Tutorial 2
Top-Down Parsing

1 Top-Level Architecture

1. In the research program manager, make a copy of your model before you incorporated word-length influence, and rename it Parser Architecture.

2. As was explained in the previous tutorial, it is a good idea to create logical modules in your COGENT model. In the context of these tutorials, we will use the architecture described below:

   a) Create two generic compound boxes to your model. Name them Computer and Participant.

   b) Use the context menu to demote List of Stimuli, Current Stimulus and Present Stimulus to the Computer module.

   c) Similarly, move Read and Press Key to the Participant module.

   The display remains at the top-level as an "interface" between the two modules, as does the output sink because this constitutes an acceptable simplification for our purposes.

2 The *Participant* Module

1. In the Participant module, create an additional process, Parse, in which to implement the parsing algorithm.

2. Create two buffers called Lexicon and Grammar, to which Parse needs read access only. Leave them empty for now.

3. In order to store temporary parsing results, you are going to need what is called a *"stack"*: a special type of list where elements can only be added to or taken off from the top.

   Unfortunately, COGENT's built-in stack buffers will not work for our purposes, because only the first element can be accessed. To implement the parsing algorithms covered in the lecture, we need an extended form of stacks in which more than one element at the top can be manipulated, and we are therefore going to model the stack in our parser using a Prolog list.

   a) Create a propositional buffer named Stack.

   b) Initialize it with the empty list.

4. Change the name of Press Key to Input/Output and remove Read.

5. Create two additional buffers Current Word and Previous Words. We will need these because when we implement backtracking, the parser will have to go back to re-parse previous words, but the word on the display does not change when the parser backtracks.

6. Current Word will contain the word on which the parser is currently focussing. Add a rule to Input/Output that "reads" the input from the display and adds it to Current Word. You can think of this rule as replacing lexical processing, which we will not model. Think about the conditions under which the rule should fire: What should the state of the system be like, especially the buffers Display and Current Word?

7. Previous Words will contain the words from the beginning of the sentence in reverse order.

   a) Initialize the buffer with the empty list.

   b) Then add the necessary conditions and actions to the rule you defined above to add the current word to the onset of the list.

      What happens here is basically the opposite of what we do in the Computer module when we take the list of words in a sentence apart: Here instead we add a word to the beginning of the list, but the notation is the same. Go back to the previous tutorial on list processing if you need to.

8. Also add a rule to Parse which deletes words from Current Word. This is a stub for the parsing process, on which we will expand in future versions of the model.

9. Remember that the same word can occur more than once in a sentence. What do you have to change to allow for this? Which rules does this affect?

10. Add the trigger system_quiescent to the "press key" rule to make sure the display is not cleared before all processing is done.

Importantly, note that we will not be keeping track of the parse tree the participant builds. Although we could, we are not really interested in saving it for our evaluation of the human plausibility of the different parsing algorithms, and it would require advanced list processing which we do not want to get into.

3 Debugging the Model Architecture

Now try to run the model. If your rules are correct, including refractedness, you will still notice a problem, namely that the same word is added over and over to Current Word and Previous Words. The reason for this is that the "read" rule (the second rule in Input/Output) fires each time the "parse" stub deletes a word from Current Word, so the system never reaches quiescence, and the display is never cleared.

In order to solve this problem, we need to make sure that once a word has been added by the "read" rule to Current Word and Previous Words, no more words are added until the space bar is pressed.

1. Add a buffer Number of Words to the module and initialize it with 0.

2. Then add the following to the rules in Input/Output:
   `N` is in Number of Words — to the "press key" rule
   `N1` is N+1 (from the arithmetic menu)
   …
   delete `N` from Number of Words
   add `N1` to Number of Words

   `N` is in Number of Words — to the "read" rule
   `L` is the length of PreviousWords
   `L` is not greater than `N`

4 Stimuli, Lexicon and Grammar

1. Our first parsers will be very basic, so they will only be able to deal with very simple sentences. Replace your stimuli with the two following sentences only: *"John loves Mary"* and *"John chases the cat"*.

2. In this version of the parser, we will use only binary grammar rules, i. e. rules which have exactly two symbols on the right-hand side (no unary and no ternary rules). They will be represented as Prolog terms of the form `rule(s, np, vp)` for S → NP VP. Enter this rule and the two following in the Grammar buffer:

   NP → Det N VP → TV NP

3. Then enter all the words in the lexicon along with their part-of-speech category using the following notation: `pos(john, np)`.

