7_counter_automata_in_ACTR

April 2, 2021

1 Counter automata in ACTR/`pyactr`

[1]:
```
# uncomment the line below to install pyactr
# !pip3 install pyactr

import pyactr as actr
```

The last example we discuss is the implementation of a counter automaton in ACT-R / `pyactr`. A counter automaton is a type of push-down automaton: it is a push-down automaton that allows only two symbols to appear on the stack. One well-known example of a language recognized by a counter automaton, but not generated by a regular grammar, is the language $\{a^n b^n : n \geq 1\} = \{ab, aabb, aaabbb, aaaabbbb, \dots\}$, for two arbitrary terminals a and b. One-counter automata recognize a subset of the context-free languages (see **Hopcroft, J.E., R. Motwani, and J.D. Ullman. 2001. Introduction to automata theory, languages, and computation. Addison-Wesley**, pp. 351 et seqq, for more details).

Let's implement a grammar corresponding to this automaton in ACT-R, and use it to generate (a finite subset of) this language. Our implementation builds on the counting model we discussed above since we need to count the number of a and b occurrences. Let's initialize the model and incorporate the counting model specification first. Everything below is the same as in the counting model with the exception of the fact that we add another slot `terminal` to our `countFrom` chunk type.

[2]:
```
counter = actr.ACTRModel()

actr.chunktype("countOrder", "first, second")
actr.chunktype("countFrom", ("start", "end", "count", "terminal"))

dm = counter.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
"""))
```

We will let the model start with the goal of generating two adjacent sequences of 3 elements each, the first sequence consisting only of the letters *a*.

[3]:
```
counter.goal.add(actr.chunkstring(string="""
    isa         countFrom
    start       1
    end         3
    terminal    a
"""))
```

We can now specify our production rules. The `"start"` rule below is the same as in the counting model. The other two rules – `"increment"` and `"restart counting"` below – are almost identical to the rules of the counting model except: (*i*) whenever we increment, we print the terminal (*a* or *b*); (*ii*) when we are done counting the first sequence of terminals (the sequence of *a*s), we do not stop but switch to counting and printing *b*s.

[4]:
```
counter.productionstring(name="start", string="""
    =g>
    isa         countFrom
    start       =x
    count       None
    ==>
    =g>
    isa         countFrom
    count       =x
    +retrieval>
    isa         countOrder
    first       =x
""")
```

[4]:
```
{'=g': countFrom(count= None, end= , start= =x, terminal= )}
==>
```

```
{'=g': countFrom(count= =x, end= , start= , terminal= ), '+retrieval':
countOrder(first= =x, second= )}
```

[5]:
```
counter.productionstring(name="increment", string="""
    =g>
    isa         countFrom
    count       =x
    end         ~=x
    =retrieval>
    isa         countOrder
    first       =x
    second      =y
    ==>
    !g>
    show        terminal
    =g>
    isa         countFrom
    count       =y
    +retrieval>
    isa         countOrder
    first       =y
""")
```

[5]:
```
{'=g': countFrom(count= =x, end= ~=x, start= , terminal= ), '=retrieval':
countOrder(first= =x, second= =y)}
==>
{'!g': ([(['show', 'terminal'], {})], {}), '=g': countFrom(count= =y, end= ,
start= , terminal= ), '+retrieval': countOrder(first= =y, second= )}
```

[6]:
```
counter.productionstring(name="restart counting", string="""
    =g>
    isa         countFrom
    count       =x
    end         =x
    terminal    a
    ==>
    +g>
    isa         countFrom
    start       1
    end         =x
    terminal    b
""")
```

[6]:
```
{'=g': countFrom(count= =x, end= =x, start= , terminal= a)}
==>
{'+g': countFrom(count= , end= =x, start= 1, terminal= b)}
```

We can now run the model. Notice that it prints 2 *a*s (since we start at 1 and count up to 3), followed by the same number of *b*s.

[7]:
```
counter_sim = counter.simulation(trace=False)
counter_sim.run()
```

```
terminal a
terminal a
terminal b
terminal b
```

The model can in principle generate indefinitely long expressions in the $a^n b^n$ language (if we add enough number knowledge to declarative memory), but in practice, it is limited by time and memory constraints. One might see this as a limitation of the implemented model, but this actually makes the model cognitively more realistic since humans are also limited by time and memory constraints.