 [Translated from French to English - www.onlinedoctranslator.com](https://www.onlinedoctranslator.com/en/?utm_source=onlinedoctranslator&utm_medium=docx&utm_campaign=attribution)

TP2 - Magic words

The objective of this practical work is to implement in Prolog both a solver and a problem generator in the context of the combinatorial puzzle of magic words.

# Useful information

It is possible that in the present document viewed in GitLab, the display of certain mathematical symbols is sensitive to the internet browser used. Refer to the pdf version of this document if applicable.

The TP is due on Tuesday December 12, 2023 before 11:59 p.m.

## Constituent files of the TP

* README.md — THE subject ofTP.
* README.pdf — The pdf version of the topic.
* MagicWords.pl — The skeleton Prolog le to complete.
* Tests.pl — The Prolog le containing tests. This le imports

MagicWords.pl.

## Change history

* 2023-11-14 — Topic published.

# Definitions

A word is a finite sequence of strictly positive integers such as (1, 2, 1, 3, 2). Here we will note the words more concisely by simply juxtaposing their letters. Thus, the preceding word is noted 12132. The empty word, which is the only word of length 0, is noted ϵ. A factor of a word w is a word w′ such that there exist two words v1 and v2 satisfying w = v1w′v2. When v1 = ϵ, the factor w′ is a prefix of w. For example, the words 213 and 32 are factors of 12132. The words ϵ, 12 and 121 are prefixes of 12132.

A rewriting rule is a pair r = (u1, u2) of words. A word v1 is rewritten by r into a word v2 if v1 = w1u1w2 and v2 = w1u2w2 where w1 and w2 are words. This property is denoted v1 ⇒r v2. In other words, v1 ⇒r v2 if there exists a factor u1 of v1, which, replacing it with u2, gives the word v2. For example, by setting r = (12, 212), we have

312312 ⇒r 3212312

And

312312 ⇒r 3123212.

If R is a set of rewriting rules, we denote by v1 ⇒R v2 the fact that there exists a rewriting rule r ∈ R such that v1 ⇒r v2.

Given a set R of rewriting rules, a word w is R-magic if there exists an integer ℓ and a sequence of words s = (w0, w1, …, wℓ) such that

* w0 = w,
* for all i ∈ {1, ▷ ▷ ▷ , ℓ}, wi−1 ⇒R wi,
* wℓ = ϵ.

In other words, this means that it is possible to completely erase the word wby successively applying rewritings which come from rewriting rules of R.

The sequence s of words is a witness to the fact that w is R-magic and the integer ℓ is the

*length*of witness s.

For example, let R = {r1, r2, r3} with

* r1 = (11, 1),
* r2 = (12, 212),
* r3 = (21, ϵ).

We have

1211 ⇒R 121 ⇒R 2121 ⇒R 21 ⇒R ϵ.

This shows that the word w = 1211 is R-magic. The length of this witness is

4. Note that there are other witnesses to the fact that w is R-magic. Wefor example, we have

1211 ⇒R 21211 ⇒R 211 ⇒R 21 ⇒R ϵ

inlength 4 also and

1211 ⇒R 21211 ⇒R 221211 ⇒R 2211 ⇒R 21 ⇒R ϵ

in length 5.

An R-magic word is perfect when it admits a single witness.

# Description of the puzzle

An instance of the magic word puzzle is specified by a pair (R, w) where R is a set of rewriting rules and w is an R-magic word. A solution of (R, w) is a witness to the fact that w is R-magic.

# Programming

Now let's describe the expected work. The main location choices will also be clarified. The most important predicates to implement will be described.

## Goals

The goal of the TP is twofold:

1. implement a magic word puzzle solver. It must be able to calculate all solutions of a given length for an instance of the puzzle;
2. implement a puzzle instance generator. It must be able to generate, on the input of a set R of rewriting rules and an integer ℓ, an R-magic word whose solution is of length ℓ.

## Main terms

Here isthe implementation specifications relating to the various data structures that appear in this lab.

* + A word is represented by a term word(LST) where LST is the list of integersspecifying the letters of the word. For example

word([1,2,1,1])

is a term that represents the word 1211.

