Remarks on Assignment 4

Language and Computation, spring 2014

Tamás Biró

1 Understand the distinction between the type of a grammar and the type of a language

1.1 On grammar types and language types

The former refers to the form of the rules in the grammar, whereas the later to the grammars that can possibly generate this language.

A **grammar** (N, Σ, S, P) belongs to class type-i, if its production rules in P are conform to the rule skeleton(s) of that class. A **language** $L \subseteq \Sigma^*$ belongs to class type-i, if it can be generated by a grammar of the corresponding class. However, a language can often be generated by several grammars.

First, observe that if a production rule p matches the rule skeleton of type-i, then it will also match the rule skeletons of type-j for all $j < i$.[1] Therefore, if a grammar belongs to type-i, then it will also belong to type-j for $j < i$.[2] As it is very easy to give examples of grammars that belong to type-j, but not to type-i for $j < i$, we conclude that the class of type-$(i+1)$ grammars is a proper subset of the class of type-i grammars.

It also follows that the class of type-$(i + 1)$ languages is a subset of the class of type-i languages. Refer to the lecture (03/25, slide 8) for examples demonstrating that the type-$(i + 1)$ language class is a *proper* subset of the type-i language class.

Focusing on the current assignment, observe that a grammar with (non-regular) context-free rules may still generate a regular language. For instance, the context free grammar with rules $S \rightarrow S\ S$ and $S \rightarrow a$ will generate the language `a+`.[3] However, the same language can also be generated by the weakly equivalent regular grammar $S \rightarrow a\ S$ and $S \rightarrow a$. In sum, the regular language `a+` can be generated by regular grammars (hence, it is a regular language), but also by "inherently" context-free grammars that are not regular.

[1] With the exception of the use of ϵ in rules. Refer to Fig. 16.2 (section 16.1) of the textbook.

[2] All regular grammars are context-free grammars, and all context-free grammars are context-sensitive grammars. In less precise speech, however, "grammar G is context-free" might also mean "context-free, but not regular". Likewise for "is context-sensitive" denoting "context-sensitive, but not context-free"; and likewise for formal languages.

[3] Here is a less self-evident example: (1) $S \rightarrow X\ A\ y\ B\ X$, (2) $A \rightarrow A\ A$, (3) $B \rightarrow B\ B$, (4) $A \rightarrow a$, (5) $B \rightarrow b$, (6) $X \rightarrow x$, which will generate the regular language `xa+yb+x`.

1.2 The language of mathematics

The question in the assignment pertained to the class of the language generated, and not to the class of the grammar. True, the grammar rules on p. 1 of the assignment did not match the "rule skeleton" of a regular grammar, but the rule skeleton of a context-free grammar. Therefore, most of the rules are to be "blamed" for this **grammar** to be context-free, and not regular.

And yet, if the grammar did not have parentheses, that is, if we removed the rule $E \rightarrow (E)$, then the language generated would be a regular language. Here is a regular expression describing the language without parentheses (while I leave it to you to formulate a corresponding regular grammar, as well as a corresponding finite-state automaton[4]):

```
/[0-9]+ ([+|-|*|/] [0-9]+)* [=|<|>] [0-9]+ ([+|-|*|/] [0-9]+)*/
```

The language generated by the grammar proposed for math is not regular since the rule $E \rightarrow (E)$ requires as many opening as closing parentheses, and, importantly, this number can be unlimited.[5] This is the only rule to be "blamed" for the **language generated** not being regular. One can draw parallels between using parentheses here and recursive *center embedding* in syntax (JM 16.2.2).

1.3 A formal proof

Informally speaking, we need an unlimited register to check the parentheses. Imagine the following procedure: setting the value of the register initially to zero, we read the string left to right, increase its value by 1 for each opening parenthesis, and decrease its value by 1 for each closing parenthesis. The string is well-formed for the parentheses only if the value of this register is never negative, and is exactly 0 after having read the left-hand side of the string, as well as after having read the whole string. Such an unlimited register is beyond finite-state technology, and is a typical example of the *stack* that characterizes the pushdown automata (PDA), the automata theoretic construction corresponding to context-free grammars.

However this is not yet a formal proof, which can only be provided by the *Pumping Lemma*. The Pumping Lemma has several formulations, but below I will refer to the one provided by Jurafsky and Martin, section 16.2.1:

Pumping Lemma: Let L be an infinite regular language. Then, there are strings x, y and z such that $y \neq \epsilon$, and $xy^n z \in L$ for all $n \geq 0$.

First observe that the strings x, y and z are provided by the Pumping Lemma and the language L, and it is not up to you to decide what "to pump".[6] Second, x and z might very well be the empty string. Third, although the truth value

[4] Hint: use separate non-terminals / states for the left-hand side and the right-hand side.

[5] With a limited number of parentheses, technically, the language would be regular.

[6] If you could choose what to pump, for instance, the relation symbol, you would be able to produce invalid strings, even in the case of the regular language without the parentheses.

of a mathematical formula might change after "pumping", this fact does not influence the syntactic validity of the string (see below).

