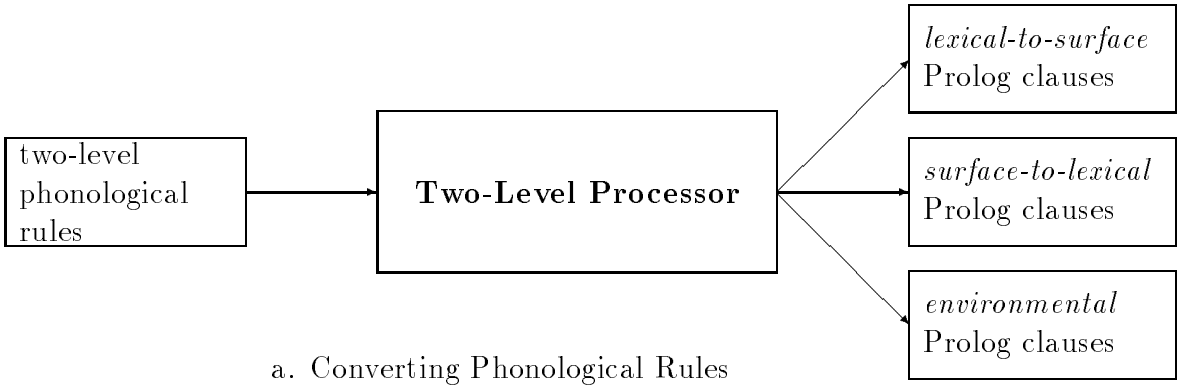
Our *two-level processor* takes two-level phonological rules and morphotactic rules for a language, and produces Prolog clauses representing these rules. These Prolog clauses are used as parts of a morphological processor. Some of these clauses are used in the *recognizer* of the morphological processor, and some of them in the *generator*. There are also clauses used in both the *recognizer* and *generator*. Figure 3 describes the inputs and the outputs of our *two-level processor*.

Since two-level phonological rules can be represented as a two-way transducer, they can be separated into two FAs that work in parallel. Similarly, the produced Prolog clauses corresponding to the two-level phonological rules are separated into two groups: *lexical-to-surface* and *surface-to-lexical*. The *lexical-to-surface* clauses are used in the *generator*, and the *surface-to-lexical* clauses are used in the *recognizer*. In addition to these two groups, the *two-level processor* produces the Prolog clauses corresponding to environmental restrictions in two-level phonological rules. These clauses are used by both *lexical-to-surface* and *surface-to-lexical* clauses.

From the morphotactic rules, the *two-level processor* creates a lexico-graphic tree for root words, and a set of Prolog clauses representing the FA defined by these morphotactic rules. Both the lexico-graphic tree and the FA for morphotactic rules are used only by the *recognizer*.

The *generator* consists of the produced Prolog clauses which represent phonological rules and a small hand-written Prolog code (cf. Figure 4). The *generator* takes the lexical representation of a word and produces its surface representation. For each character in the lexical level, the generator tries to find a corresponding surface character by using the produced Prolog clauses representing phonological rules. The correspondence between a lexical character and a surface character must satisfy the environmental constraints given in the phonological rules.

The *recognizer* consists of the produced Prolog clauses which represent phonological rules and morphotactic rules, and a small hand-written Prolog code (cf. Figure 4). The recognizer also contains the lexico-graphic tree created for root words. The *recognizer* takes the surface representation of a word and finds all possible lexical representations of that word by using the root and the suffix knowledge of the language in concern. The recognizer first tries to find a root word by searching the lexico-graphic tree and checking constraints imposed by the phonological rules. Then, it tries to add morphemes to this root word.



