In [ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

CS187

Lab 3-1 – Context-free grammars introduced

In [ ]:
```
# Please do not change this cell because some hidden tests might depend on i
import os

# Only install packages if not in cs187-env
if 'cs187-env' not in os.environ.get('CONDA_PREFIX', ''):
    import subprocess
    import sys
    subprocess.run([sys.executable, '-m', 'pip', 'install', '-q', '-r', 'req
                    check=True)
```

Preparation – Loading packages

In [ ]:
```
import functools
import math
import nltk
```

Writing CFGs

Recall that a context free grammar (CFG) is a set of rules of the form $A \rightarrow \beta$, where A is a nonterminal symbol and β is a sequence of terminal and nonterminal symbols. The set of nonterminals is N and the set of terminals is Σ. One of the nonterminals is a special start symbol, conventionally denoted S.

A CFG generates a string by starting with the start symbol and repeatedly replacing a nonterminal symbol by the right-hand side of a rule whose left-hand side matches that nonterminal.

We will use the Natural Language Tool Kit (NLTK) to define, represent, and store context-free grammars and syntactic parse trees in data structures. The toolkit provides for constructing a grammar from a textual description, such as this example:

In [ ]:
```
simple_grammar1 = nltk.CFG.fromstring("""
    S -> NP VP
    NP -> 'dogs'
    NP -> 'cats'
```

```
    NP -> 'husky' 'dogs'
    VP -> V
    VP -> V NP
    V -> 'bark'
    V -> 'jump'
    V -> 'chase'
""")
```

Some things to notice about the NLTK encoding of grammars:

- Nonterminals are those symbols that appear on the left-hand side of a rule.
- Terminals are any other Python object, but typically (as here) a string. (Notice how to write multi-word expressions on the right-hand side: each word needs to be quoted separately, since the tokenizer breaks at whitespace.)
- By convention, the start symbol is the left-hand side of the first production of the grammar, and is typically denoted by the nonterminal `S`.

We can print the grammar to observe it:

In [ ]:
```
print(simple_grammar1)
```

Some sentences that are *generated* by this grammar include "dogs bark", "cats jump", "husky dogs chase cats". The last of these is generated as specified by the following derivation:

```
S => NP VP
  => husky dogs VP
  => husky dogs V NP
  => husky dogs chase NP
  => husky dogs chase cats
```

> This grammar also generates sentences that are ungrammatical in English, such as "dogs bark cats", as it makes no distinction between intransitive verbs (like "bark") and transitive verbs (like "chase"). For now, we'll ignore this issue.

The `nltk.CFG.fromstring` function also accepts grammars in a shorthand notation using the "or" operator `|` to combine multiple productions with the same left-hand side.

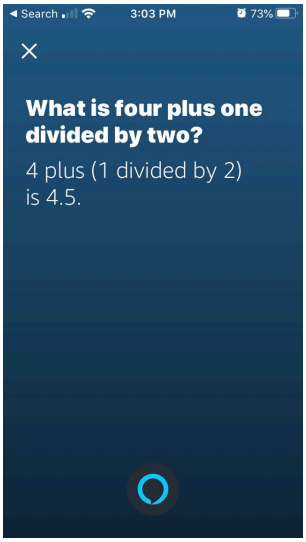
In [ ]:
```
simple_grammar2 = nltk.CFG.fromstring("""
    S -> NP VP
    NP -> 'dogs' | 'cats' | 'husky' 'dogs'
    VP -> V | V NP
    V -> 'bark' | 'jump' | 'chase'
""")
```

You can verify that the grammar is identical by printing it.

In [ ]:
```
print(simple_grammar2)
```

Textual arithmetic expressions

What is four plus one divided by two? As you can see in the screenshot at right, Alexa has a specific answer to this question.



In this lab, you will learn how to implement the first part of answering such "arithmetic in English" questions. In particular, you will write CFGs for a subset of the language of arithmetic expressions. You can assume that numbers are integers between zero and twenty and that the allowed operations are addition, subtraction, multiplication, and division. First, construct a CFG that generates the following expressions and similar ones.

1. twenty plus two
2. fifteen minus five
3. four divided by two plus one
4. two plus nine times five minus three
5. sixteen plus two minus ten times one

Your grammar can be written in the "semantic grammar" style, in which the nonterminals tend to correspond to semantic classes of phrases (like `NUM` (for numerals) or `OP` (for operators)), rather than syntactic classes (like `NP`, `VP`, etc.).

For now, don't worry about issues of ambiguity; your grammar may allow for multiple parses, for example for strings such as "four divided by two plus one".

In [ ]:
```
#TODO - construct a CFG for simple arithmetic expressions in English
arithmetic_grammar = ...
```

In [ ]:
```
grader.check("arithmetic_grammar")
```

Trees and grammars

Create a parse tree each for sentences 2 and 3 according to your grammar. Draw the parse trees on a piece of paper or in your favorite drawing application. You'll use these drawings in this notebook shortly.

