18_left_corner_parsing_production_rules

April 12, 2021

Open in Colab

The left-corner parsing model so far:

[1]:
```
import pyactr as actr

environment = actr.Environment(focus_position=(320, 180))

actr.chunktype("parsing_goal", "task stack_top stack_bottom\
                                parsed_word right_frontier")
actr.chunktype("parse_state", "node_cat mother daughter1\
                               daughter2 lex_head")
actr.chunktype("word", "form cat")

parser = actr.ACTRModel(environment, motor_prepared=True)

dm = parser.decmem
g = parser.goal
imaginal = parser.set_goal(name="imaginal", delay=0)

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
"""))

g.add(actr.chunkstring(string="""
```

```
    isa             parsing_goal
    task            read_word
    stack_top       S
    right_frontier  S
"""))
```

0.1 The production rules for the left-corner parser

With the lexicon in place, we can start specifying the production rules.

Our first rule is the `"press spacebar"` rule below. This rule initializes the actions needed to read a word: - if: - the task is `read_word` (line 4) - the top of the stack is not empty (line 5), that is, we have some parsing goals left to accomplish - the motor module is `free` (available) - then: - we should press the space bar to display the next word

[2]:
```
parser.productionstring(name="press spacebar", string="""
    =g>
    isa             parsing_goal
    task            read_word
    stack_top       ~None
    ?manual>
    state           free
    ==>
    =g>
    isa             parsing_goal
    task            encode_word
    +manual>
    isa             _manual
    cmd             'press_key'
    key             'space'
""")
```

[2]:
```
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= , stack_top=
~None, task= read_word), '?manual': {'state': 'free'}}
==>
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= , stack_top=
, task= encode_word), '+manual': _manual(cmd= press_key, key= space)}
```

Assuming the next word has been displayed and the visual module has retrieved its form, we trigger the `"encode word"` rule below, which:

- gets the current value stored in the `visual` buffer, and
- initializes a new `get_word_cat` task

[3]:
```
parser.productionstring(name="encode word", string="""
    =g>
    isa             parsing_goal
    task            encode_word
```

```
    =visual>
    isa             _visual
    value           =val
    ==>
    =g>
    isa             parsing_goal
    task            get_word_cat
    parsed_word    =val
    ~visual>
""")
```

[3]:
```
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= , stack_top=
, task= encode_word), '=visual': _visual(cmd= , color= , screen_pos= , value=
=val)}
==>
{'=g': parsing_goal(parsed_word= =val, right_frontier= , stack_bottom= ,
stack_top= , task= get_word_cat), '~visual': None}
```

The `get_word_cat` task consists of placing a retrieval request for a lexical item stored in declarative memory.

As the rule `"retrieve category"` below shows, the retrieval cue consists of the form/value we got from the `visual` buffer. - while we wait for the result of this retrieval request, we enter a new `retrieving_word` task

[4]:
```
parser.productionstring(name="retrieve category", string="""
    =g>
    isa             parsing_goal
    task            get_word_cat
    parsed_word     =w
    ==>
    +retrieval>
    isa             word
    form            =w
    =g>
    isa             parsing_goal
    task            retrieving_word
""")
```

[4]:
```
{'=g': parsing_goal(parsed_word= =w, right_frontier= , stack_bottom= ,
stack_top= , task= get_word_cat)}
==>
{'+retrieval': word(cat= , form= =w), '=g': parsing_goal(parsed_word= ,
right_frontier= , stack_bottom= , stack_top= , task= retrieving_word)}
```

If we are in a `retrieving_word` task and the declarative memory retrieval was successfully completed, which we know because the retrieved word is in the `retrieval` buffer, we can start building some syntactic structure, i.e., we can *sensu stricto* parse.

The first parsing action is the "shift and project word" rule below.