a. Converting Phonological Rules

b. Converting Morphotactic Rules

Figure 3: Two-Level Processor Creating A Morphologic Processor

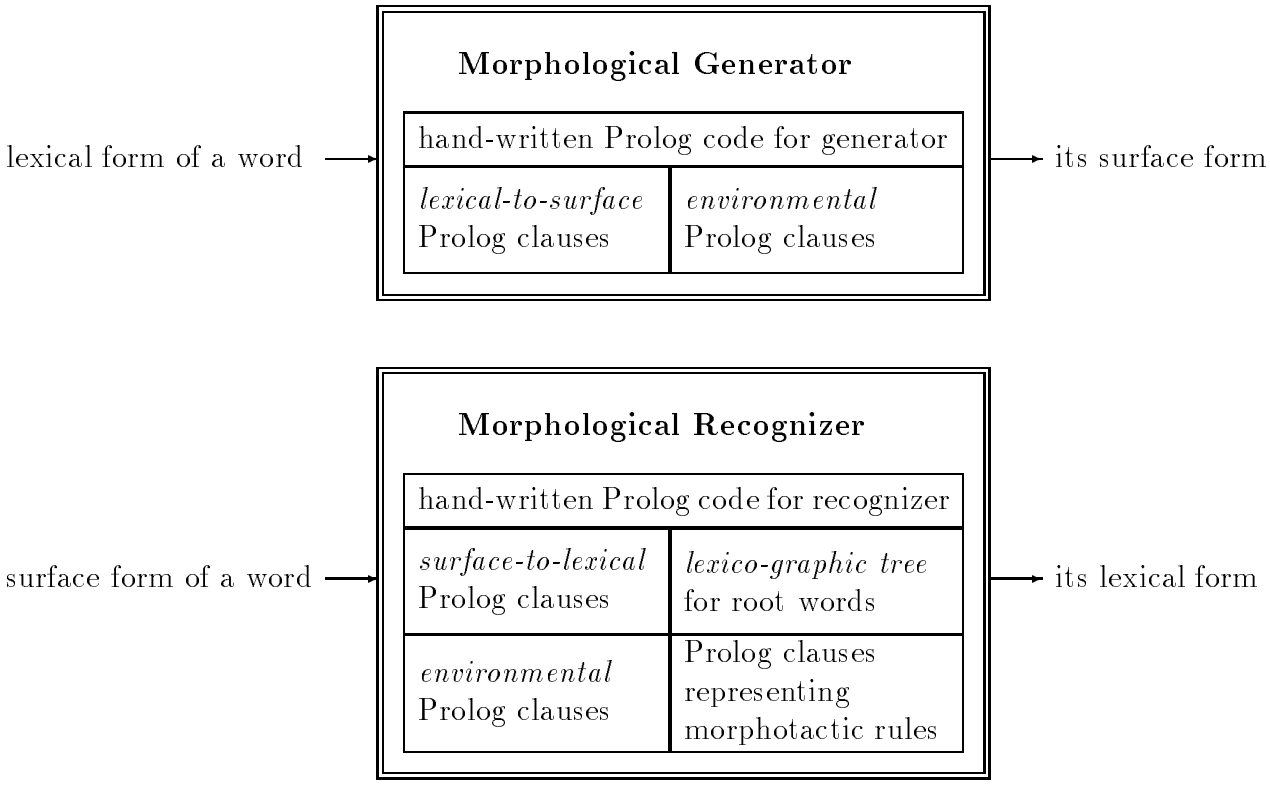


Figure 4: Produced Morphologic Processor

4.1 Representation of Phonological Rules

From the definitions of the rule operators in two-level phonology, operators $\Leftrightarrow$ and $\Leftarrow$ must have a higher priority than the operator $\Rightarrow$ since they define an *always* condition for an environment. So, when a correspondence with an operator $\Leftrightarrow$ or $\Leftarrow$ is legal in some context, the realizations from other rules with the operator $\Rightarrow$ and default correspondences for the same lexical character must be avoided. In order to achieve this in the *lexical-to-surface* part, the rules with operators $\Leftrightarrow$ and $\Leftarrow$ must be checked first, and if they succeed, other correspondences must be blocked for that environment. For example, the following rules define two correspondences of the lexical character `t` for a particular language:

`t:c` $\Leftrightarrow$	*left-context* `_` *right-context*	a special correspondence of `t` to `c`.
`t:t`		a default correspondence of `t` to `t`.