Go ahead; we'll wait....

You're back. Good.

Drawing parse trees is a helpful visualization, but we need a machine-readable format for trees. One such format is a bracket notation. For example, a parse tree for "dogs bark" can be made as follows, using the `nltk.Tree.fromstring` function. (Notice that you don't need quotation marks for terminals here.)

In [ ]:
```
tree = nltk.Tree.fromstring("(S (NP dogs) (VP (V bark)))")
```

We can visualize the tree using the `pretty_print` method:

In [ ]:
```
tree.pretty_print()
```

You can get the rules that generated a tree using the `productions` method, which returns a list of productions according to a preorder tree traversal:

In [ ]:
```
tree.productions()
```

Convert the parse trees you drew for sentences 2 and 3 into NLTK format by converting them from a string using bracket notation, as done above for `tree`. Call them `tree2` and `tree3`.

In [ ]:
```
#TODO -- Construct trees for sentences 2 and 3 by
#        converting from the bracket notation.
# "fifteen minus five"
tree2 = ...
# "four divided by two plus one"
tree3 = ...
```

In [ ]:
```
grader.check("parse_trees")
```

It's useful to draw the trees to make it easier to visually inspect them.

In [ ]:
```
tree2.pretty_print()
tree3.pretty_print()
```

The following function validates that a tree only contains productions that are valid according to a given grammar.

In [ ]:
```
def validate(tree, grammar):
    return all([production in grammar.productions()
                for production in tree.productions()])
```

Using the `validate` function, we can validate that the trees your wrote are valid with respect to your grammar.

In [ ]:
```
print(validate(tree2, arithmetic_grammar))
print(validate(tree3, arithmetic_grammar))
```

Expanding the grammar

The arithmetic expressions we considered so far were rather limited. In practice, there are many ways to express such expressions. Expand your grammar to generate also the following expressions, which use other ways of indicating the arithmetic operations, in addition to the previous ones:

6. the sum of twenty and two
7. the difference between fifteen and five
8. the quotient of three and five
9. the sum of the product of four and two and one

In [ ]:
```
#TODO
expanded_arithmetic_grammar = ...
```

In [ ]:
```
grader.check("expanded_arithmetic_grammar")
```

Create parse trees for sentences 6 and 9 according to your grammar. Again, you might find it useful to first draw the trees by hand and then write them in bracket notation using the `nltk.Tree.fromstring` function.

In [ ]:
```
#TODO
# 6. "the sum of twenty and two"
tree6 = ...
# 9. "the sum of the product of four and two and one"
tree9 = ...
```

In [ ]:
```
grader.check("expanded_arithmetic_trees")
```

In [ ]:
```
tree6.pretty_print()
tree9.pretty_print()
```

Testing the grammar

Now that you have a CFG for arithmetic expressions, it is time to test its capabilities. Can your grammar generate the following new expressions? If not, edit the grammar to make sure it can handle such expressions.

10. three added to eight
11. the sum of two and nine times the difference between five and three
12. ten

In [ ]:
```
#TODO
expanded_arithmetic_grammar = ...
```

In [ ]:
```
grader.check("further_testing")
```

Preview to parsing

Verifying by hand that a sentence can be generated by a grammar is cumbersome and impractical. We would like an automatic procedure for doing that at scale, that is, a *parser*. The `nltk` system can construct a parser given a grammar. In later labs, you'll write your own parsers.

> Strictly speaking, an algoithm that determines *whether* a sentence can be generated by a grammar (that is, returns a simple `true` or `false`) is properly referred to as a *recognizer*. A *parser* goes further, in returning one or more parse trees for grammatical sentences.

We'll make use of NLTK's "bottom up chart parser" facility to build a parser for your arithmetic grammar.

In [ ]:
```
parser = nltk.parse.BottomUpChartParser(expanded_arithmetic_grammar)
```

Using this parser we can get parses for a given sentence.

In [ ]:
```
test_sentence = "the sum of two and nine times the difference between five a
for tree in parser.parse(test_sentence):
    tree.pretty_print()
```

You may have noticed that some of the arithmetic expressions were *ambiguous*; they had multiple distinct valid parses. In the next few labs, we will deal with the important matter of ambiguity.

