Language that Yields Medical Process HAndling

Manual
version 0.1

Rickard Verner Hultgren
rihu0003@student.umu.se

March 18, 2016

Contents

The value of LYMPHA 2
Structure 4
How to read scripts 6
Examples 8

The value of LYMPHA

What should be done in what situation? That is a central question in medicine. The answer is what I call a medical algorithm. Most often through the history of humanity, those algorithms have been a central part of medical scriptures and practice. The construction of the algorithms have usually been fairly simple: When the patient has *these* symptoms and findings (indications), then the condition is called *that*, and treated in *this* way, except for the case of *those* contraindications. The equivalent in computer languages would be the McCarthy formalism: `IF X THEN Y`. A medical practitioner's job has been to examine exactly what indications are present, and then by using algorithms decide what treatment should be done. The examination has often been to sense what is wrong, and to conclude these notions into a medical term. The decision part has often been about to compare indications and contraindications of different treatments as well as with the patient's condition and wishes. In other words, even though the algorithms are clearly formulated, the practitioner's work of getting a feeling of the situation might be a very hard. Therefore the examination by the experienced and confident doctor has always been associated with good care. The role of the physicians using those algorithms, that do this hard work is indispensable, since it has to do with subjective feelings of what is right and wrong, what is true and what is false.

But with more complex algorithms the subjective work of the practitioners subjective feeling is less needed. And this is exactly what has happened during the last decades. With the evolution of diagnostic methods and treatments, even the algorithms have become more complex, e.g. CHADS2 score, Sgarbossa's criteria, Systemic Inflammatory Response Syndrome (SIRS), Advanced Trauma Life Support (ATLS). But with the increase of procedures and complexity of algorithms, not only the possibilities of treatments increase, but also the risk of doing mistakes. And mistakes are done very too often. In order to reduce the amount of mistakes, there have been attempts to involve computers into the process. This have been done as far as I know in three ways:

- In order to decrease the amount of mistakes, solutions where big quantities of information is more accessible the before. The disadvantage with this solution is that there is a lot of irrelevant information that can will disturb the decision-making process. This leads to that the practionar must spend extra time searching and sifting through information. If the time and monetary cost could be unlimited information increasing solutions are a winning concept. Unfortunately finances, the shortage of medical practitioners and their time is a global problem.

- Computer programs, called *expert systems*, are written in order to calculate the complex algorithms, and somewhat replace the practitioner in the decison making process. Different programming languages have been used, e.g. CLIPS, Jess, &c. When writing expert systems there are two approaches, either you write from the perspective of symptoms, or you write the program from the perspective of diseases. The symptom approach is extremely impracticle when it comes to commonly occurrig symptoms, as for example cough, since you have to write a huge program that includes every possible causes. In practice this would mean that for example, when a new drug against lung cancer comes, the every program that handles a symptom that can be caused by lung cancer must be edited. The disadvantage with the other approach is that by using McCarthe formalism, the computer must assume that the patient has every possible illness, and then exclude one after one by diagnostic testing. The traditional strategy among physicians and other health practitioners is to combine those two previously explained strategies. Along the symptoms, a bunch of relevant diagnoses are selected, known as *differential diagnoses*. After that diagnostics are made in order to confirm the different diagnoses. The risk with those two approaches combined is that the practitioner has often a tendency to be too selective when

coming up with differential diagnoses. The two strategies could perhaps successfully be applied in computer-driven algorithms as well, but there are some obstacles:

  - No matter if the McCarthy formalism is used by computers or phycisions, the greatest trap is that it is easy to focus more on defining the condition than doing something. This can have very tragical results, especially in emergent situations. As a result emergency algorithms have been developed, such as Advanced Trauma Life Support (ATLS).
  - The computer could have big problems with selecting relevant differential diagnoses according to a subtle interview.
  - The computer can come up with that many differential diagnoses, so that the cost of the diagnostic part will be too big. This could on the other hand rather easily been solved by prioritization algorithms.

- Machine learning is started to be used i medical data analyzes. I suppose this will replace the need for the intuition of the experienced doctor. I believe that this has a future in examination of complicated diagnostical tests, screening tests, science and medical robotics.

In other words, there are a lot of ways that the computer can be beneficial, but as shown above, there are some deficits. I believe this can be solved with a language that could easily be understood both by health care professionals as well as computers. It would be an expert system focusing on procedures, rather defining causes. Such a language will not only meet the needs of today but will also give the ability to develop decision support systems, automatic decision-making and new ways of understanding medicine. By changing the fundamentals in a language, a system of communication, the basis of how one understands the system, the whole system will change. Let me introduce you to LYMPHA.

Structure

The aim of a LYMPHA script is to make a step by step plan for what medical procedures should be done in the given situation. Everything varies in medicine depending on the circumstances. In LYMPHA everything is therefor presented as data elements that are *variables*, *sets* of variables, or as a sets of *subsets*. Hence there are two *data types*, variables and sets. A data element is presented by writing a word that contains at least one letter in upper or lower case A–Z. Other characters that can be included are numbers 0–9 and the following characters:

```
- _ ?
```

A name of en element without any more information is called an *unspecified data element*. The element becomes a variable if the element equals a real number ($\mathbb{R}$), `NULL` ($\emptyset$) indicating no specified value, or an equation. The valid symbols in equations are `+ - * / ^ ( )`. Each value can be followed by a unit written in brackets (`[ ]`).

```
A_variable = 3[kg] * Another_variable
```

When the unspecified data element equals a set, then it becomes a set:

```
An_empty_set = { }
```

In order to give the data a context elements must be related to each other. There are three ways to relate elements:

- **Followed-by** (`->`) indicates that when a certain data element occurs while reading a script it should be followed by the other given data element:

  ```
  pain -> take_pain_killer
  ```

