Programming Languages (CS 421/521), Spring 2019
Assignment 01 (100 points)

Due 5:00 PM, Friday, 22 February 2019

This assignment must be type-written, with any diagrams rendered in electronic form. It will consist of a document answering a number of mathematical questions, and some code. Submit an archive containing both the document and (in a separate source folder) the code, by uploading it to the D2L dropbox by the given date and time.

[**1**] (*15 pts.*) Using the calculator language grammar found in the text (Figure 2.16, page 74), draw parse trees for each of the following assignment statements (one parse tree each):

(a) `x := x * b + 3`

(b) `a := b / c * d`

(c) `j := k + i * j - 7`

Note 01: In your parse trees, you can start with *stmt* as the root node, rather than beginning with the *program* symbol and associated ending characters.

Note 02: Remember that in the text, `id` and `number` are used as shorthand, and you can assume that you have additional rules for replacing those with actual variable identifiers and integers:

$$\texttt{id} \longrightarrow (\texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z})^+$$
$$\texttt{number} \longrightarrow (\texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9})^+$$

[**2**] (*10 points*) Consider the grammar defined by the following production rules, where S is the start symbol, other nonterminal symbols are in *italics*, `hey` and `ho` are the terminal symbols, and ϵ is the empty string.

$$\begin{array}{lcl} S & \rightarrow & X \mid Y \mid Z \\ X & \rightarrow & \texttt{hey}\, S\ \texttt{ho}\, S \\ Y & \rightarrow & \texttt{ho}\, S\ \texttt{hey}\, S \\ Z & \rightarrow & \texttt{heyho} \mid \texttt{hohey} \mid \epsilon \end{array}$$

(a) Write a rightmost derivation of the sentence `hoheyheyho`

(b) Is this grammar ambiguous? Why or why not? (Give as much detail as necessary.)

[**3**] (*10 pts.*) Consider a programming language in which variable identifiers are defined as such:

- An identifier is a combination of one of more of any of the following characters: {`x`, `y`, `0`, `1`, `_`}
- An identifier ***must not*** begin with a digit (`0` or `1`).

Thus, in this language, '`xyx_011`' is a valid identifier, but '`011_xyx`' is not.

Give a ***regular*** grammar for this language, and use that grammar to give a ***right-most*** derivation of the valid identifier `xyx_011`.

Reminder: a regular grammar is more restricted than a general context-free grammar, as every production rule must have one of three specific forms (see lecture notes for 4 February 2019).

[**4**] (*10 pts.*) Consider an alphabet with terminal symbols {`a`, `b`}. Write a context-free grammar in BNF (or EBNF) that will generate the set of all strings matching the pattern:

$$\texttt{a}^{2m}\texttt{b}^{n}\texttt{a}^{m} \quad (m \geq 1,\, n \geq 0)$$

That is, your grammar must produce strings consisting of a non-zero even number of `a`'s, followed by 0 or more `b`'s, followed by half as many `a`'s as in the original substring of `a`'s. Examples are:

`aaa`, `aaba`, `aabba`, `aabbba`, `aabbbba`, . . . , `aaaaaa`, `aaaabaa`, `aaaabbaa` . . .

You should also show, in as much detail as is necessary, that the grammar does what we want it to do (that is, argue for why your grammar generates *all* those strings, and *only* those).

[**5**] (*10 pts.*) Using the same alphabet {`a`, `b`} as before, write a context-free grammar to generate:

$$\texttt{a}^{m}\texttt{b}^{n} \quad (m \geq 2,\, n \geq 0,\, m \neq n)$$

That is, each string consists of 2 or more `a`'s, followed by any number of `b`'s, provided that the number of `a`'s and `b`'s are not equal. Examples are:

`aa`, `aaa`, `aaaa`, . . . , `aab`, `aabbb`, `aabbbb`, . . . , `aaab`, `aaabb`, `aaabbbb`, . . .

Again, write the grammar in BNF/EBNF, and show that it generates all and only correct strings.

[**6**] (*45 pts.*) This is a coding exercise. I prefer that you code the solution in Java, but if you wish to use another language, make sure to provide a `README` file with your code to explain how it should be run. Your program will run from the command-line, and will take a string as a command-line argument. The string will be an expression of basic arithmetic, involving the binary prefix operators (`+`, `-`, `*`, `/`); operands will be single integer digits or single-letter variable names. Every symbol within an expression is separated by a single space.

The language is defined by the following BNF grammar where *Exp* is the start symbol, all non-terminals are given in *Italics*, terminals are given in `typewritten font`, and ϵ is the empty string:

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \texttt{+}\ \mathit{Exp}\ \mathit{Exp} \mid \texttt{-}\ \mathit{Exp}\ \mathit{Exp} \mid \texttt{*}\ \mathit{Exp}\ \mathit{Exp} \mid \texttt{/}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Literal} \\
\mathit{Literal} & \rightarrow & \mathit{Var} \mid \mathit{Int} \\
\mathit{Var} & \rightarrow & \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \\
\mathit{Int} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9}
\end{array}
$$

Your program will take an expression from the language of this grammar and generate a **right-most derivation** of that expression. Expressions will be entered on the command line (with quotes around them if they contain more than one symbol, so that the result is sent to the program as a single input). Thus, your program, if called as follows:

```
java Derive "+ x - z 3"
```

will generate the following derivation for the expression, printing it to standard output:

```
Exp
+ Exp Exp
+ Exp - Exp Exp
+ Exp - Exp Literal
+ Exp - Exp Int
+ Exp - Exp 3
+ Exp - Literal 3
+ Exp - Var 3
+ Exp - z 3
+ Literal - z 3
+ Var - z 3
+ x - z 3
```

You can assume that only correct expressions of the language will be given as input. On incorrect expressions, your code may fail (or you may choose to throw a specific exception or otherwise display an error message to that effect).

You will hand in source-code for the program `Derive`. Whatever language it runs in should be such that the instructor can compile it, and can test it from the command line by giving it strings of input. Your code should be adequately commented, and function properly. If special instructions are needed for any step (compile or run), especially if you have used some language other than Java, then you must provide those instructions.

Note: Along with this document, the assignment materials include a text-file containing sample runs of the program. If yours is working correctly, you should be able to handle all such cases, and get exactly the same results.