Parser evaluation

To evaluate the quality of a syntactic parser, we need an evaluation metric to compare a predicted *hypothesis* parse with a gold *reference* parse. Common evaluation metrics include:

1. Exact match – 1 if the predicted and reference parses are identical; 0 otherwise
2. Precision (P) – the proportion of constituents in the hypothesis that are correct (that is, match the gold parse)

$$P = \frac{\cnt\text{correct constituents in a hypothesis parse}}{\cnt\text{constituents in a hypothesis parse}}$$

1. Recall (R) – the proportion of constituents in the gold parse that are predicted (that is, match the hypothesis parse)

$$R = \frac{\cnt\text{correct constituents in a hypothesis parse}}{\cnt\text{constituents in a reference parse}}$$

1. F_1 score – the harmonic mean of precision and recall

$$F_1 = \frac{2}{1/P + 1/R} = \frac{2PR}{P+R}$$

We consider a constituent in one tree to match a constituent in another if they ***share the same nonterminal*** and ***cover the same span of terminal symbols***. We don't count the terminals by themselves as constituents, so, for instance, the `ref_tree` we define below has five constituents, not eight.

Consider the following trees: a reference tree `ref_tree` (the "gold" or "ground truth") and a hypothesis tree `hyp_tree` (the "predicted" tree that we wish to measure).

In [ ]:
```
ref_tree = nltk.Tree.fromstring("(S (NP dogs) (VP (V chase) (NP cats)))")
ref_tree.pretty_print()
```

In [ ]:
```
hyp_tree = nltk.Tree.fromstring("(S (S  (NP dogs) ) (VP (V chase) (NP cats))
hyp_tree.pretty_print()
```

Calculate the precision, recall, and $F1$ of the hypothesis tree `hyp_tree` relative to the reference tree `ref_tree`.

In [ ]:
```
#TODO
precision = ...
recall = ...
f1 = ...
```

In [ ]:
```
grader.check("metrics")
```

The precision-recall tradeoff

Often there is a tradeoff between precision and recall. In the above example, the recall is good, but at the expense of precision.

There can also be trees with high precision and low recall.

Find the smallest tree (that is, with the fewest nodes) that has a precision of 1 with regards to the above `ref_tree`. The tree should, like the reference tree, have the nonterminal `S` at its root and cover the terminal string `dogs chase cats`. However, it ***need not otherwise be valid*** according to the grammar. What is its recall?

In [ ]:
```
#TODO -- Fill in the minimal precision-1 tree and its recall
minimal_high_precision_tree = ...
recall_of_minimal_high_precision_tree = ...
```

```
minimal_high_precision_tree.pretty_print()
```

In [ ]:
```
grader.check("tradeoff")
```

Normal forms

As you have seen above, there are many ways to write a grammar for a given language. It is often convenient, however, to work with a standard format. A *normal form* for context-free languages is a restricted production format for context-free grammars that still allows expressing arbitrary context-free languages.

As an example — which will come in handy later — we examine *Chomsky normal form*. A grammar in Chomsky Normal Form (CNF) has rules of only two kinds:

1. $A \rightarrow B\,C$ – a rule mapping a nonterminal symbol to exactly two nonterminal symbols
2. $A \rightarrow a$ – a rule mapping a nonterminal symbol to exactly one terminal symbol

> Actually, this version of CNF can't express languages that contain the empty string. To allow expression of any context-free language, we can allow a third type of rule $S \rightarrow \epsilon$, where S is the start symbol of the grammar and ϵ represents the empty string. But for our purposes, we'll stick to the simpler version here.

A CFG parse tree can be transformed to one generable by a CNF grammar in a variety of ways, typically by introducing special new nonterminals. Here, we use `nltk` to perform the transformation. The result is a binary tree. The binary branching property will turn out to be useful when we turn to implementing parsers.

In [ ]:
```
tree = nltk.Tree.fromstring("(S (NP dogs) (CONJ and) (NP cats) )")
tree.pretty_print()
tree.chomsky_normal_form()
tree.pretty_print()
```

Some parsing algorithms require the grammar to be in CNF. Manually convert the arithmetic grammar you wrote in the first part of this lab (`arithmetic_grammar`) to CNF. You may need to introduce "dummy" nonterminals to allow that. Your CNF grammar should recognize exactly the same strings as the original CFG, though the parse trees will be different.

In [ ]:
```
#TODO - convert the arithmetic grammar you wrote in the *first part* of
#       this lab (arithmetic_grammar) to CNF.
cnf_arithmetic_grammar = nltk.CFG.fromstring(
    ...
)
```

In [ ]:
```
grader.check("cnf_conversion")
```

The NLTK grammar method `is_chomsky_normal_form` allows us to verify that the grammar is indeed in CNF:

In [ ]:
```
cnf_arithmetic_grammar.is_chomsky_normal_form()
```

Lab debrief

Question: We're interested in any thoughts you have about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there general additions or changes you think would make the lab better?

Type your answer here, replacing this text.

Question: What *specific single* change to the lab would have made your learning more efficient? This might be an addition of a concept that was not explained, or an example that would clarify a concept, or a problem that would have captured a concept in a better way, or anything else you can think of that would have made this a better lab.

Type your answer here, replacing this text.

End of lab 3-1 {-}

To double-check your work, the cell below will rerun all of the autograder tests.

In [ ]:
```
grader.check_all()
```