01 – Introduction to symbolic system – the count model

May 20, 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]: from pyactr.model import ACTRModel
```

```
In [2]: counting = ACTRModel()
        #initialize ACTRModel

        #Each chunk type should be defined first.
        counting.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:
        counting.chunktype("countOrder", "first, second")

        #create declarative memory
        dm = counting.DecMem()

        for i in range(1, 6):
            dm.add(counting.Chunk("countOrder", first=i, second=i+1))
            #adding chunks to declarative memory

        #create buffer for dm
        counting.dmBuffer("retrieval", dm)

        #create goal buffer
        g = counting.goal("g")


        counting.chunktype("countFrom", ("start", "end", "count"))
        #add chunk to goal buffer
        g.add(counting.Chunk("countFrom", start=2, end=4))

        #production rules:
        def start():
            yield {"=g":counting.Chunk("countFrom", start="=x", count=None)}
            yield {"=g":counting.Chunk("countFrom", count="=x"),
```

```
                    "+retrieval": counting.Chunk("countOrder", first="=x")}
    #e.g., this rule would look as follows in Lisp ACT-R:
    #(p
    #(p start
    #=goal>
    #  ISA          countFrom
    #  start        =x
    #  count        nil
    #==>
    #=goal>
    #  ISA          countFrom
    #  count        =x
    #+retrieval>
    #  ISA          countOrder
    #  first        =x
    #)

    def increment():
        yield {"=g":counting.Chunk("countFrom", count="=x", end="~=x"),
                    "=retrieval": counting.Chunk("countOrder", first="=x", second="=y")}
        yield {"=g":counting.Chunk("countFrom", count="=y"),
                    "+retrieval": counting.Chunk("countOrder", first="=y")}

    def stop():
        yield {"=g":counting.Chunk("countFrom", count="=x", end="=x")}
        yield {"!g": ("clear", (0, counting.DecMem()))}

    #add all rules into productions
    counting.productions(start, increment, stop)
```

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 [3]: 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', 'EXECUTED')
(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 [4]: 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 [5]: counting.dmBuffer("retrieval", dm)
```

```
Out[5]: 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 [6]: 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. There are two ways to create a chunk:

1. First, specify a chunk type and all attributes it carries. The name of the chunk type (corresponding to 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 [7]: counting.chunktype("capital", "state, city")
```

Then, write a chunk belonging to the chunk type. For example, if we want a chunk representing someone's knowledge of the capital of USA:

```
In [8]: usacapital = counting.Chunk("capital", state="USA", city="Washington")
```

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

```
In [9]: usapresident = counting.Chunk("president", state="USA", name="Barack Obama")
```

```
/home/jakub/Dropbox/Documents/moje/computations and corpora/python/pyactr/pyactr/chunks.py:96: UserWarni
  warnings.warn("Chunk type %s was not defined; added automatically" % typename)
```

Specifying a chunk type 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 [10]: usapresident2 = counting.Chunk("president", state="USA", name="Barack Obama", years="2009-2017
```

```
/home/jakub/Dropbox/Documents/moje/computations and corpora/python/pyactr/pyactr/chunks.py:91: 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:

```
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]: import warnings

         warnings.simplefilter("ignore")
         usainhabitant = counting.Chunk("inhabitant", state="USA", name="Barack Obama")
         usainhabitant < usapresident2
```

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

Still, you have to specify some kind of name for the chunk you want to create.

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 [13]: dm.add(usapresident2)
         dm

Out[13]: {president(years=2009-2017, state=USA, name=Barack Obama): {0.0}}
```

It might be tiresome (and error-prone) to add chunks one by one to a memory. In the count model, we used a loop for that:

```
In [14]: for i in range(1, 6):
             dm.add(counting.Chunk("countOrder", first=i, second=i+1))
```

This adds chunks "countOrder" with two attributes, first - having numbers 1 to 5, and second - having the number first + 1. All the chunks can be inspected in dm:

```
In [15]: dm

Out[15]: {president(years=2009-2017, state=USA, name=Barack Obama): {0.0}, countOrder(first=5, second=6)
```

Adding a chunk to the goal module is similar:

```
In [16]: g.add(counting.Chunk("countFrom", start=2, end=4))
         g

Out[16]: {countFrom(start=2, end=4, count=None)}
```

1.5 Procedural knowledge

Procedural knowledge consists of production rules. These are functions creating generators. Each function has two yields. The first yield specifies buffer tests. If the tests are passed then production receives the second yield, which specifies actions (buffer changes and requests).

Production rules have to be added to the model, by calling its method productions:

```
In [17]: counting.productions(start, increment, stop)
```

1.5.1 Buffer tests

Every yield has to yield a dictionary. The dictionary has, as its keys, buffers (in our case, either retrieval or g), prefixed by a symbol specifying what should be done with the buffer. In buffer tests, a buffer is often prefixed with "=". This indicates that the buffer will be tested against the chunk that follows. For example:

```
In [18]: {"=g": counting.Chunk("countFrom", start=2)}

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

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 second yield (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. (Notice that variables are written as strings. This is necessary - otherwise, we would end up with a syntactically illicit line from the perspective of Python.)

```
In [19]: {"=g": counting.Chunk("countFrom", start="=x", end="=x")}

Out[19]: {'=g': countFrom(start==x, end==x, count=None)}
```

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 [20]: {"=g": counting.Chunk("countFrom", start="=x", end="~=x")}

Out[20]: {'=g': countFrom(start==x, end=~=x, count=None)}
```

Finally, information can be combined. If this is so, values must be prefixed by "!". The example below states 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 [21]: {"=g": counting.Chunk("countFrom", start="=x!2", end="=y~!4")}

Out[21]: {'=g': countFrom(start==x!2, end==y~!4, count=None)}
```