5 Implementing the Top-Down Algorithm

1. Make a copy of Parser Architecture and rename it Top-Down.

2. Initialize the stack as indicated in the lecture slides.

3. Delete the rule in Parse or comment it using Ignore in the context menu.

4. Formalize the parse rules in COGENT. First, "lookup":

   a) Preterminals are grammar symbols for which the lexicon contains at least one entry. Therefore, *"if the top of the stack is a pre-terminal P…"* can be translated in COGENT as the conjunction of:

      `[PreTerminal|_]` is in Stack
      `pos(_, PreTerminal)` is in Lexicon

   b) If we add *"get the next word W from the input"* and *"if P → W"*, this becomes:

      `Word` is in Current Word
      `pos(Word, PreTerminal)` is in Lexicon

c) *"Pop P from the stack"* is formalized as two actions, and requires us to give the remaining list of categories on the stack a name:

delete `[PreTerminal|Rest]` from Stack
add `Rest` to Stack

d) We must not forget to bind `Rest` in the conditions, so we modify them to:

`[PreTerminal|Rest]` is in Stack

e) In addition, `Word` must be deleted from Current Word.

f) You should also remember to make the rule unrefracted.

Note that failure is not added to the rule itself; it will occur automatically when the model finds that no more rules can be applied, at which point it will jump to the next trial since it cannot yet backtrack.

5. Now formalize the second parse rule, "rewrite", on your own, using
   `rule(Mother, LeftDaughter, RightDaughter)`
   to represent the grammar rule, where `Mother` is the non-terminal and the right-hand side of the rule is `LeftDaughter, RightDaughter` (hint: to add two elements to the beginning of a list, you can use `[a,b|rest]`).

6 Debugging the Model

1. Parse the first sentence, using only the necessary grammar rules and commenting other ones out for now. Make sure the basic functioning of your rules is correct.

2. Then reactivate all rules and try to parse the second sentence. Even if your model worked on the first sentence, you are likely to notice it has a problem with this one. Can you figure out what it is?

3. Draw the search tree for the sentence, using the complete grammar and lexicon. Make sure you do not miss any choice points. Now can you see the problem?

(Answer:) On trial 1, the stack contains `[np, vp]`. Therefore there should be two possibilities (your tree should have two branches): Either look up `john`, or rewrite the `np` as `det n`. But although the first option is the correct one, the parser applies the second one, and therefore fails. Why?

If you check your parse rules, you will notice that lookup would require `john` to be in Current Word, but it hasn't arrived there yet! The fix is thus: Make sure your parser does not start parsing before there is any input in Current Word, otherwise **choice points at which both parse rules apply will not be detected** (this can be enforced by adding an extra condition to the `rewrite` rule).

7 Serializing the Parser

Having made the change in the previous section, however, increases the likelihood that both parse rules are applied at the same time, since all choice points are now correctly detected.

Indeed, if you now try to parse the stimuli, you will notice that your model still does not behave as it should: Whenever there is more than one possibility (either because different parse rules apply, or because the same parse rule can be applied with different variable bindings, which includes different grammar rules), all rules fire in parallel and the result of each is added to the Stack, leaving you with multiple elements in that buffer.

In some cases, one of the alternatives on the Stack will eventually succeed, but not even that can be guaranteed, as you can see when parsing the second sentence: On cycle 5, `john` is looked up, which is correct, but the `np` is also rewritten to `det n`, so that both `[vp]` and `[det n vp]` are on the stack. `chases` is parsed correctly, removing the `[vp]` and adding `[np]` in its stead, but `[det n vp]` remains. Therefore on cycle 14, when `the` has reached Current Word, although the `np` is correctly rewritten, lookup also fires and "eats up" the determiner, which is therefore not available anymore when it is needed at the next step!

Although parallel parsing is not not wrong *per se* and this could be fixed in other ways, since we want to investigate a linking hypothesis involving memory load, what we actually need is a serial parser. Thus we are now going to serialize the model by adding a mechanism for the parser to notice whenever several rules apply, and a selection mechanism to select among the alternatives. In the next tutorial, we will then add backtracking in order to obtain a complete model.

7.1 Splitting the Parse Rules into *"Propose"* and *"Execute"*

First, we are going to split each rule into two parts: A part that proposes a parsing operation and a part that takes up on the proposed operations and executes them. If we then notice at the proposal stage that there are more than one possibility, we will still have possibility to intervene and select among the alternatives, before the stack is actually modified.

We are going to put the old rule's conditions in a first rule which will send a kind of "signal" to the model that the rule can be applied, and then have a second rule pick up on that signal and actually carry out the old rule's actions.

