01 – Introduction to symbolic system – the count model

August 30, 2016

1 Example: Count model

1.1 Introduction

This introduction shows the construction of the count model, discussed as the first model in LispACT-R. It is assumed that the reader is familiar with ACT-R. An introduction to ACT-R can be found in the paper "An integrated theory of the mind", which is available on the ACT-R website at: http://act-r.psy.cmu.edu/?post_type=publications&p=13623. Further details can also be found in LispACT-R units.

We import the package and create the relevant model (the model is in detail explained below).

```
In [1]: import pyactr as actr

        #Each chunk type should be defined first.
        actr.chunktype("countOrder", ("first", "second"))
        #Chunk type is defined as (name, attributes)

        #Attributes are written as an iterable (above) or as a string, separated by comma:
        actr.chunktype("countOrder", "first, second")

        counting = actr.ACTRModel()

        #this creates declarative memory
        dm = counting.DecMem()

        dm.add(actr.chunkstring(string="\
            isa countOrder\
            first 1\
            second 2"))
        dm.add(actr.chunkstring(string="\
            isa countOrder\
            first 2\
            second 3"))
        dm.add(actr.chunkstring(string="\
            isa countOrder\
            first 3\
            second 4"))
        dm.add(actr.chunkstring(string="\
            isa countOrder\
            first 4\
            second 5"))

        #creating buffer for dm
        retrieval = counting.dmBuffer(name="retrieval", declarative_memory=dm)
```

```
#creating goal buffer
g = counting.goal(name="g")

actr.chunktype("countFrom", ("start", "end", "count"))

#production rules follow; using productionstring, they are similar to Lisp ACT-R

counting.productionstring(name="start", string="""
    =g>
    isa countFrom
    start =x
    count None
    ==>
    =g>
    isa countFrom
    count =x
    +retrieval>
    isa countOrder
    first =x""")

counting.productionstring(name="increment", string="""
    =g>
    isa     countFrom
    count       =x
    end         ~=x
    =retrieval>
    isa     countOrder
    first       =x
    second      =y
    ==>
    =g>
    isa     countFrom
    count       =y
    +retrieval>
    isa     countOrder
    first       =y""")

counting.productionstring(name="stop", string="""
    =g>
    isa     countFrom
    count       =x
    end         =x
    ==>
    ~g>""")

#adding stuff to goal buffer
g.add(actr.chunkstring(string="isa countFrom start 2 end 4"))
```

The model has a method "simulation", which creates a discrete event simulation. This can be run to produce the output of the simulation (trace of the model).

In [2]:
```
sim = counting.simulation()
sim.run()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: start')
(0.05, 'PROCEDURAL', 'RULE FIRED: start')
(0.05, 'g', 'MODIFIED')
(0.05, 'retrieval', 'START RETRIEVAL')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'NO RULE FOUND')
(0.1, 'retrieval', 'CLEARED')
(0.1, 'retrieval', 'RETRIEVED: countOrder(first=2, second=3)')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: increment')
(0.15, 'PROCEDURAL', 'RULE FIRED: increment')
(0.15, 'g', 'MODIFIED')
(0.15, 'retrieval', 'START RETRIEVAL')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
(0.2, 'retrieval', 'CLEARED')
(0.2, 'retrieval', 'RETRIEVED: countOrder(first=3, second=4)')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: increment')
(0.25, 'PROCEDURAL', 'RULE FIRED: increment')
(0.25, 'g', 'MODIFIED')
(0.25, 'retrieval', 'START RETRIEVAL')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: stop')
(0.3, 'retrieval', 'CLEARED')
(0.3, 'PROCEDURAL', 'RULE FIRED: stop')
(0.3, 'retrieval', 'RETRIEVED: countOrder(first=4, second=5)')
(0.3, 'g', 'CLEARED')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'NO RULE FOUND')
```

1.2 Breaking the model into parts

The model is an instance of ACTRModel class. The model makes use of three parts: a declarative memory module, the goal module and procedural knowledge.

