CS 8803: Compilers : Theory and Practice
Homework II
Due: October 1st , 11:55 pm EST
Total Points : 100

Guidelines:

1. Georgia Tech Honor Code will be enforced.
2. Answers should be concise, complete and precise

Question I: Consider the following grammar (<list> is the start symbol) (**25 point**s)

<lexp> → <atom> | <list>
<atom> → number | identifier
<list> → (< lexp-seq>)
<lexp-seq> → <lexp> , <lexp-seq> | <lexp>

a. Left factor this grammar.
b. Construct First and Follow sets for the non-terminals of the resulting grammar.
c. Show that the resulting grammar is LL(1).
d. Construct the LL(1) parsing table for the resulting grammar.
e. Show the actions of the corresponding LL(1) parser, given the following input string:
(a , (b , (2)) , (c))
f. Find the average number of derivations done for the above input per token consumed.

Question II: Consider a LL(1) grammar with k rules. Answer the following questions: **(25 points)**

a. First construct a grammar for k=5 (that is, the grammar in question has 5 rules) which forces the LL(1) parsing algorithm to produce the highest number of derivations (expansions) to be done on the stack before a token is consumed. We will call this as worst case complexity of LL(1) parsing.
b. For this grammar, construct an input that will force such highest number of derivations.
c. If the number of tokens were to be n, what is the complexity (in terms of total number of derivations to consumer n tokens) of this grammar? Generalize your observation in (a) in terms of k and give a formula using n and k for worst case complexity of any LL(1) grammar

Question III (25 points)

Write an attribute grammar for determining the floating point value of a decimal number given by the following grammar. You will be parsing the string of digits using the following grammar and construct the decimal value at the end of the parse in dnum.val attribute (Hint: Use a *count* attribute to count and track the number of digits to the right of the decimal point.)

dnum → *num.num*
num → *num digit* | *digit*
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

<u>Question IV</u> **(25 points)**
With many grammars, operations may be interpreted differently, depending on whether they are floating point or strictly integer operations. Division, in particular, is quite different, depending on whether fractions are allowed. If not, division is often called a **div** operation, and the value of **5 / 4** is **5 div 4** = 1 (integer quotient of the division operation). If floating point division is meant, then **5 / 4** has value 1.2 (regular floating point quotient with zero remainder).

Suppose that a programming language requires mixed expressions to be promoted to floatingpoint expressions throughout, and the appropriate operations to be used in their semantics. Thus, the meaning of expression **5 / 2 / 2.0** (assuming left associativity of division) is 1.25, while the meaning of **5 / 2 / 2** is 1. (Notice that this is the not the same rule as the one used in C.) In the above example subexpression **5 / 2** is deemed as a floating point evaluation (**5.0 / 2.0** in fact) since the 2nd subexpression involved **2.0** is float thereby yielding **2.5/2.0** = **1.25** In case of 5/2/2 however since the 2nd subexpression is integer, the 1st one **5/2** is also deemed as an integer evaluation and thus it yields the result as **2/2** = **1**. To describe these semantics requires three attributes: a ***synthesized*** Boolean attribute ***isFloat*** which indicates if any part of an expression has a floating point value; an ***inherited*** attribute **etype**, with two values *int* and *float*, which assigns the type to each of the sub expressions and which depends on *isFloat* and is defined and passed on to sub expressions accordingly; and, finally, the computed *val* of each sub expression, which depends on the ***inherited etype***.

Now consider the following grammar:

$S \rightarrow exp$
$exp \rightarrow exp$ / exp | ***num*** | ***num.num***

And the following attribute equations:

<u>Grammar Rule</u>	**<u>Semantic Rules</u>**
S → exp	*exp.etype =* **if** *exp.isFloat* **then** *float* **else** *int* *S.val = exp.val*

$exp_1 \rightarrow exp_2$ / exp_3	$exp_1.isFloat$ = $exp_2.isFloat$ **or** $exp_3.isFloat$ $exp_2.etype = exp_1.etype$ $exp_3.etype = exp_1.etype$ $exp_1.val$ = **if** $exp_1.etype = int$ **then** $exp_2.val$ ***div*** $exp_3.val$ **else** $exp_2.val \,/\, exp_3.val$

exp → ***num***	*exp.isFloat* = **false**
	 exp.val =
	 if *exp.etype* = *int* **then** ***num***.*val*
	 else *Float(**num**.val)*

exp → ***num.num***	*exp.isFloat* = **true** *exp.val* = ***num.num***.*val*

Here, *Float(**num**.val)* indicates a function that converts the integer value ***num***.*val* into a floatingpoint value. Also, / is used for floatingpoint division and **div** for integer division.

If an attribute A depends on attributes B and/or C for its computation, the directed edges from attributes B or C to A depict the dependence of attribute A on attribute B and/or C. For each grammar rule, such an attribute dependence graph can be thus generated is called as the attribute dependence graph of a grammar rule.

a. Draw the attribute dependency graphs corresponding to each grammar rule and show it for the expression **5 / 2 / 2.0** using its parse tree.

b. The *isFloat*, *etype*, and *val* attributes in this example can be computed by two passes over the parse or syntax tree. The first pass computes the synthesized attribute *isFloat* by a postorder traversal. The second pass computes both the inherited attribute *etype* and the synthesized attribute *val* in a combined preorder/postorder traversal. Show how these two passes evaluate the attributes and the corresponding attribute computations that are required to compute the attributes on the syntax tree of **5 / 2 / 2.0**, including a possible order in which the nodes could be visited and the attribute values computed at each point.