# Language and Computation

LING 227 01 / 627 01 / PSYC 327 01
**Assignment #4**
**Due:** April 10, 2014

The solutions to the problem set must be handed in on paper at the beginning of the class. It must be typed (printed), no handwritten solution will be accepted. If you do not know how to draw trees, graphs, etc. electronically, then draw them by hand on a sheet of paper, scan that paper, and include the image into your file.

Additionally, you will also send your Python code in an email by 4 pm on the day of the deadline to `tamas.biro@yale.edu`. The file name must contain both your first name and last name.

Each problem set is worth 10 points in total.

## Problem 1: Parsing mathematics (10 points)

The "language" of mathematics is not a natural language. It has been consciously created by humans, and it is not a means to communicating a broad range of messages in spontaneous speech or other contexts. Therefore, it has a structure that is simple enough for you to practice important aspects of natural language processing, even if the problem does not seem to be language-related.

Our goal is to judge if statements such as the following are *well-formed*:

$$\begin{aligned} 3 + 45 &= 6 * 8 \\ 40 + (10 - 2) &= 3 + 3 * 8 \\ (4 * 10 + 8 &> * 50 - (1 + 1 + 1 \end{aligned}$$

Judging their **syntactic** well-formedness is the first step toward actually deciding whether they are true or false. But we will not go so far, into **semantics**, right now. Here is the *context free grammar* that serves as our starting point:

$\Sigma = \{$`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, =, >, <, (, )`$\}$
$N = \{S, E, N, D\}$

We will not make use of exponentiation and more advanced operations. We also ignore $\leq$ and $\geq$, as well as negative numbers and the negative of some expression. $S$ is the designated *start symbol*, and here are the *production rules*:

$S \rightarrow E$ `=` $E$ $\quad S \rightarrow E$ `>` $E$ $\quad S \rightarrow E$ `<` $E$
$E \rightarrow E$ `+` $E$ $\quad E \rightarrow E$ `-` $E$ $\quad E \rightarrow E$ `*` $E$ $\quad E \rightarrow E$ `/` $E$
$E \rightarrow$ `(` $E$ `)` $\quad E \rightarrow N \quad N \rightarrow D \quad N \rightarrow N\ D \quad D \rightarrow$ `0|1|2|3|4|5|6|7|8|9`

## Part 1.1: Understanding this grammar (2 point)

Question 1: Describe in words the formal language generated by this grammar.
Question 2: Which class in the Chomsky hierarchy does this language (over $\Sigma^*$) belong to? Which production rule can be "blamed" for it not belonging to a narrower class? Can you formally prove that this language does not belong to a narrower class?

## Part 1.2: Ambiguity of our grammar (1 point)

This grammar is highly ambiguous: a well-formed string can be assigned multiple parses. Is this a problem?

Technically speaking, it may be a problem, but it really depends on what we are using our grammar for. In linguistics, ambiguities are crucial if they influence *semantics*, that is, the meaning of the sentence. Remember sentences such as *I shot an elephant in my pajamas*: two different meanings are associated with two different parse trees (depending on the place of the preposition phrase *in my pajamas*, see Fig. 13.5 of Jurafsky and Martin).

In our "language of mathematics", the 'meaning' of an expression would be its value (a number or a truth value). Each syntactic rewrite rule comes with some semantics: for instance, whenever the $E \rightarrow E$ + $E$ rule is used by syntax, semantics should know that the 'meaning' (mathematical value) of the higher node should be made equal to the sum of the meanings (values) of the left and right daughters.

Question: Provide two strings that are well-formed in this grammar, and which are both ambiguous. In the first case, ambiguity will not have a semantic consequence for the entire string, while it will in the second case. Show why each expression is ambiguous, and why there is or there is no consequence to this ambiguity.

A suggestion: you can visualize the semantics (the 'meaning', the value) of each phrase as an index to the node on the syntactic tree. Hereby one can easily keep track of *compositional semantics*: how the meaning of a phrase derives from the meanings of its components.

## Part 1.3: Improving our grammar (2 points)

Mathematical notations are not ambiguous to this extent. So we have to conclude that the grammar, as presented above, may be **weakly equivalent**, but not **strongly equivalent** to the "language of mathematics".
Question 1: What does *weak* and *strong* equivalence means in this case?

Moreover, beside ambiguity, our grammar has further problems.
Question 2: Can you identify at least one such problem?
Question 3: Refine the grammar so that it will not allow semantic ambiguity.

## Part 1.4: Chomsky Normal Form (1 point)

Turn your refined grammar into Chomsky Normal Form. Explain why and how you have done it.

## Part 1.5: Implementing the CKY Algorithm (3 points)

Implement the (non-probabilistic) **CKY Algorithm for recognition** in plain Python (i.e., without using the NLTK package), for this refined grammar. Your program will read a string from the standard input, and return either `accept` or `reject`. The grammar itself should already be hardcoded into your program. Suppose that the input string does not contain white spaces.

Practicalities: Here is an excerpt of how I have encoded my grammar, which you are welcome (but do not have) to follow:

```
rules_bin=[("S" , "E", "RE"),  # stands for the rule S -> E RE
           ("RE", "R", "E"),   # etc.
           ...,
           ("N" , "N", "N")
           ]

rules_fin=[("R", "="),         # stands for the rule R -> '='
           ("R", ">"),         # etc.
           ...,
           ("E", "9")
          ]
```

## Part 1.6: Testing your implementation with NTLK (1 point)

Needless to say, NLTK also has several parsing tools. Read Chapter 8 of the NLTK book (`http://www.nltk.org/book/ch08.html`) and the extra material (`http://www.nltk.org/book/ch08-extras.html`).

Do you want to check whether you have made any error when you transformed your grammar to a CNF and then implemented the CKY algorithm? Make use of the `parser.nbest_parse()` method to test your own implementation. Whenever your code returns `reject`, this method should return an empty list `[]`.

**You will submit** (both on paper and in an email) the code that includes both your implementation and the comparison of its output to the NLTK output.

Practicalities: Observe that the `nltk.parse_cfg()` method requires single quotes around terminals. The `parser.nbest_parse()` method accepts a string as its argument, but you can also use type conversion (`list(string)`) to convert you string input into a list `sent`, as done in the NLTK book example.