Declarative memory is an instance of class DecMem (DecMem allows an optional argument, the set –or dictionary– of chunks that are in the declarative memory). Thus, the following command instantiates dm as a (empty) declarative memory:

```
In [3]: dm = counting.DecMem()
```

Declarative memory standardly communicates with procedural knowledge via a retrieval buffer, which has to be instantiated, as well. To instantiate the buffer of dm:

```
In [4]: counting.dmBuffer("retrieval", dm)

Out[4]: set()
```

The method "dmBuffer" takes two obligatory arguments and one optional one. The obligatory arguments are "name" and "declarative_memory". "name" specifies the name that will be used in production rules to call retrieval. In this case, we call the retrieval "retrieval" but we could be more original, as long as we stay consistent within the model.

The other argument specifies to what declarative memory the buffer should be bound (i.e., from what memory it should retrieve and to what memory it has to clear). In our case, the choice is simple, since we only have one memory moduleat this point - dm.

The goal module is created by calling the method "goal". Again, "name" has to be specified under which the goal would be called in productions.

```
In [5]: g = counting.goal("g")
```

Finally, procedural knowledge consists of production rules, which are functions creating generators. We will discuss them shortly.

1.3 Chunks

Chunks are attribute-value matrices, building blocks of declarative knowledge. You can create chunks using chunkstring (shown here). The function chunkstring is really just a wrapper of the function makechunk. We will shortly mention it below.

We start by specifying a chunk type and all attributes it carries. The name of the chunk type (corresponding to the isa attribute in ACT-R) is written first, followed by an iterable of attributes (or the string of attributes, separated by commas). For example, to specify a chunk type "capital" which will have two attributes, "state" and "city" (it will store the knowledge of what city is the capital of what state), we write:

```
In [6]: actr.chunktype("capital", "state, city")
```

The chunk itself is written similarly as in Lisp ACT-R (as a attr value pair, wherein attr and value are separated by one or more spaces and pairs are separated by spaces from each other).

```
In [7]: usacapital = actr.chunkstring(string="isa capital state 'USA' city 'Washington'")
```

It is also possible to specify a chunk directly, skipping the chunk type definition. For example, if we want to have a chunk representing our knowledge of the current president of USA:

```
In [8]: usapresident = actr.chunkstring(string="isa president state 'USA' name 'Barack Obama'")
```

```
/home/jakub/Dropbox/Documents/moje/computations and corpora/python/pyactr/pyactr/chunks.py:89: UserWarni
  warnings.warn("Chunk type %s was not defined; added automatically" % typename)
```

Specifying chunk types is optional. However, it is recommended, as doing so might clarify what kind of attribute-value matrices you will need in your model. Notice also that if you don't specify the chunk type that your chunk uses, Python prints a warning message. This might help you debug your code (e.g., if you accidentaly named your chunk "ppresident", you would get a warning message that a new chunk type has been created - probably, not what you wanted). (If you don't want warning messages to be printed, you can suppress them by importing the package warnings and specifiying warnings.simplefilter("false"), or by passing ignore to -W when running Python. (See Python documentation for more on warnings.)

It is recommended that you only use attributes you defined first (or you used in the first chunk of a particular type). However, you can always add new attributes along the way (it is assumed that other chunks up to now had no value for those attributes in that case). For example, here is a chunk usapresident2, which is like usapresident but it adds the information about the years of presidency:

```
In [9]: usapresident2 = actr.chunkstring(string="isa president state 'USA' name 'Barack Obama' years '2
```

```
/home/jakub/Dropbox/Documents/moje/computations and corpora/python/pyactr/pyactr/chunks.py:84: UserWarni
  warnings.warn("Chunk type %s is extended with new attributes" % typename)
```

Notice that creating this chunk prints a warning that you extended the original chunk type president with new attributes (this might again help you debug your code).

