9_top_down_parsing_production_rules

April 8, 2021

Open in Colab

0.1 Building a top-down parser in `pyactr` [Part 2]

Model up to this point:

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

[2]:
```
actr.chunktype("parsing_goal", "stack_top stack_bottom parsed_word task")
actr.chunktype("sentence", "word1 word2 word3")
actr.chunktype("word", "form, cat")
```

[3]:
```
parser = actr.ACTRModel()
dm = parser.decmem
g = parser.goal
imaginal = parser.set_goal(name="imaginal", delay=0.2)
```

[4]:
```
g.add(actr.chunkstring(string="""
    isa parsing_goal
    task parsing
    stack_top S
"""))

imaginal.add(actr.chunkstring(string="""
    isa sentence
    word1 Mary
    word2 likes
    word3 Bill
"""))
```

[5]:
```
dm.add(actr.chunkstring(string="""
    isa word
    form Mary
    cat ProperN
"""))
dm.add(actr.chunkstring(string="""
    isa word
    form Bill
    cat ProperN
"""))
dm.add(actr.chunkstring(string="""
    isa word
    form likes
    cat V
"""))
```

0.1.1 Production rules

We now turn to the production rules that encode both our context-free grammar rules and the top-down parsing strategy represented in the `expand` and `scan` rules.

The first rule is an expanding rule, encoding the first phrase structure rule of our grammar: we expand S into NP and VP, in that order.

[6]:
```
parser.productionstring(name="expand: S ==> NP VP", string="""
    =g>
    isa parsing_goal
    task parsing
    stack_top S
    ==>
    =g>
    isa parsing_goal
    stack_top NP
    stack_bottom VP
""")
```

[6]:
```
{'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= S, task= parsing)}
==>
{'=g': parsing_goal(parsed_word= , stack_bottom= VP, stack_top= NP, task= )}
```

- the rule pops the `S` goal off the stack and replaces it with two subgoals `NP` and `VP`, in that order
- we do not modify the current `task`, which should remain in an active `parsing` state, so we omit it from the specification of the action:
  - the chunk in the consequent / right-hand side of the production rule only specifies the slots whose values should be updated, namely `stack_top` and `stack_bottom`

The second rule is once again an expanding rule: NP is expanded into ProperN.

[7]:
```
parser.productionstring(name="expand: NP ==> ProperN", string="""
    =g>
    isa parsing_goal
    task parsing
    stack_top NP
    ==>
    =g>
    isa parsing_goal
    stack_top ProperN
""")
```

[7]:
```
{'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= NP, task=
parsing)}
==>
{'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= ProperN, task= )}
```

- the rule only updates the top of the stack
- the bottom of the stack is left unmodified, so it is omitted throughout the rule

The third production rule expands VP into V and NP:

[8]:
```
parser.productionstring(name="expand: VP ==> V NP", string="""
    =g>
    isa parsing_goal
    task parsing
    stack_top VP
    ==>
    =g>
    isa parsing_goal
    stack_top V
    stack_bottom NP
""")
```

[8]:
```
{'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= VP, task=
parsing)}
==>
{'=g': parsing_goal(parsed_word= , stack_bottom= NP, stack_top= V, task= )}
```

- this rule is almost identical to the first rule, we only changed the syntactic category symbols
- the rule is triggered only when the 'parse a VP' goal is at the *top* of the stack
  - to trigger this third rule, something must happen after the successive application of the first and second rules `"expand: S ==> NP VP"` and `"expand: NP ==> ProperN"` that will promote the VP goal from the bottom of the stack to the top of the stack
- goals at the bottom of the stack can be promoted to the top when the top goal is popped off the stack and is not replaced by another goal
- this is what happens in a *scan* step
- in our case, a `scan` rule needs to pop the `ProperN` goal off the top of the stack
  - at the same time, the rule scans the first word `Mary` of our target sentence
- that is, once we have a terminal (e.g, ProperN or V) at the top of our stack, we have to check that the terminal matches the category of the word to be parsed
  - if it does, the word is parsed

We achieve this by means of three rules.

First, we place a retrieval request for a lexical item stored in declarative memory whose form is the current word to be parsed.

If a lexical item is successfully retrieved and the syntactic category of that lexical item is the same as the terminal at the top of our stack: - the current word is scanned, and - the top symbol on our stack is popped

The two retrieval rules for our two terminal symbols (ProperN, V) are provided below.