* + A rewriting rule is represented by a term rule(W1, W2) where W1 and W2 are the words that form the two members of the rule. For example,

rule(word([1,2])**,**word([2,1,2]))

is a term that represents the rewriting rule (12, 212).

* + A set of rewrite rules is represented by a term rule\_sset(LST) where LST is a list that contains the set's rewrite rules. For example,

rule\_set( [

rule(word([1,1])**,**word([1])),

rule(word([1,2])**,**word([2,1,2])),

rule(word([2,1])**,**word([]))

]

)

is a term that represents the set of rewriting rules

{(11, 1), (12, 212), (21, ϵ)}.

* + A path is represented by a term path(LST) where LST is the list of words that form the path. For example,

path(

[

word([1,2,1,1]),

word([1,2,1]),

word([2,1,2,1]),

word([2,1]), word([])

]

)

is a term that represents the path

1211 ⇒R 121 ⇒R 2121 ⇒R 21 ⇒R ϵ.

* + A puzzle instance is represented by a term puzzle(RULE\_SET, WORD) where RULE\_SET is a term that represents a set Rrewrite rules and WORD is a magic R-word. For example,

puzzle(

rule\_set( [

rule(word([1,1])**,**word([1])),

rule(word([1,2])**,**word([2,1,2])),

rule(word([2,1])**,**word([]))

]

)**,**

word([1,2,1,1])

)

Easta term that represents the instance of puzzle

({(11, 1), (12, 212), (21, ϵ)}, 1211).

## Obligatory predicates

The following explanations will be based on examples based on the set

rule\_set\_123 of rules and the puzzle\_1 instance of puzzles defined by

rule\_1(rule(word([1,1])**,**word([1])))**.**

rule\_2(rule(word([1,2])**,**word([2,1,2])))**.**

rule\_3(rule(word([2,1])**,**word([])))**.**rule\_set\_123(rule\_set([\_rule\_1,\_rule\_2,\_rule\_3]))**:-**

rule\_1(\_rule\_1)**,**rule\_2(\_rule\_2)**,**rule\_3(\_rule\_3)**.**

puzzle\_1(puzzle(\_rule\_set**,**word([1,2,1,1])))**:-**

rule\_set\_123(\_rule\_set)**.**

The two predicatesfollowing are mandatory to implement.

1. all\_puzzle\_solutions(\_puzzle, \_len, \_solutions)

This predicate relates the puzzle instance \_puzzle, the integer \_len andthe list of paths \_solutions when the puzzle instance in question has as solutions of length \_len exactly the paths of

\_solutions. For example, the test predicate

test\_1**:-**

puzzle\_1(\_puzzle)**,**

\_len is4**,**write("**Puzzle**:\not")**,**print\_puzzle(\_puzzle)**,**nl**,**

format("**Solutions of length**~**d**:\not",[\_len])**,**all\_puzzle\_solutions(\_puzzle**,**\_len**,**\_solutions)**,**print\_path\_list(\_solutions)**.**

has the effect of printing

Puzzle puzzle\_1:

Rule set:

11 -> 1

12 -> 212

21 -> e

Word:

1211

Solutions of length 4:

1211 => 121 => 2121 => 21 => e

1211 => 21211 => 211 => 21 => e

1211 => 21211 => 2121 => 21 => e

1. all\_magic\_words\_of\_rule\_set(\_rule\_set, \_len, \_magic\_words)

This predicate relates the set of rules \_rule\_set, the integer

\_len and the word list \_magic\_words when the words in this list form, with \_rule\_set, all the puzzle instances admitting a solution of length \_len. For example, the test predicate

test\_2**:-**

rule\_set\_123(\_rule\_set)**,**

\_len is3**,**write("**Rule set**:\not")**,**

print\_rule\_set(\_rule\_set)**,**

format("**Magic words having solutions of length**~**d**:\not",[\_len])**,**all\_magic\_words\_of\_rule\_set(\_rule\_set**,**\_len**,**\_magic\_words)**,**print\_word\_list(\_magic\_words)**.**

has the effect of printing

Rule set:

11 -> 1

12 -> 212

21 -> e

Magic words having solutions of length 3: 121

2111

21121

21211

212121

212211

22111

221121

221211

222111

## Intermediate predicates

Here are predicates that are not obligatory to implement but which can greatly helpment in obtaining the two previous primordial predicates.