We can see that the last two chunks could be matched, that is, usapresident2 has the same attribute-value pairs as usapresident, plus something extra. That is, the two chunks are not identical, but one is part of another. The comparison of chunks is a very common operation of ACT-R. In pyactr, you run it using the standard comparison operators

```
In [10]: usapresident == usapresident2
```

```
Out[10]: False
```

```
In [11]: usapresident < usapresident2
```

```
Out[11]: True
```

The name of chunktype corresponds to ISA-attribute in ACT-R. In LispACT-R, version 6, this attribute is a "syntactic sugar" and plays no role in determining how one chunk compares to another. This is true here, too. See:

```
In [12]: usainhabitant = actr.chunkstring(string="isa inhabitant state 'USA' name 'Barack Obama'")
         usainhabitant < usapresident2
```

```
/home/jakub/Dropbox/Documents/moje/computations and corpora/python/pyactr/pyactr/chunks.py:89: UserWarni
  warnings.warn("Chunk type %s was not defined; added automatically" % typename)
```

```
Out[12]: True
```

(Notice that the warning was printed because the chunktype inhabitant is new.) You can also leave out the chunk type completely (in that case, pyactr assumes a default chunk type undefined + some number, and it assumes that the chunk type is only defined by attributes that you mentioned:

```
In [13]: usainhabitant2 = actr.chunkstring(string="state 'USA' name 'John Doeville'")
         print(usainhabitant2)
```

```
undefined0(name=John Doeville, state=USA)
```

```
/home/jakub/Dropbox/Documents/moje/computations and corpora/python/pyactr/pyactr/chunks.py:89: UserWarni
  warnings.warn("Chunk type %s was not defined; added automatically" % typename)
```

```
In [14]: usainhabitant.typename
```

```
Out[14]: 'inhabitant'
```

```
In [15]: usainhabitant2.typename
```

```
Out[15]: 'undefined0'
```

Apart from chunkstring, you can also use makechunk, which takes three arguments: nameofchunk, typename and any number of slot-value pairs. One of our previous chunks would appear as follows in this function:

```
In [16]: usainhabitantnew = actr.makechunk(typename="inhabitant", state="USA", name="John Doeville")
         print(usainhabitantnew)
```

```
inhabitant(name=John Doeville, state=USA)
```

1.4 Adding chunks to modules

Chunks can be added to declarative memory using the method add. The following code adds our last chunk to dm and checks that it's there:

```
In [17]: dm.add(usapresident2)
         dm
```

```
Out[17]: {president(name=Barack Obama, state=USA, years=2009-2017): {0.0}}
```

Adding a chunk to the goal module is similar:

```
In [18]: g.add(actr.chunkstring(string="isa countFrom start 2 end 4"))
         g
```

```
Out[18]: {countFrom(count=None, end=4, start=2)}
```

1.5 Procedural knowledge

Procedural knowledge consists of production rules. These are written in a similar way as in Lisp ACT-R: left-hand side rules (tests) precede the double arrow ('==>'); right-hand side rules (actions) follow the arrow.

1.5.1 Buffer tests

When the content of a buffer is tested against some chunk, its name is prefixed with "=" and followed by '>'. This indicates that the buffer will be tested against the chunk that follows. For example:

```
In [19]: "=g> isa countFrom start 2"

Out[19]: '=g> isa countFrom start 2'
```

This tests whether the chunk in the goal buffer has value 2 in the attribute start. If this is so, production would proceed (to the next test or actions).

In tests, one can specify whether an attribute carries a particular value (like the value 2 above). Alternatively, it could be specified that an attribute carries no value. This is done using the keyword None. Finally, the value of an attribute could be assigned a variable, which is done by prefixing "=" to the name of the variable. The scope of the variable is the production rule - thus within one production rule, any variable will keep the same value. For example, the buffer test below requires that the goal buffer has a chunk in which start and end attributes carry the same value.

```
In [20]: "=g> isa countFrom start =x end =x"

Out[20]: '=g> isa countFrom start =x end =x'
```

Variables and values can be prefixed by "~". This is negation (corresponding to "-" in LispACT-R). For example, the following dictionary tests that the chunk in the goal buffer has a different value for start than for end.

```
In [21]: "=g> isa countFrom start =x end ~=x"

Out[21]: '=g> isa countFrom start =x end ~=x'
```

Finally, information can be combined. The example below would state that the goal chunk must have 2 as its value, which is assigned to x, and a value different from 4 which is assigned to y.

```
In [22]: "=g> isa countFrom start =x start 2 end =y end ~4"

Out[22]: '=g> isa countFrom start =x start 2 end =y end ~4'
```

A buffer does not need to be tested, it can be queried. This is done by prefixing the buffer with "?".

```
In [23]: "?g> buffer full"

Out[23]: '?g> buffer full'
```

This is true if the goal buffer has a chunk.

```
In [24]: "?g> buffer full"

Out[24]: '?g> buffer full'
```

This is true if the goal buffer is empty (has no chunk).