These two rules are represented by Prolog clauses corresponding to the following if-construct if they are the only rules for the lexical character `t`.

if (*left-context* **and** *right-context*) holds
  then `t` is realized by `c`
  else `t` is realized by `t`

If conditions in a rule with the operator $/\Leftarrow$ are satisfied, the correspondence imposed by that rule must be rejected. For example, consider the following rules:

`t:c` $\Rightarrow$	*left-context*$_1$ `_`	*right-context*$_1$	a special correspondence of `t` to `c`.
`t:c` $/\Leftarrow$	*left-context*$_2$ `_`	*right-context*$_2$	a special limiting correspondence of `t` to `c`.

These two rules are represented by Prolog clauses corresponding to the following if-construct:

if ((*left-context*$_1$ **and** *right-context*$_1$) **and not** (*left-context*$_2$ **and** *right-context*$_2$)) holds
  then `t` is realized by `c`

Now, let us assume that the following two-level phonological rules are the only rules for the lexical character `t`:

1. `t:a` $\Leftrightarrow$ lc_1 `_` rc_1
2. `t:b` $\Leftarrow$ lc_2 `_` rc_2
3. `t:c` $\Rightarrow$ lc_3 `_` rc_3
4. `t:c` $/\Leftarrow$ lc_4 `_` rc_4
5. `t:d` $\Rightarrow$ lc_5 `_` rc_5
6. `t:d` $\Rightarrow$ lc_6 `_` rc_6
7. `t:t`

These rules are represented by the following five Prolog clauses[2]. Each clause realizes the lexical character `t` with a different surface character. A subgoal `rule_i` in those clauses checks whether the conditions given in left and right contexts of the i^{th} rule hold in the environment `Env` or not. These environment restriction clauses (`rule_i`) are also used by the *surface-to-lexical* part.

```
lex_to_sur_t(~a,Env) :- rule_1(Env), !.
lex_to_sur_t(~b,Env) :- rule_2(Env), !.
lex_to_sur_t(~c,Env) :- rule_3(Env), not(rule_4(Env)).
lex_to_sur_t(~d,Env) :- ( rule_5(Env) ; rule_6(Env) ).
lex_to_sur_t(~t,Env).
```

[2]In ALS Prolog, `~c` means that the ASCII code of the character `c`.

First two clauses may block the realizations of the lexical character `t` with the surface characters different from `a` and `b` if the environmental constraint of the first rule (`rule_1`) or the second rule (`rule_2`) is satisfied.

The simple representation of rule operators in the *lexical-to-surface* part does not work in the *surface-to-lexical* part. So, we generate different Prolog clauses to represent two-level phonological rules in the *surface-to-lexical* part. Since there may be more than one lexical character corresponding to a surface character, each correspondence should be checked to see whether it is appropriate in that environment or not. For example, let us assume that the following four rules are only rules for surface characters `a` and `b`, and lexical characters `a`, `b` and `t`.

1. `t:a` $\Leftrightarrow$ $lc_1 \_ rc_1$
2. `t:b` $\Rightarrow$ $lc_2 \_ rc_2$
3. `a:a`
4. `b:b`

These rules are represented by the following Prolog procedures in the *surface-to-lexical* part. Each procedure represents the correspondences of a surface character, and each clause of a procedure represents the correspondence of that surface character with a lexical character.

```
sur_to_lex_a(~t,Env) :- rule_1(Env).
sur_to_lex_a(~a,Env).
sur_to_lex_b(~t,Env) :- rule_2(Env), not(rule_1(Env)).
sur_to_lex_b(~b,Env).
```

Although the first rule with the correspondence `t:a` looks like it has nothing to do with the process of realizing the surface character `b` with the lexical character `t`, it plays a key role in that process. The reason for this is that the first rule enforces that the lexical character `t` must be realized by the surface character `a` in the environment given in that rule. So, the third Prolog clause must check that the environment does not hold before it can realize the surface character `b` with the lexical character `t`. In general, before a surface character S can be realized by a lexical character L, the conditions in the rules with the correspondence containing lexical character L and the operator $\Leftrightarrow$ or $\Leftarrow$ should not hold.