1. Make a copy of your model, and rename the first one Top-Down Parallel and the second Top-Down with Rule Selection.

2. Add a buffer Possible Operations to the participant module, in which alternatives (or rather, their "signals") will be stored.

3. Make two additional copies of each rule in Parse and comment out the first of the three so you can refer to it while making further changes.

4. Then modify the lookup rules as shown below

Rule 1 (unrefracted): *Propose looking up a word*
IF: Word is in Current Word
pos(Word, PreTerminal) is in Lexicon
[PreTerminal|_] is in Stack
THEN: add lookup(Word, PreTerminal) to Possible Operations

Rule 2 (unrefracted): *Execute a lookup operation*
IF: lookup(Word, PreTerminal) is in Possible Operations
[PreTerminal|Rest] is in Stack
THEN: delete [PreTerminal|Rest] from Stack
add Rest to Stack
delete Word from Current Word

In this example, the "signal" we are adding to Possible Operations is `lookup(Word, PreTerminal)`. However, there are often several variants you can use for the signal, as long as it is specific enough to be unique whenever several rules apply.

For example, in this case, we could also have chosen to use
`signal(lookup(Word, PreTerminal), stack(PreTerminal|Rest))`
thereby binding `Rest` and allowing us to remove
`[PreTerminal|Rest]` is in Stack
from the execution rule. What *is* important, of course, is that, whatever **changes** you make **to the signal in the propose rule**, you also make the **same changes to your execution rule** so it can "pick up" on the signal in Possible Operations.

5. Make sure you understand what happens in the new lookup rules, and then modify the rewrite rules accordingly.

7.2 The Selection Mechanism

The mechanism to select among alternatives actually consists of two parts: A Prolog predicate that counts the number of elements in Possible Operations, and a rule that selects one alternative among all the possibilities.

In addition to rules, COGENT processes can also have Condition Definitions. These allow you to do everything you can do in the conditions of a rule, but they have the advantage of helping to avoid repeating bits of code in several rules, thereby making your rules more readable and easier to change later on. We are going to use one of these condition definitions to build a list of the alternatives in Possible Operations and count how many there are.

1. Add a condition definition to Parse by clicking on f(X) :- g(X) in the components bar. Condition definitions use the same syntax as Prolog predicates: `f(X) :- g(X)` means that if the right hand side of the rule is satisfied, then the left-hand side will also be true. You can also think of the variables on the left-hand side as "returning" a value.

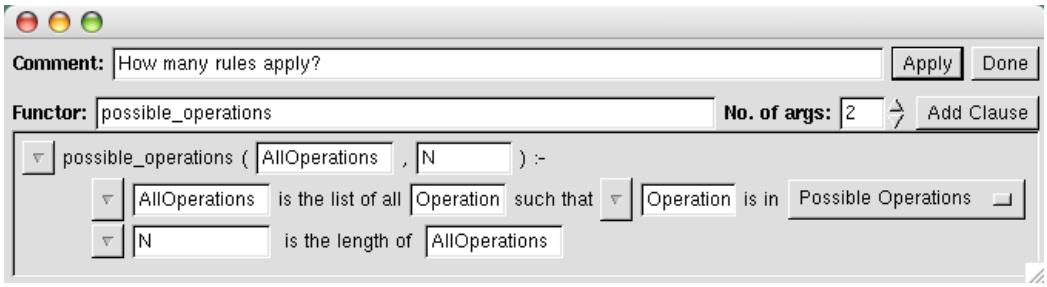
2. Now edit the condition as in the picture below:
   a) Give the predicate a name by giving the functor the value ***`possible_operations`***,
   b) Add a comment to remind you what it does,
   c) Specify that it has two arguments,

d) Add a single clause,

e) Give the arguments of the clause a name,

f) Add two subconditions to the clause by opening the drop-down menu on its left and selecting find all and length in the list processing menu,

g) And fill in their fields in as shown.



3. Now enter the rule below to select one of the alternatives in Possible Operations when there are too many.

   a) First, the condition you have just defined is called by selecting possible_operations/2 from the user defined menu. This "returns" a list of the alternatives and their number.

   b) Then we use arithmetic comparisons to check that the number of alternatives is greater than one.

   c) If this is the case, we delete all operations from Possible Operations,

   d) Bind the first element in the list of alternatives via list notation,

   e) And re-add it to Possible Operations.

Rule 5 (unrefracted; once): *Select an operation*
IF: possible_operations([First|_], N)
N is greater than 1
THEN: delete all _ from Possible Operations
add First to Possible Operations

Note that the rule is unrefracted and allowed to fire only once per cycle.