- the syntactic category of the retrieved word is pushed onto the top of the stack (pushing whatever was previously on top to the bottom of the stack)
- we store a new parse_state in the imaginal buffer
- the parse state is a unary branching tree with the syntactic category of the retrieved word as the mother/root node and the phonological form of the word as the only daughter
- we also enter a new parsing task in which we see if we can trigger any other parsing, i.e., syntactic structure building, rules

[5]:
```
parser.productionstring(name="shift and project word", string="""
    =g>
    isa             parsing_goal
    task            retrieving_word
    stack_top       =t
    stack_bottom    None
    =retrieval>
    isa             word
    form            =w
    cat             =c
    ==>
    =g>
    isa             parsing_goal
    task            parsing
    stack_top       =c
    stack_bottom    =t
    +imaginal>
    isa             parse_state
    node_cat        =c
    daughter1       =w
    ~retrieval>
""")
```

[5]:
```
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= None,
stack_top= =t, task= retrieving_word), '=retrieval': word(cat= =c, form= =w)}
==>
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= =t,
stack_top= =c, task= parsing), '+imaginal': parse_state(daughter1= =w,
daughter2= , lex_head= , mother= , node_cat= =c), '~retrieval': None}
```

We now reached the point in our parser specification where we simply encode all the grammar rules into parsing rules.

The first two rules, listed below, project an NP node on top of a ProperN node. - NP projection comes in two flavors, depending on whether we are expecting an NP at the time we try to project one or not

If we do not expect an NP, we fire the "project: NP ==> ProperN" rule below: - this rule is triggered if: - the top of our stack is a ProperN, and - the bottom of our stack is not an NP, that is, we do not expect an NP at this time (`~NP` on line 5 below) - then: - we pop the ProperN category off the top of our stack - we replace it with an NP category, and - we add the newly built structure to the `imaginal` buffer - this newly built structure is a unary branching NP node with ProperN as its only daughter - in turn the NP node: - is attached to whatever the current right frontier `=rf` is - is indexed with the lexical head that projected the ProperN node in a previous parsing step

[6]:
```
parser.productionstring(name="project: NP ==> ProperN", string="""
    =g>
    isa             parsing_goal
    stack_top       ProperN
    stack_bottom    ~NP
    right_frontier  =rf
    parsed_word     =w
    ==>
    =g>
    isa             parsing_goal
    stack_top       NP
    +imaginal>
    isa             parse_state
    node_cat        NP
    daughter1       ProperN
    mother          =rf
    lex_head        =w
""")
```

[6]:
```
{'=g': parsing_goal(parsed_word= =w, right_frontier= =rf, stack_bottom= ~NP,
stack_top= ProperN, task= )}
==>
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= , stack_top=
NP, task= ), '+imaginal': parse_state(daughter1= ProperN, daughter2= , lex_head=
=w, mother= =rf, node_cat= NP)}
```

The second case we consider is an NP projection on top of a ProperN when an NP node is actually expected, as shown in rule `"project and complete: NP ==> ProperN"` below:

- if:
  - the current parsing goal has a ProperN at the top of the stack
  - and there is an NP right below it (at the bottom of the stack), that is, we are expecting an NP
- then:
  - we pop both the ProperN and the NP category off the stack (lines 14-15)
  - we add the relevant unary-branching NP structure to the `imaginal` buffer
  - we reenter a `read_word` task

[7]:
```
parser.productionstring(
    name="project and complete: NP ==> ProperN",
    string="""
        =g>
```

```
        isa             parsing_goal
        stack_top       ProperN
        stack_bottom    NP
        right_frontier  =rf
        parsed_word     =w
        ==>
        =g>
        isa             parsing_goal
        task            read_word
        stack_top       None
        stack_bottom    None
        +imaginal>
        isa             parse_state
        node_cat        NP
        daughter1       ProperN
        mother          =rf
        lex_head        =w
""")
```

[7]:
```
{'=g': parsing_goal(parsed_word= =w, right_frontier= =rf, stack_bottom= NP,
stack_top= ProperN, task= )}
==>
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= None,
stack_top= None, task= read_word), '+imaginal': parse_state(daughter1= ProperN,
daughter2= , lex_head= =w, mother= =rf, node_cat= NP)}
```

Now that we implemented the NP projection rules, we can turn to the S and VP grammar rules, implemented below.

