CKY

The main CKY program is located in `main.py`. It contains a small CLI for different tasks:

```
usage: main.py [-h] [--cp] [--ce] [--recognize] [--draw] [--bad-examples] [--grammar-data] [--text-data]

CYK parsing

optional arguments:
  -h, --help        show this help message and exit
  --cp              count using partials
  --ce              count using enumeration
  --recognize       only recognize
  --draw            draw trees with 2 to 5 parses
  --bad-examples    show ungrammatical examples
  --chart           silently create CKY chart
  --grammar-data
  --text-data
```

Recognizer

The result of the recognizer (`main.py --recognize`) is in `output/recognize.tsv` (0 for ungrammatical, for grammatical). Even though the ATIS sentences already contain some ungrammatical ones, here are few manual ungrammatical examples:

```
i ate unknown_word .    0
.       0
how many how many .     0
que ?   0
what .  0
can i have the having . 0
having done what .      0
```

To test whether the chart produces a sentences which covers the whole input (and therefore is grammatical), it we can check if `SIGMA` is located in the `chart[0][0]` cell.

Parser

The result of the parser (`main.py --draw`) are in `output/drawings.out`. Here Python yield/generator syntax came in incredibly useful, as it was possible to iteratively yield variants of subtrees. Here is a drawing example:

```
               SIGMA
    _____________|________
   |                    JKB
   |              _______|__________
   |           NP_NNS               |
   |       ______|________          |
VERB_VB AVP_RB        NOUN_NNS pt_char_per
   |      |              |          |
  show   the          flights       .
```

The number of parses for every sentence is in `output/count_enumerate.tsv`. It can be compared for sameness to the number of trees produced by `generate_gold_counts.py`, which has its output in `output/count_gold.tsv`.

Computing Trees Faster

The counts can be computed inductively: for single words, it is the number of rules which create the specific non-terminals. For an inner node with a specific tag it is the number of trees under the left child *times* the number of trees under the right child. The structure is very similar to building and yielding the trees, but instead of building a Leaf, the number of producing rules is returned, and instead of composing two Subtrees together, the counts are multiplied. This output is in `output/count_partial.tsv`

Performance

On a single run, I measured the following run times. Output was redirected to `/dev/null`.

Task	Arguments	Time	Without Chart
Load grammar		4.2s	-
CKY chart	`--chart`	82.5s	0s
Recognize	`--recognize`	85.1s	2.6s
Draw trees (2 to 5 parses)	`--draw`	87.7s	5.2s
Compute using enumeration	`--ce`	101.2s	18.7s
Compute using partials	`--cp`	84.4s	1.9s

Values in the table demonstrate that the alternative way of computing the number of parse trees based on multiplication is significantly slower. Furthermore, it shows that building the CKY chart takes the most time.

Grammar

I also implemented my own CFG -> ChNF 'converter'. Interestingly enough, the resulting grammar in ChNF has the same number of rules as the provided grammar (20326). I did not expect this to happen, yet it makes sense, that if all redundant rules are removed, the number of rules is the same. Question: How do we know a grammar in ChNF is minimal and there does not exist another grammar yielding the same language with less number of production rules?

The correctness of this conversion was tested on the number of parse trees (`output/count_partial_custom.tsv`), which should remain the same in comparison to the original grammar (`output/count_partial.tsv`). Obviously, the parse trees themselves are different (`output/drawing_custom.out`), because I chose a different renaming scheme for new non-terminals (uniquely identified by the current line and the decomposition step). These new tags were appended to the previous tags, so it is more apparent where the specific rule came from. This is at the cost of the grammar file being almost +50% larger. Here is an example of parse with the new grammar:

```
               SIGMA
    _____________|__________
   |                  1387_1_IMPR_VB
   |              __________|_____________
   |           NP_NNS                     |
   |       ______|__________              |
VERB_VB AVP_RB           NOUN_NNS    pt_char_per
   |      |                 |             |
  show   the             flights          .
```

Generating the grammar is quite fast (~2.3s) and does not affect the runtime of the main program. The generated grammar can be found in `grammars/atis-grammar-customcnf.cfg`. The program `convert_grammar.py` default to `grammars/atis-grammar-original.cfg`, which can be changed with `--grammar-data` argument and outputs the new grammar to stdout.

On a final note I would like to mention, that the grammar converter would not work in a real environment, because it does not check for the following issues: (1) top-level sentence token on the right side of a production rule, (2) terminals mixed with terminals in the production and (3) epsilon productions. None of these was however, present in `grammar/atis-grammar-original.cfg`, which made this task easier.