Let's get started by THE predicates Who provoke of the effects secondaryprinting.

1. print\_word(\_w)

This predicate is true when\_w is a word and prints it.

1. print\_word\_list(\_lst)

This predicate is true when \_lst is a list of words and prints it, with one word per line.

1. print\_rule(\_rule)

This predicate is true when \_rule is a rule and prints it.

1. print\_rule\_set(\_rule\_set)

This predicate is true when \_rule\_set is a set of rules and prints it, with one rule per line.

1. print\_puzzle(\_puzzle)

This predicate is true when the term \_puzzle is an instance of puzzleand print it.

1. print\_path(\_path)

This predicate is true when \_path is a path and prints it.

1. print\_path\_list(\_lst)

This predicate is true when \_lst is a list of paths and prints it, with one path per line.

Here are the other intermediate predicates, which make it possible to represent the mechanics of the game.

1. word\_concatenation(\_w1, \_w2, \_w3)

This predicate connects the three words \_w1, \_w2 and \_w3 when \_w3 isthe concatenation of \_w1 and \_w2.

1. rewrite\_prefix\_by\_rule(\_rule, \_w1, \_w2)

This predicate relates the rule \_rule and the two words \_w1 and \_w2 when \_w2 can be obtained from \_w1 by applying \_rule to one of its prefixes.

1. rewrite\_factor\_by\_rule(\_rule, \_w1, \_w2)

This predicate relates the rule \_rule and the two words \_w1 and \_w2 when \_w2 can be obtained from \_w1 by applying \_rule to one of its factors.

1. rewrite(\_rule\_set, \_w1, \_w2)

This predicate relates the set of rules \_rule\_set and the two words

\_w1 and \_w2 when \_w2 can be obtained from \_w1 by applying a \_rule\_set rule to one of its factors.

1. connecting\_path(\_rule\_set, \_len, \_w1, \_path, \_w2)

This predicate relates the set of rules \_rule\_set, the integer \_len, the word \_w1, the path \_path and the word \_w2 when \_path is a path of length \_len of rewritings making it possible to transform \_w1 into \_w2 using the rules of rewriting of \_rules.

1. puzzle\_solution(\_puzzle, \_len, \_solution)

This predicate putsrelated to the puzzle instance \_puzzle, the integer \_len

and the path \_solution when \_solution is a solution of length

\_len from instance \_puzzle puzzles.

1. rule\_reversion(\_rule\_1, \_rule\_2)

This predicate relates the rules \_rule\_1 and \_rule\_2 when \_rule\_2

is obtained by exchanging the two words constituting \_rule\_1.

1. magic\_word\_of\_rule\_set(\_rule\_set, \_len, \_w)

This predicate relates the set of rules \_rule\_set, the integer \_len and the word \_w when \_w is an R-magic word admitting a solution of length \_len where R is the set \_rule\_set.

## Tips and ideas

* + Write unit tests for all predicates. It is important to ensure that everything works well and that everything is used appropriately (note in particular that there is no real type checking in Prolog).
  + First try to fully understand the mechanism of rewriting words. It is useful to express the application of a rewriting rule on a factor of a word by recursively applying the rewriting on its prefixes.
  + Also try to understand how to detect if a path is a solution. To do this, define the connecting\_path predicate introduced above using a recursive description.
  + It is possible to use the predefined findall predicate to collect all possible solutions for a given query into a list.
  + Along the same lines, it is possible to use the predefined predicatespellto sort a list and remove duplicates. This is useful for improving a list that contains solutions.
  + To generate an R-magic word from a set R of rules, a simple and effective idea consists of considering the set R′ of rules obtainedby reversing the rules of R and generating a word from ϵ by applying rewritings to it using rewriting rules of R′.