Example: In this example, we give a set of two-level phonological rules, and some of the corresponding Prolog clauses in the *lexical-to-surface* part and the *surface-to-lexical* part. Now, let us assume that the following six two-level rules are given:

1. `t:a` $\Leftrightarrow$ `i:i _ j:j`
2. `t:b` $\Rightarrow$ `_` or(`m:m`,`n:n`)
3. `t:b` $/\Leftarrow$ `_ p:p`
4. `t:t`
5. `a:a`
6. `b:b`

The conditions given in the left and right contexts of the environments of the first three rules are represented by the following Prolog clauses. The subgoal `previous_lex_char` (`previous_sur_char`) checks whether the given character is the previous lexical (surface) character in the given environment or not. Similarly, the subgoals `next_lex_char` and `next_sur_char` checks the next lexical and surface characters.

```
rule_1(Env) :-
   previous_lex_char(Env,~i), previous_sur_char(Env,~i),
   next_lex_char(Env,~j), next_sur_char(Env,~j),
rule_2(Env) :-
   ( next_lex_char(Env,~m), next_sur_char(Env,~m) ) ;
   ( next_lex_char(Env,~n), next_sur_char(Env,~n) ).
rule_3(Env) :-
   next_lex_char(Env,~p), next_sur_char(Env,~p).
```

The following three Prolog procedures are produced in the *lexical-to-surface* part for three different lexical characters in the rules above, and each clause of a Prolog procedure represents the realization of that lexical character with a different surface character.

```
lex_to_sur_t(~a,Env) :- rule_1(Env), !.
lex_to_sur_t(~b,Env) :- rule_2(Env), not(rule_3(Env)).
lex_to_sur_t(~t,Env).
lex_to_sur_a(~a,Env).
lex_to_sur_b(~b,Env).
```

In addition to these clauses, our two-level processor also produces a dispatcher procedure `lex_to_sur`, and each clause of this procedure calls one of the procedures given above depending on the incoming value of the lexical character. For example, this procedure contains the following clauses.

```
lex_to_sur(~a,SurChar,Env) :- lex_to_sur_a(SurChar,Env).
lex_to_sur(~b,SurChar,Env) :- lex_to_sur_b(SurChar,Env).
lex_to_sur(~t,SurChar,Env) :- lex_to_sur_t(SurChar,Env).
```

Since there are three different surface characters in the two-level rules, we also have three Prolog procedures in the *surface-to-lexical* part.

```
sur_to_lex_a(~t,Env) :-  rule_1(Env).
sur_to_lex_a(~a,Env).
sur_to_lex_b(~t,Env) :- rule_2(Env), not(rule_3(Env)), not(rule_1(Env)).
sur_to_lex_b(~b,Env).
sur_to_lex_t(t,Env) :- not(rule_1(Env)).
```

In addition to these clauses, a dispatcher procedure `lex_to_sur` is also produced, and each clause of this procedure calls one of the procedures given above depending on the incoming value of the surface character. For example, this procedure contains the following clauses.

```
sur_to_lex(~a,LexChar,Env) :- sur_to_lex_a(LexChar,Env).
sur_to_lex(~b,LexChar,Env) :- sur_to_lex_b(LexChar,Env).
sur_to_lex(~t,LexChar,Env) :- sur_to_lex_t(LexChar,Env).
```