- **Comma** (`,`) between two elements of data indicates that no additional relation between the two elements is added:

  ```
  myopia, humerus_fracture
  ```

- A data element can be a part of the other. This is expressed by a set:

  ```
  back_pain = {lower_back_pain, upper_back_pain}
  ```

  In the example above the elements of the set were separated by a comma and hence not followed by each other. But elements of a set can also be followed by one another

  ```
  operation = {preoperative_phase -> anesthetic_induction -> surgery
  -> post-operative_care}
  ```

In order to evaluate a given situation it is necessary by make conditional statements. In LYMPHA such a statement is made by comparing two elements using *relational operators*:

relational operator	*read as*
==	if and only if (≡)
>	greater than
>=	greater than or equal (≥)
<	lesser than
<=	less than or equal (≤)
!=	not equal to (≠)

The two data elements that are connected by a relational operator are merged into one data element called a statement that is either true (T) or false (F). Through set-builder notation either true or false statements will be extracted from the set. In this way the number of elements in a set can be reduced. In the example below the set has only one element:

```
lower_back_pain = 1,
upper_back_pain = 0,
back_pain = {T:
  lower_back_pain == 1,
  upper_back_pain == 1,
},
```

This set will have different numbers of elements depending on how many true statements there are in the set. The number of elements of the set is declared by putting a vertical bar (|) before and after the name of the set. This can be made into a conditional statement as follows:

```
|back_pain| == 1 -> treatmentX,
|back_pain| == 0 -> treatmentY,
```

How to read scripts

A language would be useless if there is no way to understand it. Therefore I below present a schematic for interpretation. The main goal is to calculate a proposal for the next step in a patientâĂŹs investigation. In future, a prioritization feature is planned. Before reading a script, one must decide the following things:

- Where to start reading. There are two alternatives:
  - Specify starting points for reading.
  - Use objects that has no previous object linked to them.
- How many steps should be read? The followig alternatives are available:
  - Specify how many steps should be read.
  - Read until the script ends.

The process of interpretation LYMPHA scripts works as follows:

1. All the objects are stored into a workig memory.
2. Linking the objects together.
3. The nested linked list is gone through.

Pseudocode for each of those steps is presented below. Important to keep in mind is that a *followed-by* relation is just valid inside the set it is written in. If it is written outside all sets it is called *global* and is valid in all sets in the script. In order to easily write LYMPHA-based programs in other languages than Python, I try to use a general approach to the pseudocode.

The code parts that follow is licensed under a free-BSD license.

ALL THE OBJECTS ARE STORED INTO A WORKIG MEMORY.

By the use of REGEX, data objects will be made. Each object has the capacity of pointing to multiple previous-node-objects as well as next-node-objects. Additionaly a list will be made, that stores pointers to all the objects.

```
p = ( head-of-object-list→next-object-list )
if REGEX(object found) then
   object = malloc(sizeof(struct node))
   ( p→next-object-list ) = object
   p = object
end if
```


LINKING THE OBJECTS TOGETHER.

An object is linked to a previous-node-object by pointing to the object that has name of the registered previous-node-objects. This is also done through REGEX. The procedure is done the same way with the next-node-objects.

```
for object in object-list do
  for previous-object in object-list do
    if REGEX(previous-object) ≡ previous-object then
      object→previous = previous-object
      if no head nodes are specified then
        remove object from head-node-list
      end if
    end if
  end for
  for next-object in object-list do
    if REGEX(next-object) ≡ next-object then
      object→next = next-object
    end if
  end for
end for
```

THE NESTED LINKED LIST IS GONE THROUGH

After examining every object in this way, pointers to the head-nodes will be stored in a list call first-row. While going through each object, pointers the next-node-objects of each object in the first-row-list will be stored in a list called second-row. After going through every object in the first-row-list will be deleted. Then the second-row-list will be renamed to first-row and a new empty second-row-list will be made.

```
while first-row do
  for object in first-row do
    for next-object in next-node-objects do
      add pointer to next-object in second-row
    end for
  end for
  delete first-row
  rename second-row first-row
end while
```

The first reading software in the LYMPHA project is planned to be an interpreter written in Python. This interpreter's goal is to make data objects out of the data elements in a script. While running the script the names of the objects can be printed into stdout. External functions are called through plugins. The format depends on what kind of program is in question. It might be data tables in a database, a data structure or a data object.

Examples

The first example shows the definition of low blood pressure and that this indicates fluid therapy. The second example is BMI. The third and fourth examples show two ways of writing the the SIRS criteria.

Example 1

```
hypovolemia {T:
  systolicBP < 90[mmHg],
  diastolicBP < 60[mmHg],
},
fluid_therapy = {
  insertPVC ->
  mountfluid
},
|hypovolemia| > 0 -> fluid_therapy,
```

Example 2

```
BMI = mass/(length^2),
```

Example 3

```
respiration = {T:
  resprate > 20[breaths/min],
  pCO2 < 4.3[kPa],
},
SIRScriteria = {T:
  HR > 90[BPM],
  temp < 36[C],
  temp > 38[C],
  |respiration| >= 1,
  WBC < 4000[cells/mm3],
  WBC > 12000[cells/mm3],
},
|SIRScriteria| == 2 -> SIRS,
```

Example 4; This says exactly the same as exmple 3.

```
respiration = {T:
  resprate > 20[breaths/min],
  pCO2 < 4.3[kPa],
},
|{T:
  HR > 90[BPM],
  temp < 36[C],
  temp > 38[C],
  |{T:
      resprate > 20[breaths/min],
      pCO2 < 4.3[kPa],
  }| >= 1,
  WBC < 4000[cells/mm3],
  WBC > 12000[cells/mm3],
}| = 2 -> SIRS,
```