COMP 5113 Programming Languages
Fall 2015
Homework 2

The second homework includes:

1. Refer to Figure 2.6 (automaton) and Figure 2.12 (scanner table), implement a scanner for the calculator language using C/C++ to recognize tokens in arithmetic expressions. The input will be an arithmetic expression or an assignment statement (composed of variables, integers, decimal numbers, (,), +, -, *, / and =), and the output reports each token in the expression as well as errors. Test your code using examples (e.g. EOL means the end of the expression)

   e.g., Execution 1:
   Please enter an arithematic expression: 20 * (40.5 - 15) /8
   NUMBER: 20
   TIMES
   LPREN
   NUMBER: 40.5
   MINUS
   NUMBER: 15
   RPREN
   DIVIDE
   NUMBER: 8
   EOL

   e.g., Execution 2:
   Please enter an arithematic expression: AA := 3 + B1
   VARIABLE: AA
   EQALS
   NUMBER: 3
   PLUS
   VARIABLE: B1
   EOL

   e.g., Execution 3:
   Please enter an arithematic expression: A3 ;
   VARIABLE: A3
   ERROR
   EOL

2. a. Using C/C++, write a recursive descent parser for the following context free grammar:

   $S \rightarrow A\ B$
   $A \rightarrow a\ C$
   $B \rightarrow -\ b \mid *\ (AB)$
   $C \rightarrow \varepsilon \mid +\ a\ C$

   e.g., Execution 1:

Please enter a program: a + a + a - b
Syntatically Correct!

e.g., Execution 2:
Please enter a program: a + b
Syntatically Wrong!

e.g., Execution 3:
Please enter a program: a * (a – b)
Syntatically Correct!

b. (**BONUS**) Using C/C++, implement a recursive descent parser for the calculator grammar show in Figure 2. 15 (grammar) and Figure 2.19 (LL(1) parsing table).

3. Chapter 2, ex. 5, 13, 14, 24.

Turn in both the softcopy and hardcopy of your documents. If you have more than one file, create a folder and put all files in, including subfolders. Your **name**, **your ID number** and **homework number** should be written on the upper right-hand corner of the top sheet of the hardcopy.

Date Due: The first program is due on Oct. 5, the rest are due on Oct. 12, Monday, 2015

Appendix: (textbook questions)

2.5 Starting with the regular expressions for *integer* and *decimal* in Example 2.3, construct an equivalent NFA (**only one** automaton, not two), the set-of-subsets DFA, and the minimal equivalent DFA. Be sure to keep separate the final states for the two different kinds of token (see the sidebar on page 58). You will find the exercise easier if you undertake it by modifying the machines in Examples 2.13 through 2.15.

integer → *digit digit* *
decimal → *digit** (.*digit* | *digit*.) *digit* *

2.13 Consider the following grammar:

stmt → *assignment*
→ *subr_call*
assignment → id := *expr*
subr_call → id (*arg_list*)
expr → *primary expr_tail*
expr_tail → *op expr*
→ ε
primary → id
→ *subr_call*
→ (*expr*)
op → + | - | * | /
arg_list → *expr args_tail*
args_tail → , *arg_list*
→ ε

(a) Construct a parse tree for the input string foo(a, b).
(b) Give a canonical (rightmost) derivation of this same string.
(c) Prove that the grammar is not LL(1).
(d) Modify the grammar so it is LL(1).

2.14 Consider the language consisting of all strings of properly balanced parentheses and brackets.
(a) Give LL(1) and SLR(1) grammars for this language.
(b) Give the corresponding LL(1) and SLR(1) parsing tables.
(c) For each grammar, show the parse tree for ([ ]([ ]))[ ](()).
(d) Give a trace of the actions of the parsers in constructing these trees.

2.24 Construct the CFSM for the id list grammar in Example 2.20 (page 65) and verify that it can be parsed bottom-up with zero tokens of look-ahead.

id_list → id *id_list_tail*
id_list_tail → , id *id_list_tail*
→ ;