6_regular_grammars_in_ACTR

April 2, 2021

Open in Colab

1 Regular grammars in ACTR/pyactr

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

import pyactr as actr
```

Regular grammars can be classified into right-regular and left-regular grammars. Right-regular grammars are grammars whose rules are of the following form:

- X $\rightarrow$ a Y (where a is a terminal and X, Y are non-terminals)
- X $\rightarrow$ a (where a is a terminal and X is a non-terminal)
- X $\rightarrow$ ϵ (where ϵ is the empty string and X is a non-terminal)

That is, the right-hand side of all production rules is constrained so that non-terminal symbols can only occur in the second position / on the right. Right-regular grammars are famously not expressive enough for natural languages (**Chomsky, Noam. 1956. Three models for the description of language. *IEEE Transactions on information theory* 2:113–124**), but they make for a good introductory example of modeling basic linguistic patterns in ACT-R.

Let us implement a right-regular grammar in ACT-R, which will generate NP (noun phrase) constituents consisting of indefinitely long strings of nouns. We will represent nouns with the terminal symbol 'N'. We effectively restrict ourselves to one rule. This rule is of the form NP $\rightarrow$ N NP. That is, every run of the model will generate an NP consisting of a potentially infinite number of Ns.

We need only one chunk type – `goal_chunk` on line 2 below – encoding the rule NP (`mother`) $\rightarrow$ N (`daughter1`) NP (`daughter2`). In addition to these three slots, this chunk type has a fourth slot `state`, which will enable us to toggle between printing the value of `daughter1` and applying the 'NP $\rightarrow$ N NP' rule recursively to the NP in the `daughter2` slot.

[2]:
```
regular_grammar = actr.ACTRModel()
actr.chunktype("goal_chunk", "mother daughter1 daughter2 state")
```

We initialize the goal buffer to an NP `mother` node. The value of `state` will be `rule`, which will simply signal that the rewrite rule should be triggered.

[3]:
```
regular_grammar.goal.add(actr.chunkstring(string="""
    isa         goal_chunk
    mother      NP
    state       rule
"""))
```

We need only three rules:

- one which implements our 'NP → N NP' rule: we rewrite the NP mother node as the daughters N and NP (in that order);
- another rule that prints the first daughter, i.e., the terminal node N;
- a final rule that sets the second daughter, which is the non-terminal NP, as the current node so that the rewrite rule can apply again;

The `"NP ==> N NP"` rule is triggered if our `goal_chunk` has NP as the mother node, no daughters, and is in a state expecting the rule to be applied. If these preconditions are satisfied, we generate the daughter nodes and we enter a `show` state in which the first daughter will be printed.

[4]:
```
regular_grammar.productionstring(name="NP ==> N NP", string="""
    =g>
    isa         goal_chunk
    mother      NP
    daughter1   None
    daughter2   None
    state       rule
    ==>
    =g>
    isa         goal_chunk
    daughter1   N
    daughter2   NP
    state       show
""")
```

[4]:
```
{'=g': goal_chunk(daughter1= None, daughter2= None, mother= NP, state= rule)}
==>
{'=g': goal_chunk(daughter1= N, daughter2= NP, mother= , state= show)}
```

The `"print N"` rule below is triggered only when the `goal_chunk` is in a `show` state. In that case, the value of the `daughter1` slot is printed and the `state` is switched back to a `rule` application state. Printing is done by specifying that a buffer should execute an action (that is what `!` encodes; see line 6, and then specifying the action. In this particular case, the command `show` on line 7 prints the value of the slot `daughter1`.

[5]:
```
regular_grammar.productionstring(name="print N", string="""
    =g>
    isa         goal_chunk
    state       show
    ==>
    !g>
```

```
    show        daughter1
    =g>
    isa         goal_chunk
    state       rule
""")
```

[5]:
```
{'=g': goal_chunk(daughter1= , daughter2= , mother= , state= show)}
==>
{'!g': ([(['show', 'daughter1'], {})], {}), '=g': goal_chunk(daughter1= ,
daughter2= , mother= , state= rule)}
```

The final rule "get new mother" sets the value of the daughter2 slot as the new mother node (assuming this value is not None), preparing the ground for a new application of the "NP ==> N NP" rule. It also erases the current values of the daughter1 and daughter2 slots, so that the "get new mother" rule cannot apply to its own output. This way, only the "NP ==> N NP" rule can be selected after the "get new mother" rule fires.

[6]:
```
regular_grammar.productionstring(name="get new mother", string="""
    =g>
    isa         goal_chunk
    daughter2   =x
    daughter2   ~None
    state       rule
    ==>
    =g>
    isa         goal_chunk
    mother      =x
    daughter1   None
    daughter2   None
""")
```

[6]:
```
{'=g': goal_chunk(daughter1= , daughter2= =x~None, mother= , state= rule)}
==>
{'=g': goal_chunk(daughter1= None, daughter2= None, mother= =x, state= )}
```

We can now run the simulation for different amounts of time and, depending on that, we will get NPs rewritten as N sequences of varying lengths. To see only the sequence of Ns, we suppress all other output by turning off the temporal trace for the simulation – see trace=False below.

[7]:
```
regular_grammar_sim = regular_grammar.simulation(trace=False)
regular_grammar_sim.run(0.5)
regular_grammar_sim = regular_grammar.simulation(trace=False)
regular_grammar_sim.run(1)
```

```
daughter1 N
daughter1 N
daughter1 N
daughter1 N
```

```
daughter1 N
daughter1 N
daughter1 N
daughter1 N
daughter1 N
```

If we want to examine the full trace of the model, we can run it with the `trace` turned on (which is the default setting, so we do not normally needed to explicitly specify it). We see that the model runs in repeated cycles: first, the `"NP ==> N NP"` rule fires, then the `"print N"` rule fires, then the `"get new mother"` rule fires, after which this three-rule cycle begins again.

[8]:
```
regular_grammar_sim = regular_grammar.simulation(trace=True)
regular_grammar_sim.run(0.5)
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: print N')
(0.05, 'PROCEDURAL', 'RULE FIRED: print N')
daughter1 N
(0.05, 'g', 'EXECUTED')
(0.05, 'g', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: get new mother')
(0.1, 'PROCEDURAL', 'RULE FIRED: get new mother')
(0.1, 'g', 'MODIFIED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: NP ==> N NP')
(0.15, 'PROCEDURAL', 'RULE FIRED: NP ==> N NP')
(0.15, 'g', 'MODIFIED')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'RULE SELECTED: print N')
(0.2, 'PROCEDURAL', 'RULE FIRED: print N')
daughter1 N
(0.2, 'g', 'EXECUTED')
(0.2, 'g', 'MODIFIED')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: get new mother')
(0.25, 'PROCEDURAL', 'RULE FIRED: get new mother')
(0.25, 'g', 'MODIFIED')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: NP ==> N NP')
(0.3, 'PROCEDURAL', 'RULE FIRED: NP ==> N NP')
(0.3, 'g', 'MODIFIED')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'RULE SELECTED: print N')
(0.35, 'PROCEDURAL', 'RULE FIRED: print N')
daughter1 N
(0.35, 'g', 'EXECUTED')
(0.35, 'g', 'MODIFIED')
```

```
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: get new mother')
(0.4, 'PROCEDURAL', 'RULE FIRED: get new mother')
(0.4, 'g', 'MODIFIED')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'RULE SELECTED: NP ==> N NP')
(0.45, 'PROCEDURAL', 'RULE FIRED: NP ==> N NP')
(0.45, 'g', 'MODIFIED')
(0.45, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.45, 'PROCEDURAL', 'RULE SELECTED: print N')
```