4.2 Representation of Morphotactic Rules

The morphotactic rules of a language consists of two things: a set of root words and a FA. Each root word is a pair of its lexical representation and its class. A class of a root word can be verb, noun, adjective, etc., and it describes the next state in the FA after that root word is consumed. We compile the given FA into a set of Prolog clauses which represents that FA. The set of root words is converted into a lexico-graphic tree. For example, we create a lexico-graphic tree for adjective words and the following simplified Prolog clauses for the morphotactic rules given as a FA in Figure 2.

```
state_BEGIN(Env,NewEnv) :-
   get_root_word(Env,TempEnv,NextState),
   NextState = adj_root,
   state_ADJROOT(TempEnv,NewEnv).
state_ADJROOT(Env,NewEnv) :-
   ( Morpheme = [~+,~e,~r] ;
     Morpheme = [~+,~e,~s,~t] ;
     Morpheme = []) ),
   consume_morpheme(Morpheme,Env,TempEnv),
   state_END(TempEnv,NewEnv).
state_END(Env,Env) :-
   empty_lex_right_context(Env),
   empty_sur_right_context(Env),
```

The environment variable `Env` contains four different values for the lexical and surface representations of a word:

- `lex_left_context` which is a list of consumed lexical characters.
- `lex_right_contex` which is a list of unconsumed lexical characters.
- `sur_left_context` which is a list of consumed surface characters.
- `sur_right_context` which is a list of unconsumed surface characters.

Before the procedure `state_BEGIN` is entered, `sur_right_context` in `Env` contains the surface representation of the word, and all others are empty lists. The procedure `get_root_word` gets a lexical form of root word corresponding to surface character from the lexico-graphic tree. At the same time, it returns the subclass of that root word which will be the next state and it properly updates the environment information. When this procedure searches the lexico-graphic tree for a root word, it also checks the phonological constraints by calling the procedure `sur_to_lex` for each consumed surface character. When the procedure `get_root_word` returns, the values in the new environment `TempEnv` are:

- `lex_left_context` contains the lexical form of the root word.
- `lex_right_context` contains the lexical characters which are enforced by the phonological rules applied during the recognition of the root word.
- `sur_left_context` contains the surface form of the root word.
- `sur_right_context` contains the unconsumed surface characters of the word.

The procedure `consume_morpheme` tries to consume the given lexical characters in the given environment by checking phonological constraints. The procedures `empty_lex_context` and `empty_sur_context` check whether all lexical and surface characters are consumed or not, respectively. If the FA accepts the given surface representation of the word, the procedure `state_BEGIN` returns its lexical form in `left_lex_context` of the new environment variable `NewEnv`.

5 Implementation and Results

Our system is developed by using ALS Prolog [1], and tested with two-level morphology for Turkish which is givenin [9]. This system is used as a part of the ongoing natural language processing project which aims to develop computational environments for Turkish. Turkish [7, 10] is one of the languages having very complex phonological and morphotactic rules. It is an agglutinative language with word structures formed with productive affixations of derivational and infectional suffixes to root words. Morphemes can be added to a root word or a stem to convert a word from a nominal structure to a verbal structure, or vice-versa. Phonological rules constrain and modify the surface realizations of morphological constructions. For example, a vowel in the affixed morpheme has to agree with the preceding vowel to achieve vowel harmony, and consonants may be deleted or realized with different characters in certain circumstances. The following structure[3] of Turkish word `duyulamıyormuş` which can be translated into English as `"(it is said that) it could not be heard"` may give a flavor of the complexity of phonological and morphotactic rules for Turkish.

Surface Form	:	`duyulamıyormuş`
Lexical Form	:	`duy+Hl+yAmA+Hyor+ymHş`
Structure	:	`duy+PASS+NEGC+PR-CONT+NARR`

This word is constructed from the root word `duy` (`hear`) and four morphemes which are affixed according to Turkish morphotactic rules. According to the vowel harmony rules stated by Turkish phonological rules, the following realizations are done: the lexical character `H` of `+Hl` is realized by the surface character `u`; the first `A` of `+yAmA` is realized by `a` and the second `A` is dropped; `H` of `+Hyor` is realized by `ı`; `y` of `+ymHş` is dropped and `H` is realized by `u`.

