

## Exercise C2, 3/18/05

### Part 1: Review “Context-Free”

The notion of “context-freeness” is extremely important in computer science, due to its tight connection with programming languages. If you understand this class, you understand the backbone of programming languages. So, here are some review questions. It is up to you how to use these. If you understand well by now, you do not need to write up your responses. But recall that passive and active knowledge are different.

Review questions

- A. What is the most important property of “context-freeness”?
- B. Why the constraint  $|LHS| = 1$  leads to a CFG (cf. unrestricted grammars)?
- C. What kind of strings cannot be specified by CFGs?
- D. What are the differences between top-down and bottom-up parsing?
- E. Why can PDAs capture all of CFLs?
- F. Why is the class of DCFLs smaller than that of CFLs?

### Part 2: Graphing Tool (Unit C2 Group Exercise 1)

When we analyze the growth rates of two functions, it would be useful to visualize how fast those functions would grow. The demo program shown in class (<http://www.tcnj.edu/~komagata/Graphing>) just does this. In this exercise, continue the in-class exercise and give a specification of the language the graphing tool would accept as a CFL. By trying various expressions, you should be able to tell the range of acceptable forms. Although you are encouraged to do this exercise from scratch, if you need something to start with, you can use the following partial information.

- Expression is a series of Terms connected with + or –
  - $\text{Exp} \rightarrow \text{Term}$
  - $\text{Exp} \rightarrow \text{Term} + \text{Exp}$
  - $\text{Exp} \rightarrow \text{Term} - \text{Exp}$
- Term is a series of Factors connected with \* or /
- Factor is Const, Factorial, Log, or Power
- Const
- Factorial  $\rightarrow$  “n!”
- Log
  - $\text{Log} \rightarrow$  “logn” | “log” Const “n” | “log(” Exp “)” | “log” Constant (“ Exp “)”
  - Note: “log” Const Const is illegal
  - Note: not including “log(” Constant “)(” Exp “)”
- Power
  - $\text{Power} \rightarrow$  (“n” | “(” Exp “)” | Const) “^” (“n” | “(” Exp “)” | Const)

### Part 3: Foxtrot

CFGs are quite useful for various things. In my Discrete Math courses, I used examples such as robot navigation, DNA analysis, dance choreography (sort of), etc. Here is another example from ballroom dancing. A collection of steps of “Foxtrot” can be described by the following CFG (only the rewrite rules are shown; other information can be derived from them):

Foxtrot	→ Basic	
Basic	→ Basic Basic	
Basic	→ <b>featherFinish</b> WeaveSeq Turn	
Basic	→ <b>promenadePosition</b> <b>naturalWeave</b> Turn	
WeaveSeq	→ $\epsilon$	(empty string, i.e., no steps)
WeaveSeq	→ <b>weave</b> WeaveSeq	
Turn	→ <b>threeStep</b> <b>heelTurn</b>	

Here, Foxtrot (start symbol), Basic, WeaveSeq, and Turn (beginning with an uppercase letter) are nonterminals, and the others are terminals. We will and you may abbreviate nonterminals/terminals using the first letter (case sensitive), e.g., W for WeaveSeq and w for **weave**.

- A. Would the following step sequence be acceptable according to the above description? If yes, show how such a sequence can be generated. If no, explain.

**f t h f w t h p n t h f w w w t h**

- B. Discuss the advantages/disadvantages of using (a) top-down parsing and (b) bottom-up parsing for this grammar. Note that you must contrast top-down vs. bottom-up approaches in general and should *not* be limited to some particular top-down or bottom-up mechanisms (i.e., do not discuss recursive descent, LL(1), LR(k), etc.).

**Note: You must clearly state some contrastive feature(s).**

- C. Would it be possible to generate exactly the same language (the step sequences generated by the above-mentioned CFG) using a regular grammar? Explain. In addition, argue whether this language is a context-free or regular [your argument should be very brief].

**Note: A regular grammar is like CFGs except that the RHS can have only one nonterminal at the beginning or at the end. For example, a regular grammar can have rules such as “A → a b C” and “A → A b c”, but *not* “A → a B c” or “A → a B C” (uppercase symbols are nonterminals). Regular grammars is equivalent to regular expressions and finite-state automata.**

**Hint: Examine several example step sequences and figure out the pattern of generated strings. Is the pattern similar to that of parenthesis matching, which requires a CFG?**

// End