CFG Language

Context-free grammar rules are written in a well defined formal language, which I assume you know already. If you do not, check out https://matt.might.net/articles/grammars-bnf-ebnf/ and then come back to read this document.

In our case we need more than CFG, because it can only provide instructions to parse sentences to a tree. Whilst we also require a way to generate responses. Our implantation then uses CFG to parse sentences and what I informally call CFG-out to transform trees back into sentences.

Example:
Consider the following rule set:

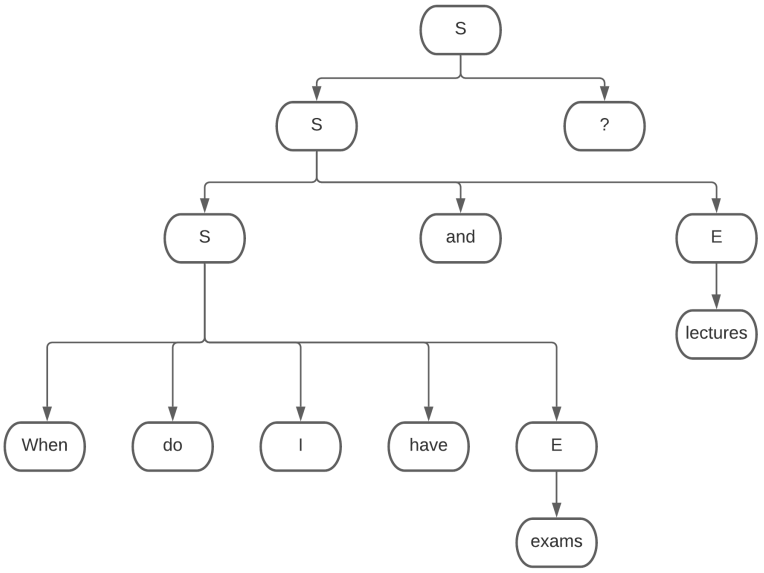
```
parse E as exams
parse E as lectures
parse S as When do I have E
parse S as S and E
parse S as S?

output E (lectures) as Monday
output E (exams) as Friday
output S (S?) as {S, 0}
output S (S and E) as {S, 0} and [E, 0] on {E, 0}
output S (When do I have E) as You have [E, 0] on {E, 0}
```

Firstly, this approach features two types of rules:

1) **Parse Rules:** These are identical to formal CFG production rules. In this example they are denoted by the keyword *parse* at the beginning.
2) **Output Rules:** These are different from formal CFG production rules. They do not apply a transformation on sequences of characters, but on trees.

How exactly do these output rules work?
Let's say we input something like "When do I have exams and lectures?". After applying the corresponding parse rules, we obtain the following tree:



After applying the output rules we obtain:

You have exams on Friday and lectures on Monday.

It may not be obvious, but the process that takes us from the initial input sentence to the output sentence is actually very simple. The following section explains how to interpret these output rules from the example and how they are executed by the interpreter.

Formal Syntax Definition:

Output *head* (*key*) **as** *transformation* instructs the interpreter to go to the parsed tree read the *current node* (which by default is the root node, until told otherwise) and test

1. If the *head* matches with the symbol of the node.
2. If the *key* matches with the children of the node.

If both conditions hold, then replace the node with *transformation*.

A *transformation* is a combination of any of three types of *modifiers*: constant, reader, mutator.

Take for example *You have [E, 0] on {E, 0}*. The words "*you*", "*have*", and "*on*" are all constant modifiers. However, "*[E, 0]*" is a reader. It retrieves the immediate content of *E* in the tree. Remember that *E* is a child of the *current node*. Lastly, "*{E, 0}*" is a mutator. It instructs the interpreter to apply the same rule set on *E* in the tree. So, this is a recursive call.

{ *S*, i **}** instructs the interpreter to apply the output rules set on the ith child node S of the *current node*.

[*S*, i **]** instructs the interpreter to read the name of the child of the ith S node of the *current node*.

And this is it. To better understand these rules, practice by going through the example by hand until you get the predicted output.