After feeding Turkish morphotactic rules which include 23,000 root words, and 22 two-level phonological rules to our two-level processor, a morphological recognizer and a morphological generator for Turkish are automatically produced. We tested the produced recognizer and generator, which are Prolog programs, with various data to ensure the correctness of rule mapper algorithms and to check the efficiency of the produced code. For example, we create two test data: one of them contains lexical forms of 1200 words and the other one contains surface forms of 800 words. We test our system and the PC-KIMMO system with this data to compare them. To generate surface forms of 1200 lexical forms takes 1 second in our system when it runs on a Sun sparc station. On the other hand, PC-KIMMO takes 9 seconds for the same data on the same machine. To recognize lexical forms of 800 surface forms, takes 14 seconds in our system and 85 seconds in the PC-KIMMO system. From these results and other test data we used, we can conclude that our produced generator is 9 times faster than the generator of the PC-KIMMO system and our recognizer is 6 times faster than the recognizer of PC-KIMMO.

The reason that our system is faster than the PC-KIMMO system can be explained as follows. The PC-KIMMO system compiles two-level phonological rules into two-way transducers. For each two-level rule, they create a two-way transducer. When they find a correspondence between a lexical character and a surface character, they have to make a move in each transducer even though the rule corresponding to that transducer may have nothing to do with that correspondence. For example, they produce 22 transducers for phonological rules for Turkish, and they make a move in each of these 22 transducers for a correspondence. In other words, they update all transducers for a single character to leave the transducers in proper positions. On the other hand, we do not map phonological rules

[3]Notation: `PASS` means passive; `NEGC` means negative capability; `PR-CONT` means present continues tense; and `NARR` means narrative.

into transducers. We designed a logical representation for phonological rules as described in Section 4.1, and they are represented by Prolog programs. When we find a correspondence, we only execute Prolog clauses corresponding to phonological rules with lexical and surface characters in that correspondence. In our system, all phonological rules are not involved in the process of finding a correspondence, only the related ones are considered.

6 Conclusion

In this paper, we present a two-level processor which takes phonological rules and morphotactic rules as input and automatically produces a morphological processor in logic programming environment. The phonological rules are mapped to their proposed logical representation and morphotactic rules are mapped into a FA. Produced Prolog programs are used as a morphological analyzer or generator. Our system is much more efficient than the PC-KIMMO system which is developed for the same purpose.

One of the main contributions of this paper is the new logical representation of two-level phonological rules. The older method for representing two-level phonological rules was to use two-way transducers which was employed by the PC-KIMMO system. The results of our system shows that our logical representation gives much better results than the method used in the PC-KIMMO system. We tested our system with the rules for Turkish, and we are planning to test it with the rules for other languages such as English and French.

References

[1] *ALS Prolog User's Guide and Reference Manual*, Applied Logic Systems Inc., 1991.

[2] Antworth E. L., *PC-KIMMO: A Two-Level Processor for Morphological Analysis*, Summer Institute of Linguistics, Occasional Publications in Academic Computing, No:16, Dallas, Texas, 1990.

[3] Karttuen L., *KIMMO: A General Morphological Processor*, Texas Linguistic Forum, Vol. 22:163-186. 1983.

[4] Karttuen L., and Wittenburg K., *A Two-Level Morphological Analysis of English*, Texas Linguistic Forum, Vol. 22:217-228. 1983.

[5] Koskenniemi K., *Two-Level Morphology: A General Computational Model for Word Form Recognition and Production*, Technical Report 11, Department of General Linguistics, University of Helsinki, 1983.

[6] Koskenniemi K., An Application of Two-Level Model to Finish, Technical Report, Department of General Linguistics, University of Helsinki, 1985.

[7] Lewis G. L., *Turkish Grammar*, Oxford University Press, 1991.

[8] Lun S., *A Two-Level Morphological Analysis of French*, Texas Linguistic Forum, Vol. 22:271-278. 1983.

[9] Oflazer K., *Two-Level Description of Turkish Morphology*, Literary and Linguistic Computing, Vol. 9, No. 2, 1994.

[10] Underhill R., *Turkish Grammar*, The MIT Press, 1976.