A buffer does not need to be tested, it can be queried. This is done by prefixing the buffer with "?", and having a dictionary as its value.

```
In [22]: {"?g": {"buffer": "full"}}

Out[22]: {'?g': {'buffer': 'full'}}
```

This is true if the goal buffer has a chunk.

```
In [23]: {"?g": {"buffer": "empty"}}

Out[23]: {'?g': {'buffer': 'empty'}}
```

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 Lisp ACT-R and later documents). Here are a few examples for retrieval.

```
In [24]: {"?retrieval": {"state": "free"}}

Out[24]: {'?retrieval': {'state': 'free'}}
```

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

```
In [25]: {"?retrieval": {"state": "busy"}}

Out[25]: {'?retrieval': {'state': 'busy'}}
```

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

```
In [26]: {"?retrieval": {"state": "error"}}

Out[26]: {'?retrieval': {'state': 'error'}}
```

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

1.5.2 Buffer updates

Every update is a dictionary. The dictionary has, as its keys, buffers (in our case, either "retrieval" or "g"), prefixed by a symbol specifying what should be happening to the buffer. In buffer updates, a buffer 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 [27]: {"=g":counting.Chunk("countFrom", count="=x")}
```

```
Out[27]: {'=g': countFrom(start=None, end=None, 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 [28]: {"+retrieval": counting.Chunk("countOrder", first="=x")}
```

```
Out[28]: {'+retrieval': countOrder(first==x, second=None)}
```

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 [29]: {"~retrieval": None}
```

```
Out[29]: {'~retrieval': None}
```

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 not in buffer updates, it is cleared (strict harvesting).

Finally, a buffer can be executed. This is done by prefixing it with "!". Then, as the value, you have to have a sequence which has the name of the method as the first position, followed by a sequence of positional arguments (potentially empty).

```
In [30]: {"!g": ("clear", (0, counting.DecMem()))}
```

```
Out[30]: {'!g': ('clear', (0, {}))}
```

This, for example, will clear the goal buffer at time 0 into a declarative memory created just for this occasion. This can be used, for example, if you want to clear a buffer without having the result in any declarative memory you care about.

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]: counting = ACTRModel()

         counting.chunktype("countOrder", "first, second")

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

         for i in range(1, 6):
```

```
        dm.add(counting.Chunk("countOrder", first=i, second=i+1))
        #adding chunks to declarative memory

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

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

    counting.chunktype("countFrom", ("start", "end", "count"))
    g.add(counting.Chunk("countFrom", start=2, end=4))
    #adding stuff to goal buffer


    def start():
        yield {"=g":counting.Chunk("countFrom", start="=x", count=None)}
        yield {"=g":counting.Chunk("countFrom", count="=x"),
                    "+retrieval": counting.Chunk("countOrder", first="=x")}


    def increment():
        yield {"=g":counting.Chunk("countFrom", count="=x", end="~=x"),
                    "=retrieval": counting.Chunk("countOrder", first="=x", second="=y")}
        yield {"=g":counting.Chunk("countFrom", count="=y"),
                    "+retrieval": counting.Chunk("countOrder", first="=y")}

    def stop():
        yield {"=g":counting.Chunk("countFrom", count="=x", end="=x")}
        yield {"!g": ("clear", (0, counting.DecMem()))}

    counting.productions(start, increment, stop)

    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', 'EXECUTED')
(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.

In [32]:
```
counting = ACTRModel()

counting.chunktype("countOrder", "first, second")

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

for i in range(1, 6):
    dm.add(counting.Chunk("countOrder", first=i, second=i+1))
    #adding chunks to declarative memory

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

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

counting.chunktype("countFrom", ("start", "end", "count"))
g.add(counting.Chunk("countFrom", start=2, end=4))
#adding stuff to goal buffer


def start():
    yield {"=g":counting.Chunk("countFrom", start="=x", count=None)}
    yield {"=g":counting.Chunk("countFrom", count="=x"),
                "+retrieval": counting.Chunk("countOrder", first="=x")}


def increment():
    yield {"=g":counting.Chunk("countFrom", count="=x", end="~=x"),
                "=retrieval": counting.Chunk("countOrder", first="=x", second="=y")}
    yield {"=g":counting.Chunk("countFrom", count="=y"),
```

```
                    "+retrieval": counting.Chunk("countOrder", first="=y")}

        def stop():
            yield {"=g":counting.Chunk("countFrom", count="=x", end="=x")}
            yield {"!g": ("clear", (0, counting.DecMem()))}

        counting.productions(start, increment, stop)

        sim = counting.simulation()
```

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

```
In [33]: sim.step()
```

We can go on for a while.

```
In [34]: for _ in range(1, 10):
             sim.step()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: start')
(0.05, 'PROCEDURAL', 'RULE FIRED: start')
(0.05, 'g', 'MODIFIED')
```

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

```
In [35]: g
```

```
Out[35]: {countFrom(start=2, end=4, count=2)}
```

```
In [36]: retrieval
```

```
Out[36]: 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 [37]: while True:
             sim.step()
             if counting.current_event.proc == 'retrieval':
                 break
```

```
(0.05, 'retrieval', 'START RETRIEVAL')
```

And we can investigate the current buffers.

```
In [38]: g
```

```
Out[38]: {countFrom(start=2, end=4, count=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 [39]: 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 [40]: retrieval
```

```
Out[40]: {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

More examples of models are in the folder tutorials. These are the same models as in LispACT-R. (Their output closely matches that of LispACT-R, the main difference is that the trace of these models is less detailed than the trace of LispACT-R.)