Here is the sketch of a proof by contradiction that closely follows the example appearing in your textbook (section 16.2.1, as well as exercise 16.3):

Let us assume the opposite of what we aim at demonstrating: that the language L generated by our grammar is regular. Moreover, consider the following language: $L' = \{(^k 1)^l = (^m 2)^n | k, l, m, n > 0\}$. Language L' is regular, as it is described by the following regular expression: `/(+ 1 )+ = (+ 2 )+/`. Assuming that L is also regular, the intersection of L and L' must be another regular language, by the closure properties of the regular languages. Their intersection is the following set: $L'' = \{(^k 1)^k = (^m 2)^m | k, m > 0\}$: strings that follow the regular expression pattern of L', but also have the correct number of parentheses (as required by the single rule involving parentheses in L).

Since L'' is an infinite language, we can apply the Pumping Lemma: There are strings x, y and z such that y is not the empty string and $xy^n z$ belongs to L'' for all $n \geq 0$. Consequently (take n=1), there is a string in L'' that can be decomposed into these x, y and z. What should this string be, and what should its non-empty substring y be? Observe that each element of L'' contains exactly one `=` character, one `1` character and one `2` character; if y contained any of these characters, then y could not be pumped. It follows that either y only contains opening parentheses, or it only contains closing parentheses. Containing both (without containing other characters) is not an option, because opening and closing parentheses are not adjacent in L''. But then, by pumping y, we change the number of opening parentheses (if y contains them), or the number of closing parentheses (if y only contains them), and so we produce strings that do not satisfy the symmetry of opening and closing parentheses that characterizes the strings in L''. Hence, the pumping lemma does not hold, which means L'' cannot be regular, whence L cannot be regular, either. *Q.e.d.* □

A final remark: Some proved that L has a non-regular subset. From this fact it does not follow, however, that L is non-regular. For instance, $\{a^n | n \text{ is prime}\}$ is a highly non-regular subset of the regular language $\{a^n | n \geq 1\}$.

2 Understand the difference between "well-formed syntactically" and "true semantically"

The sentence "*Connecticut is the largest state in the US*" is a syntactically well-formed sentence, and must be accepted by a syntactic parser, independently of whether it is true or false. In fact, a semantic evaluator can determine its truth value only after it has been syntactically analyzed. Similarly, a syntactic parser for mathematics must accept the string "2+3=7", and it is a non-syntactic question whether the statement expressed by this string is true or false.

The borderline between syntax and semantics is not always clear, however. For instance, should a syntactic parser accept the sentence "*I ate soup with a friend and a spoon*"? Purely on syntactic grounds, grammar can coordinate

(combine) the noun phrases "a friend" and "a spoon" within a prepositional phrase. However, more refined approaches to syntax will be able to block the coordination of very different nouns, serving very different roles in the sentence. Clever syntax may take over some of the systematic phenomena in semantics.

Similarly, we can exclude division by zero from the syntax of mathematics by prohibiting the substring `/0`.[7] But what about `1/(3-(6/2))=1`? In order to avoid division by zero in general, we have to refer to semantics, and we cannot do so purely on syntactic grounds. Hence, I am not very convinced that I would like to exclude strings such as `1/0>0` from the syntax of mathematics, either.

3 Understand the difference between "defining a language" and "describing a language"

You have all correctly understood that a string such as `1+1*2=9` is ambiguous by the grammar provided: the left-hand side has two parses, `[1+1]*2` and `1+[1*2]`, corresponding to two different values. It is only of secondary importance that in both cases the truth value of the entire "sentence" is false.

However, this expression is not ambiguous in the "language of mathematics". Anyone having passed the first few grades of primary school will know that the value of `1+1*2` is 3, and not 4. (The so-called "Polish notation" is a different language, and not the standard "language of mathematics".) Therefore, I have asked you to refine the original *grammar* so that it will match the "language of mathematics". That is a different task from re-designing the *language*.

Indeed, these two tasks reflect two different world views. The scholar in the humanities wishes to *describe* a phenomenon: given a language, let us find the grammar that best describes it. For instance, the grammar should not display more ambiguities than the amount of ambiguity in the language being described. The engineer, however, takes a more *prescriptive* approach: let us re-design the language in the simplest way so that it will not display any unwanted ambiguity. For instance, by enforcing parentheses around each operation. Then, `1+1*2>0` will not be a legitimate string anymore, but it should be either `(1+1)*2>0`, or `1+(1*2)>0`. Similarly, `10-2+3>0` will not be accepted, only `(10-2)+3>0` and `10-(2+3)>0`. But `1+1+1>0` will not be accepted, either.

This second approach works well if you are defining a novel standard, or a programming language, or something similar. However, most often such is not the case. Your task is to describe or process a given language, let it be a natural language or the "language of mathematics". Nobody will buy a machine translation software that allows users to only enter certain sentences, those that are easy to parse. Similarly, the language of mathematics developed in earlier centuries is sufficiently unambiguous, and therefore we should not re-design it. Rather, your task was to refine the grammar so that it should accept the string `1+1+1>0`, and unambiguously parse `1+1*2>0` or `10-2+3>0`.

[7] The language "`.* /0 .*`" is regular. Therefore its complement is also regular. So the intersection of its complement with a context free language results in a context free language.