4. Finally, you also need to call possible_operations/2 in the conditions of both execution rules, in order to enforce that they should only fire when Possible Operations contains a single possibility: `possible_operations(_,1)`.

Note that since selection among the alternatives occurs at random, it is likely that, in some cases, the parser will not find the correct parse, even when it is licensed by the grammar. In the next tutorial, we will add a buffer to store discarded alternatives and we will add backtracking to the model, so that the (first grammatically) correct parse will akways be found.

7.3 Debugging

Now it's time for cleaning up… and debugging! You can finally delete the rules you previously commented out in Parse. Then you should run the model, see how it works and make the changes that are needed. For this, you will need to observe the contents of the Current Word, Stack and Possible Operations buffers, as well as the messages in Parse. In particular, you should pay close attention to the cycle number printed in front of the elements in Possible Operations.

In is important to keep the mechanism behind rule selection in mind. By introducing the buffer Possible Operations and splitting the parse rules into "propose" and "execute" parts, we have effectively introduced a selection cycle: First, all possibilities are determined and added to Possible Operations; then, if there are more than one, selection occurs; finally the chosen operation is executed. In order for your selection rules to work correctly, you need to make sure this cycle is respected (hint: consider checking the content of the various buffers by using "`not _ is in` <Buffer>" in the conditions of some of your rules).

Note that in general, it is not a good idea to use "`delete _`" and "`delete all _`" in the actions of your rules unless absolutely necessary. If you know a certain element needs to be deleted, then you should write that as "`delete Element`" instead of using the anonymous variable.

8 N-ary Grammar Rules

You are now going to make a minor change to your parse rules so they accept n-ary grammar rules, that is, rules which do not necessarily have exactly two daughters. First however,

- Change the second sentence in your stimuli to include an adjective, as in *"John chases the black cat"*,
- Replace all NPs in the lexicon with "PN" (proper noun), and add the above adjective.
- Load the grammar provided on the course webpage by right-clicking on the canvas, clicking on Import, selecting the file and choosing Replace.

Then examine the new grammar: As you might have expected, the daughters of a rule are now combined to form a single element by using a list. This will give us more flexibility when writing grammar rules, but it requires some syntactical changes to deal with it.

8.1 Changes to the Parse Rules

1. Luckily, only one of the two parse rules needs to be changed, although both its propose and execute parts are affected. Can you find out which?

2. Changing the proposal rule is relatively easy—all that needs to be changed is the type of the term used to represent the daughters in the rule so that it matches the one in the grammar:

a) Replace `LeftDaughter, RightDaughter` with a single variable called `Daughters`, representing the whole list of daughters, whatever their number is.

b) Importantly, note that no brackets are necessary around `Daughters`. Thus, you will have to be careful not to forget its type (consider adding a reminder in the rule's comment).

3. Modifying the "execute" rule is more difficult. Remember that concatenation using the | operator always occurs between the initial elements of the list and the sublist that constitutes its tail. Therefore, if we simply added the list of daughters to the front of the stack using "|", the list of daughters would be treated as an element, resulting in an embedded list, as in `[[d1,d2,d3]|Tail]`.

   Instead, we need to "flatten" the list to obtain `[d1,d2,d3|Tail]`. This can be achieved using append in COGENT's list processing menu. As an example, if `[d1,d2,d3]` were bound to `Daughters`, you would fill in the fields as

   `NewVariable` results from appending `Daughters` to `Tail`

   The result, `NewVariable`, would in this case be bound to `[d1,d2,d3|Tail]` and could then be used as usual in further conditions or in the actions of any COGENT rule. Make the corresponding change to the "execute" rule.

4. Finally, make sure you also adapt other conditions in the rule if necessary, among others the "signal" which is being picked up by the "execute" rule in Possible Operations.

9 Output from a Complete "Experiment"

1. Add a rule to Parse that sends a signal to the Participant module when the parser finds a parse for the sentence (e.g. send `parse_found` to Participant). The conditions of the rule will be the termination conditions for the top-down algorithm, and you will also have to make sure that after the rule fires, it cannot fire again on the next cycle (otherwise the system will never reach quiescence).

2. Then add a rule to Input/Output which will pick up on that signal and write `parse_found` to the output sink. This rule should be triggered by the signal you sent to the module, which you will have to enter by hand in the trigger field, but it will have no conditions at all.

3. Finally define scripts in the OOS window for subjects and experiments:

   a) The Subject (an older synonym for participant) script should initialise the subject, call the Trial script two times, and then end the subject.

   b) The Experiment script should call the Subject script and run at least 100 participants.

4. Gather the output in a file, and report how often your model found a parse for each of the sentences. Is this consistent across runs?