Other commands can test whether the buffer is busy etc. (for a more complete list, see later documents). Here are a few examples for retrieval.

```
In [25]: "?retrieval state free"

Out[25]: '?retrieval state free'
```

This is true if the retrieval buffer is not working on retrieving a chunk.

```
In [26]: ?retrieval state busy"
```

This is true if the retrieval buffer is working on retrieving a chunk.

```
In [27]: "?retrieval state error"

Out[27]: '?retrieval state error'
```

This is true if the last retrieval failed (the chunk was not found).

1.5.2 Buffer updates

Updates follow the arrow '==>'. In buffer updates, a buffer is always followed by ">". It is often prefixed with "=". This indicates that the buffer chunk will be (immediately) modified. For example, the following dictionary requires that the goal buffer is modified in such a way that the attribute count receives the value assigned to x (whatever that is in the current production rule).

```
In [28]: "=g> isa countFrom count =x"

Out[28]: '=g> isa countFrom count =x'
```

The "+" sign indicates a buffer request. Standardly, this results in the module replacing one chunk in the buffer by another one. In case of retrieval, the request results in search of the declarative memory that the buffer connects to, and putting the correct chunk in the buffer (if one is found).

```
In [29]: "+retrieval> isa countOrder first =x"

Out[29]: '+retrieval> isa countOrder first =x'
```

The dictionary above requires that the retrieval buffer should get a chunk from dm that has the value of x as the value of the attribute "first".

The third option is to prefix a buffer with "~". This requires that the buffer is cleared. For example, the following update would clear the retrieval buffer.

```
In [30]: "~retrieval>"

Out[30]: '~retrieval>'
```

As in LispACT-R, buffers are also cleared implicitly. If a retrieval requests a chunk, it is first cleared. Also, if a buffer is tested in buffer tests (first yield, prefixed with "=") but it does not appear in buffer updates, it is cleared (strict harvesting).

1.6 Running a model

Simulation of the model can be created when the model is ready. Simulation itself does not run anything, it only prepares discrete event simulation. This can then be run with the method "run". The method specifies how many seconds it should run (1s is the default value).

In [31]:
```
dm = counting.DecMem()
#this creates declarative memory

dm.add(actr.chunkstring(string="\
    isa countOrder\
    first 1\
    second 2"))
dm.add(actr.chunkstring(string="\
    isa countOrder\
    first 2\
    second 3"))
dm.add(actr.chunkstring(string="\
    isa countOrder\
    first 3\
    second 4"))
dm.add(actr.chunkstring(string="\
    isa countOrder\
    first 4\
    second 5"))

#creating buffer for dm
retrieval = counting.dmBuffer(name="retrieval", declarative_memory=dm)

#creating goal buffer
g = counting.goal(name="g")

g.add(actr.chunkstring(string="isa countFrom start 2 end 4"))
#adding stuff to goal buffer

sim = counting.simulation()
sim.run()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: start')
(0.05, 'PROCEDURAL', 'RULE FIRED: start')
(0.05, 'g', 'MODIFIED')
(0.05, 'retrieval', 'START RETRIEVAL')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'NO RULE FOUND')
(0.1, 'retrieval', 'CLEARED')
(0.1, 'retrieval', 'RETRIEVED: countOrder(first=2, second=3)')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: increment')
(0.15, 'PROCEDURAL', 'RULE FIRED: increment')
(0.15, 'g', 'MODIFIED')
(0.15, 'retrieval', 'START RETRIEVAL')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
(0.2, 'retrieval', 'CLEARED')
(0.2, 'retrieval', 'RETRIEVED: countOrder(first=3, second=4)')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: increment')
(0.25, 'PROCEDURAL', 'RULE FIRED: increment')
(0.25, 'g', 'MODIFIED')
```

```
(0.25, 'retrieval', 'START RETRIEVAL')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: stop')
(0.3, 'retrieval', 'CLEARED')
(0.3, 'PROCEDURAL', 'RULE FIRED: stop')
(0.3, 'retrieval', 'RETRIEVED: countOrder(first=4, second=5)')
(0.3, 'g', 'CLEARED')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'NO RULE FOUND')
```

The "run" method outputs the trace of the model. Each line represents an event. The first value is the time (in seconds) at which the event took place, the second value specifies what submodule is targeted in the event, the third value is the action that took place.

Notice that the model stopped at 0.3 seconds (and it did not run until reaching 1 second). This is because there were no events left to consider.

1.7 Stepping through a model

Alternatively, it is possible to proceed step by step through the model simulation, by using method step. (Notice that to run a new simulation, we don't need to completely specify the whole model again, we just specify what's needed to get started - retrieval and goal buffers in this case. Since we don't create a new declarative memory, the model will make use of the declarative memory as it is at this point, including the modifications made by the last simulation. But that does not matter in this case.)

```
In [32]: #creating buffer for dm
         retrieval = counting.dmBuffer(name="retrieval", declarative_memory=dm)

         #creating goal buffer
         g = counting.goal(name="g")

         g.add(actr.chunkstring(string="isa countFrom start 2 end 4"))
```

Using the following method allows one to check every step in the simulation.

```
In [33]: sim = counting.simulation()
         sim.step()
```

We can go on for a while.

```
In [34]: for _ in range(1, 8):
             sim.step()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: start')
```

Notice that it takes 8 steps to get to the second event (a rule is selected). This is because there are steps in the simulation that do not yield any output (mainly, setting up the model).

To see what the model is at this stage, we can inspect its trace (shown above), or we can do it directly by checking current event:

```
In [35]: counting.current_event
```

```
Out[35]: Event(time=0, proc='PROCEDURAL', action='RULE SELECTED: start')
```

We can also inspect what is happening to pieces of the model at any step. For this reason, we bound goal and retrieval buffers to corresponding variables. So we can check them now.

```
In [36]: g

Out[36]: {countFrom(count=None, end=4, start=2)}

In [37]: retrieval

Out[37]: set()
```

Nothing much interesting at this part, but we can proceed to some more interesting part (for example, the moment of the first retrieval).

```
In [38]: while True:
             sim.step()
             if counting.current_event.proc == 'retrieval':
                 break

(0.05, 'PROCEDURAL', 'RULE FIRED: start')
(0.05, 'g', 'MODIFIED')
(0.05, 'retrieval', 'START RETRIEVAL')
```

And we can investigate the current buffers.

```
In [39]: g

Out[39]: {countFrom(count=2, end=4, start=2)}
```

As you can see, the goal buffer is changed compared to the beginning, as it should be because it was now modified by firing the rule "start". The retrieval buffer is still empty (because the retrieval only started, nothing was retrieved yet). We can move to the point at which retrieval is done to see what was retrieved.

```
In [40]: while True:
             sim.step()
             if counting.current_event.action == "RETRIEVED: countOrder(first=2, second=3)":
                 break

(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'NO RULE FOUND')
(0.1, 'retrieval', 'CLEARED')
(0.1, 'retrieval', 'RETRIEVED: countOrder(first=2, second=3)')

In [41]: retrieval

Out[41]: {countOrder(first=2, second=3)}
```

Correctly, the retrieval buffer now carries the right chunk, given the conditions on retrieval specified in "start".

2 Further examples

The second document in this folder (02 – Environment, motor and vision module - the demo model) shows how an environment can be combined with an ACT-R model. The third document (03 - Intro to subsymbolic - the paired module) shows how the subsymbolic system is done in pyactr. Several more examples of models and environments are in the folder tutorials.