- in both cases, we place a retrieval request based on the form of the first word in the sentence to be parsed (`=w1`), and
- we change the state of the parsing goal to `retrieving` (rather than `parsing`)

[9]:
```
parser.productionstring(name="retrieve: ProperN", string="""
    =g>
    isa parsing_goal
    task parsing
    stack_top ProperN
    =imaginal>
    isa sentence
    word1 =w1
    ==>
    =g>
    isa parsing_goal
    task retrieving
    +retrieval>
    isa word
    form =w1
""")
```

[9]:
```
{'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= ProperN, task=
parsing), '=imaginal': sentence(word1= =w1, word2= , word3= )}
==>
{'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= , task=
retrieving), '+retrieval': word(cat= , form= =w1)}
```

[10]:
```
parser.productionstring(name="retrieve: V", string="""
    =g>
    isa parsing_goal
    task parsing
    stack_top V
    =imaginal>
    isa sentence
    word1 =w1
    ==>
    =g>
    isa parsing_goal
    task retrieving
    +retrieval>
    isa word
    form =w1
""")
```

```
[10]: {'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= V, task= parsing),
'=imaginal': sentence(word1= =w1, word2= , word3= )}
==>
{'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= , task=
retrieving), '+retrieval': word(cat= , form= =w1)}
```

The third rule is the actual scan rule: if the retrieved lexical item matches the top of our stack in syntactic category, then:

- we parse the word
- we pop the top symbol off the stack
- we move to the next word in our sentence
  - that is, we promote `word2` in our sentence to `word1`, and `word3` to `word2`

```
[11]: parser.productionstring(name="scan: word", string="""
    =g>
    isa parsing_goal
    task retrieving
    stack_top =y
    stack_bottom =x
    =retrieval>
    isa word
    form =w1
    cat =y
    =imaginal>
    isa sentence
    word1 =w1
    word2 =w2
    word3 =w3
    ==>
    =g>
    isa parsing_goal
    task printing
    stack_top =x
    stack_bottom None
    parsed_word =w1
    =imaginal>
    isa sentence
    word1 =w2
    word2 =w3
    word3 None
    ~retrieval>
""")
```

```
[11]: {'=g': parsing_goal(parsed_word= , stack_bottom= =x, stack_top= =y, task=
retrieving), '=retrieval': word(cat= =y, form= =w1), '=imaginal':
sentence(word1= =w1, word2= =w2, word3= =w3)}
==>
{'=g': parsing_goal(parsed_word= =w1, stack_bottom= None, stack_top= =x, task=
printing), '=imaginal': sentence(word1= =w2, word2= =w3, word3= None),
'~retrieval': None}
```

- on lines 20-21, the top of the stack is popped, so the symbol on the bottom of the stack is promoted to the top of the stack
- similarly, the imaginal buffer is updated on lines 23-27:
  - the word `=w1` that we just parsed is deleted from the sentence
  - the new sentence / word string that we still need to parse contains only words `=w2` and `=w3`, which are promoted to the `word1` and `word2` positions
- we also clear the retrieval buffer (`~retrieval>` on line 28)
- as a convenience:
  - the parsed word `=w1` is stored in the `parsed_word` slot of the parsing goal chunk (line 22)
  - we enter a new `printing` state (line 19) that will enable us to print a message reporting which word was just parsed

This printing action, performed by the rule below, is helpful to us as modelers, but it should not be part of our final processing model:

```
[12]: parser.productionstring(name="print parsed word", string="""
    =g>
    isa parsing_goal
    task printing
    =imaginal>
    isa sentence
    word1 ~None
    ==>
    !g>
    show parsed_word
    =g>
    isa parsing_goal
    task parsing
    parsed_word None
""")
```

```
[12]: {'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= , task= printing),
'=imaginal': sentence(word1= ~None, word2= , word3= )}
==>
{'!g': ([(['show', 'parsed_word'], {})], {}), '=g': parsing_goal(parsed_word=
None, stack_bottom= , stack_top= , task= parsing)}
```

The rule above says:

IF: - the current parsing goal is in a `printing` state (line 4) - the slot `word1` in the imaginal buffer is not empty (the squiggle ~ on line 7 is negation) - that is, we still have words to parse

THEN: - we should print the `parsed_word` in the `goal` buffer (lines 9-10) - line 9 `!g>` indicates that our python interpreter should execute an action that involves the goal buffer - the action is specified on line 10: call the method `show`, which will print the value of the `parsed_word` slot - when we're done printing, we delete the contents of the `parsed_word` slot and re-enter an active state of `parsing` (lines 11-14)

The last production we have to consider is the 'wrap-up' production we trigger at the end of the parsing process, provided below. - the parsing process ends when the `word1` slot in the imaginal buffer chunk has the value `None` (line 7) - the task is `printing` (line 4) - we therefore print the final word of the sentence which was just parsed (lines 9-10) - we declare the parsing process done by clearing the `imaginal` and `goal` buffers (lines 11-12)

```
[13]: parser.productionstring(name="done", string="""
          =g>
          isa parsing_goal
          task printing
          =imaginal>
          isa sentence
          word1 None
          ==>
          !g>
          show parsed_word
          ~imaginal>
          ~g>
      """)
```

```
[13]: {'=g': parsing_goal(parsed_word= , stack_bottom= , stack_top= , task= printing),
      '=imaginal': sentence(word1= None, word2= , word3= )}
      ==>
      {'!g': ([(['show', 'parsed_word'], {})], {}), '~imaginal': None, '~g': None}
```

```
[ ]:
```