- both of these rules are project-and-complete rules because in both cases we have an expectation for the mother node:
  - we expect an S because that is the default starting goal of all parsing-model runs, and
  - we expect a VP because the `"project and complete: S ==> NP VP"` rule always adds a VP expectation to the stack

The project-and-complete S rule is triggered after:

- we have already parsed the subject NP, which is sitting at the top of the stack (line 6),
- we have an S expectation right below the NP.

If that is the case, then: - we pop both categories off the stack and add an expectation for a VP at the top of the stack (lines 12-13) - we reenter the `read_word` task (line 11) - we introduce the expected VP node as the current right frontier that the object NP will attach to (line 14) - finally, we add the newly built syntactic structure to the `imaginal` buffer: - this is a binary-branching structure with S as the mother/root node and NP and VP as the daughters (in that order; lines 17-19)

[8]:
```
parser.productionstring(
    name="project and complete: S ==> NP VP",
    string="""
```

```
        =g>
        isa             parsing_goal
        stack_top       NP
        stack_bottom    S
        ==>
        =g>
        isa             parsing_goal
        task            read_word
        stack_top       VP
        stack_bottom    None
        right_frontier  VP
        +imaginal>
        isa             parse_state
        node_cat        S
        daughter1       NP
        daughter2       VP
""")
```

[8]:
```
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= S, stack_top=
NP, task= )}
==>
{'=g': parsing_goal(parsed_word= , right_frontier= VP, stack_bottom= None,
stack_top= VP, task= read_word), '+imaginal': parse_state(daughter1= NP,
daughter2= VP, lex_head= , mother= , node_cat= S)}
```

The `"project and complete: VP ==> V NP"` rule is very similar to the project-and-complete S rule.

This rule is triggered if: - we have just parsed a verb V, which is sitting at the top of the stack (line 7) - we have an expectation for a VP right below it (line 8)

If that is the case, then: - we pop both categories off the stack - we introduce a new expectation for the object NP at the top of the stack (lines 13-14) - we reenter the `read_word` task (line 12) - we store the newly built binary-branching VP structure in the `imaginal` buffer (lines 17-19)

[9]:
```
parser.productionstring(
    name="project and complete: VP ==> V NP",
    string="""
        =g>
        isa             parsing_goal
        task            parsing
        stack_top       V
        stack_bottom    VP
        ==>
        =g>
        isa             parsing_goal
        task            read_word
        stack_top       NP
```

```
        stack_bottom    None
        +imaginal>
        isa             parse_state
        node_cat        VP
        daughter1       V
        daughter2       NP
""")
```

[9]: {'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= VP, stack_top= V, task= parsing)}
==>
{'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= None, stack_top= NP, task= read_word), '+imaginal': parse_state(daughter1= V, daughter2= NP, lex_head= , mother= , node_cat= VP)}

We have now implemented all parsing rules corresponding to our grammar rules.

The final rule we need is a wrap-up rule that ends the parsing process.

- if:
  - our to-parse stack is empty, i.e., we have no categories to parse at the top of the stack (line 5 below)
- then:
  - we simply flush the `g` (goal) and `imaginal` buffers, which empties their contents into declarative memory

[10]:
```
parser.productionstring(name="finished", string="""
    =g>
    isa             parsing_goal
    task            read_word
    stack_top       None
    ==>
    ~g>
    ~imaginal>
""")
```

[10]: {'=g': parsing_goal(parsed_word= , right_frontier= , stack_bottom= , stack_top= None, task= read_word)}
==>
{'~g': None, '~imaginal': None}

[ ]: